**Full Stack Application Deployment with Docker, AWS EC2, and GitHub Actions — PART 2: Deploying to the EC2 Instance**

[[](https://medium.com/@ugurcanerdogan?source=post_page---byline--c0a206b51413---------------------------------------)](https://medium.com/@ugurcanerdogan?source=post_page---byline--c0a206b51413---------------------------------------)

[Uğurcan Erdoğan](https://medium.com/@ugurcanerdogan?source=post_page---byline--c0a206b51413---------------------------------------)

·

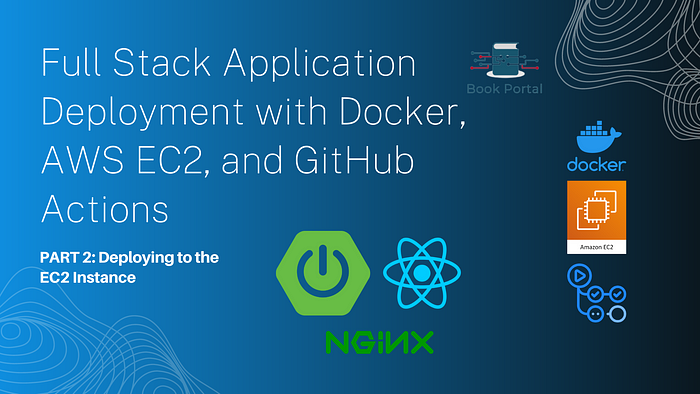
Follow

8 min read

·

Jul 2, 2023

50



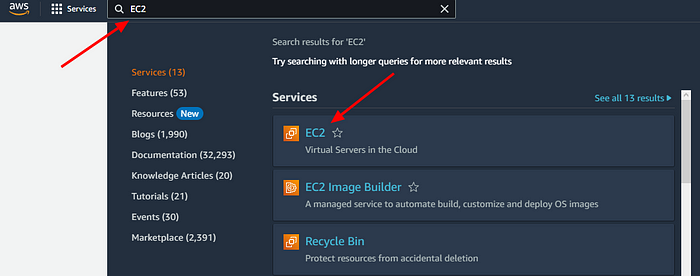
**Introduction:**

After a long break, we are continuing our series! It was a challenging time for the writer… Anyway, let’s continue with the second part of our guide, *“Full Stack Application Deployment with Docker, AWS EC2, and GitHub Actions.”*

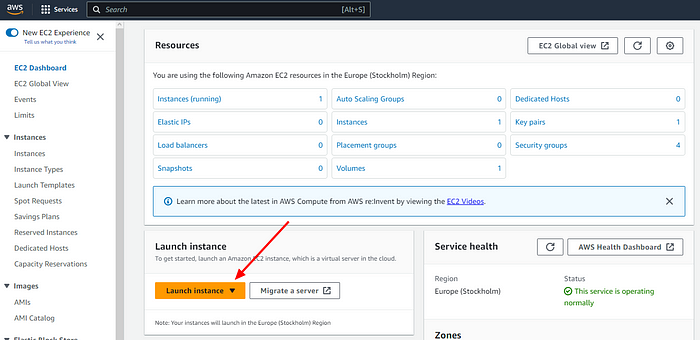
In this second part, we will deploy the images of our *BookPortal*project, which we containerized locally in the previous part, to an instance on the **AWS EC2** service. The BookPortal project, which will be moved to the cloud using a dedicated IPv4 address, will be accessible from anywhere. Let’s get started quickly!

**Creating and configuring the AWS EC2 instance:**

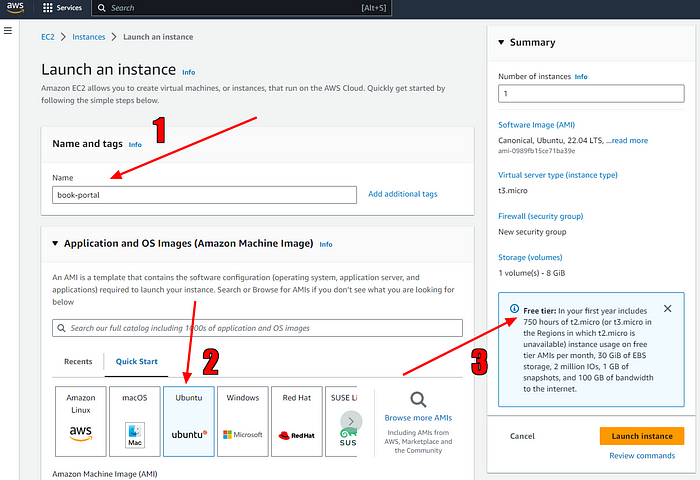
Firstly, we need to create an account on Amazon Web Services. It’s important to remember that new accounts can take advantage of the **free credits** available. After creating an account, we can proceed by searching for **EC2** in the search bar and accessing the relevant service.



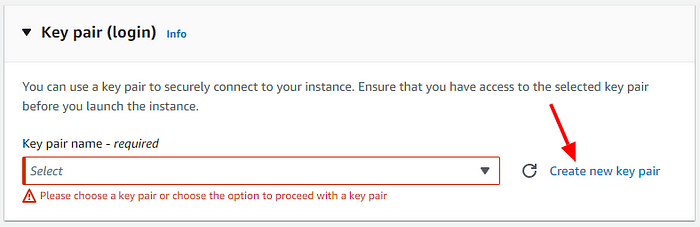
On the dashboard that appears, locate and click the “Launch Instance” button.

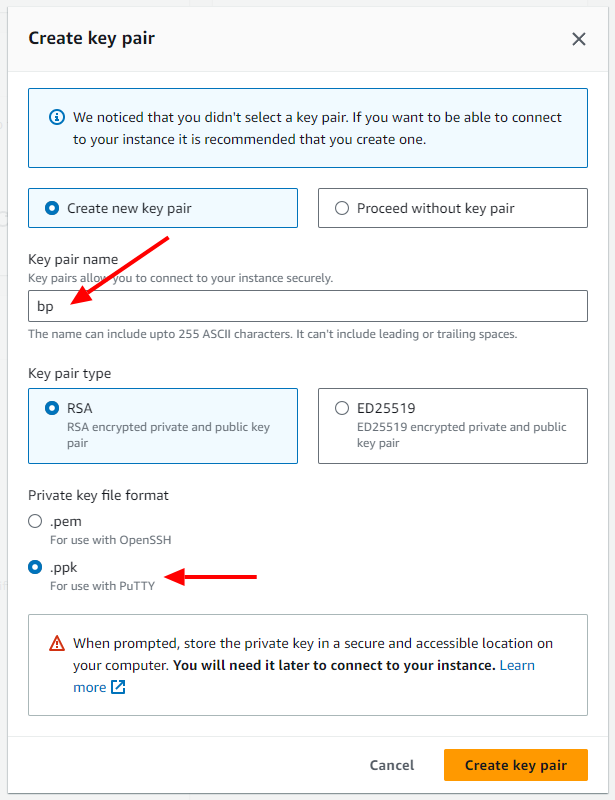


Afterward, in the first step, enter the name of your instance. In the second step, select an Ubuntu distribution for your instance, as indicated by the second arrow. With these settings, as shown in the third arrow, you can take advantage of the free tier.



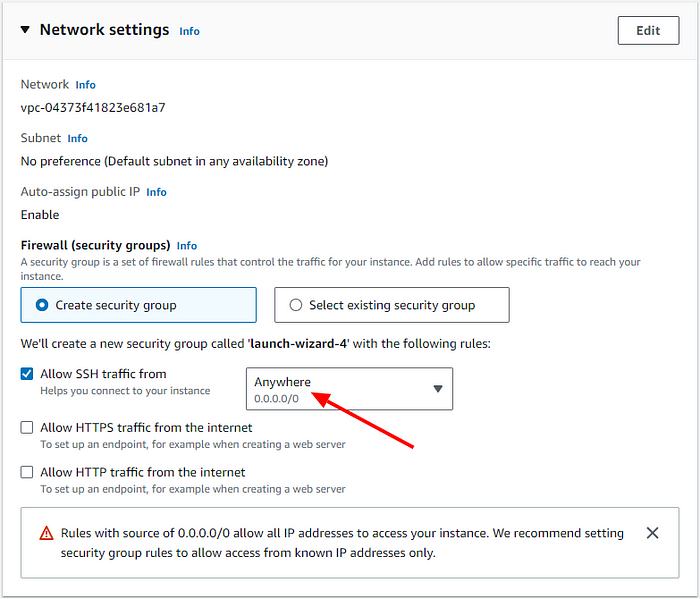
Now, we need to generate the key required for connecting to our instance via SSH to accommodate any future needs. This way, we will be able to use applications like PuTTY and establish an SSH connection using this key. We also need to ensure that this key is automatically assigned to the instance. Following the instructions below, we create a key with the extension .ppk and save it to our computer.





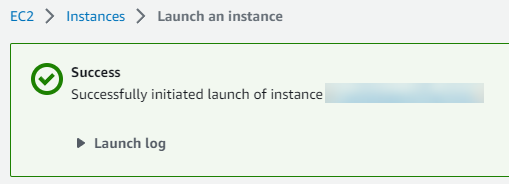
Lastly, in the remaining network settings below, configure your instance to be accessible from anywhere with this configuration.

*In projects where security is crucial, you can customize this part according to your needs.*



*Selecting the other 2 “allow” options may allow you to bypass the security settings in the next section. However, we did not select it here to see how to set the http and https ports specifically.*

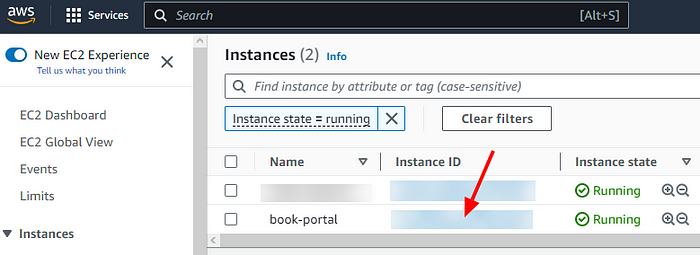
Alright, we made quite a few configurations. Now, let’s create our instance by clicking on the yellow “Launch Instance” button on the right.[🐣](https://emojipedia.org/hatching-chick/)



🎉

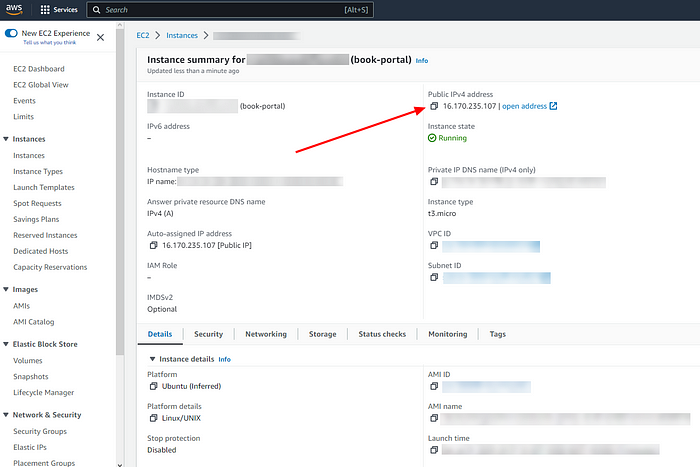
**Connecting to the instance:**

We proceed to the Instances menu on the EC2 service dashboard and click on the blue ID next to the instance we created to access the details of our instance.

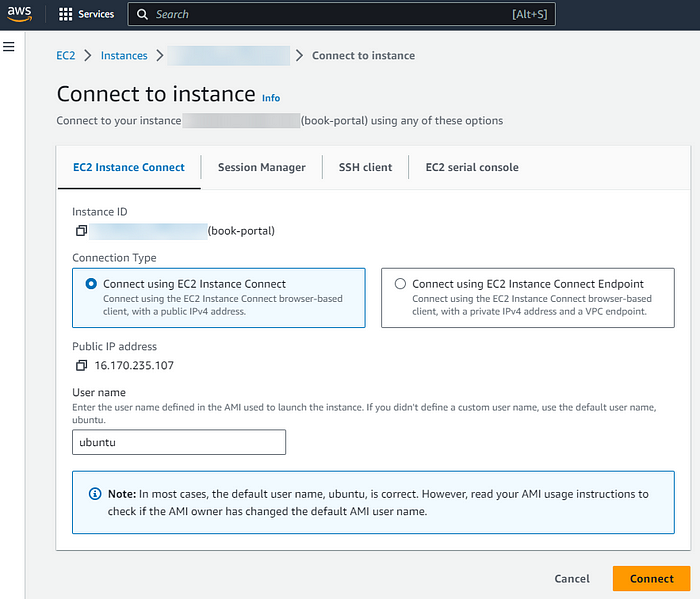


Hooray! We can now see the **public IP address** of our instance. However, if we navigate to this address at this stage, our browser will not display a proper result.

*Don’t forget that the public IP address can change when you reboot or stop-start the instance! This information can be useful if you want to acquire a domain name for this IP address in the future and need to set up the mapping correctly.*



Now, let’s establish an SSH connection to our instance by clicking on the “Connect” button located at the top right of this screen.



In the new tab that opens, we can see the terminal. Now we have to install docker on this baby instance.[🐤](https://emojipedia.org/baby-chick/)

**Performing the necessary installations on the instance:**

In this [well-explained article](https://kinsta.com/blog/install-docker-ubuntu/), you can review the descriptions of the commands we will use below. We will follow the same steps.

In the terminal that opens, we need to enter the following commands in order to install Docker and the other services.

sudo apt update  
sudo apt install ca-certificates curl gnupg lsb-release

sudo mkdir -p /etc/apt/keyrings  
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg  
sudo chmod a+r /etc/apt/keyrings/docker.gpg

echo "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/ubuntu $(lsb\_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

sudo apt install docker-ce docker-ce-cli containerd.io docker-compose-plugin

sudo groupadd docker  
sudo usermod -aG docker $USER

Checking the docker version:

docker -v

https://miro.medium.com/v2/resize:fit:331/1*ztnBxY03SjwKY4CKJDsi3Q.png

After “docker -v” command

docker-compose --version

https://miro.medium.com/v2/resize:fit:467/1*mj-bjJ39aDv7oVTVeu1gYw.png

After “docker-compose — version” command

Now, we are ready to use the docker-compose file and the images created in the previous step. Let’s give it a try.

**Pushing the local images to the DockerHub:**

Remember that in the docker-compose file we created, we were accessing Dockerfiles through the file path specified in the build line to build containers using them. However, since these Dockerfiles and project files are not present on our EC2 instance, we need to push these images to DockerHub and then pull them from the DockerHub repository to our instance. As a result, we will no longer need to manually transfer project files to the instance, which will greatly ease the setup of our CI/CD infrastructure. Additionally, this approach will simplify the CI/CD process by eliminating the need for manual file transfers. By pushing the project’s Docker images to DockerHub and pulling them from there, we can streamline the deployment and updates of our application on the instance.

You can use the following guidelines to easily push the Docker images created locally to DockerHub and view them among your DockerHub repositories:

**[Push Docker Image To Docker Hub](https://medium.com/codex/push-docker-image-to-docker-hub-acc978c76ad?source=post_page-----c0a206b51413---------------------------------------" \t "_blank)**

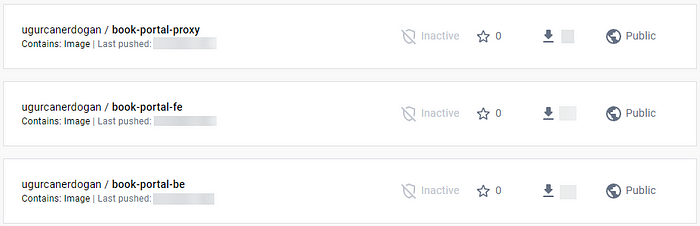
[Create Docker hub account and push Docker image.](https://medium.com/codex/push-docker-image-to-docker-hub-acc978c76ad?source=post_page-----c0a206b51413---------------------------------------" \t "_blank)

[medium.com](https://medium.com/codex/push-docker-image-to-docker-hub-acc978c76ad?source=post_page-----c0a206b51413---------------------------------------" \t "_blank)

**[Pushing and Pulling to and from Docker Hub](https://jsta.github.io/r-docker-tutorial/04-Dockerhub.html?source=post_page-----c0a206b51413---------------------------------------" \t "_blank)**

[Edit description](https://jsta.github.io/r-docker-tutorial/04-Dockerhub.html?source=post_page-----c0a206b51413---------------------------------------" \t "_blank)

[jsta.github.io](https://jsta.github.io/r-docker-tutorial/04-Dockerhub.html?source=post_page-----c0a206b51413---------------------------------------" \t "_blank)



[https://hub.docker.com/repositories/](https://hub.docker.com/repositories/ugurcanerdogan){your-dockerhub-username}

Now the images of our local BookPortal project are in DockerHub. Now we will save our docker-compose file on our EC2 instance, on which we will make minor changes, and run this file.

First of all, we create a docker-compose.yml file by following the commands below and fill its content.

https://miro.medium.com/v2/resize:fit:462/1*9XG1hNE1vqRGsTFC400iBw.png

We will add the content of our docker-compose file, which we will change the content of, to the editor opened after the nano command.

**The initial version of our docker-compose file was as follows:**

version: "3"  
services:  
 book-portal-be:  
 build: ./book-portal-be  
 container\_name: "book-portal-be"  
  
 book-portal-fe:  
 build: ./book-portal-fe  
 container\_name: "book-portal-fe"  
 depends\_on:  
 - book-portal-be  
 proxy:  
 build: ./nginx  
 container\_name: "book-portal-proxy"  
 restart: always  
 ports:  
 - "80:80"

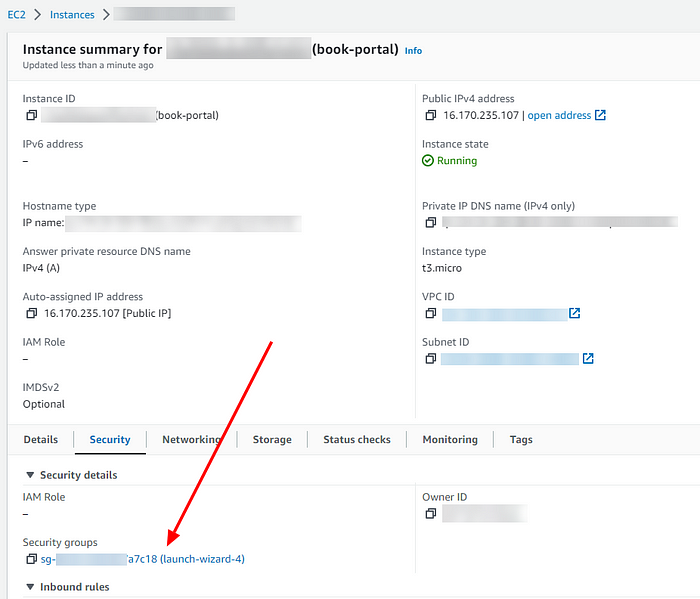
We will modify the “build” lines in the file and replace them with the “image” command to pull directly from DockerHub repositories.

***Do not forget the adjust the repository names according to your DockerHub account and name of the project components!***

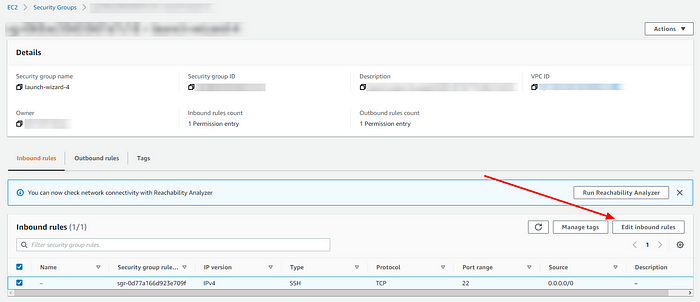
**Now the file should look like this:**

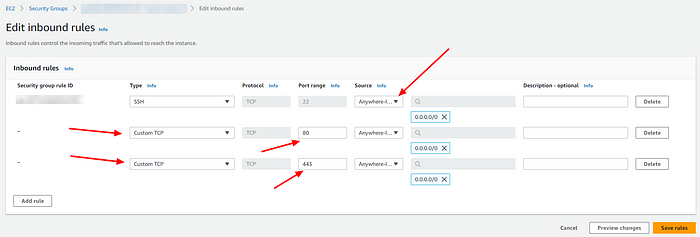
version: "3"  
services:  
 book-portal-be:  
 image: {dockerhub-username}/book-portal-be:latest  
 container\_name: "book-portal-be"  
  
 book-portal-fe:  
 image: {dockerhub-username}/book-portal-fe:latest  
 container\_name: "book-portal-fe"  
 depends\_on:  
 - book-portal-be  
 book-portal-proxy:  
 image: {dockerhub-username}/book-portal-proxy:latest  
 container\_name: "book-portal-proxy"  
 restart: always  
 ports:  
 - "80:80"

Before we create our containers, we need to allow requests from some ports in the networking settings of our EC2 instance. For this, we click on the Security tab at the bottom of the instance details screen and proceed to the blue link under security groups.

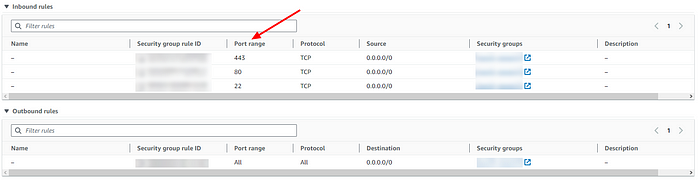


From the inbound settings here, we need to set the following ports accordingly.





Our final configs should look like this:



Now, that magical command is with us once again! We are running our saved docker-compose file with this command and expecting the images to be pulled from DockerHub.

https://miro.medium.com/v2/resize:fit:411/1*qqcPSSuuAkrOJ3z0oja_vQ.png

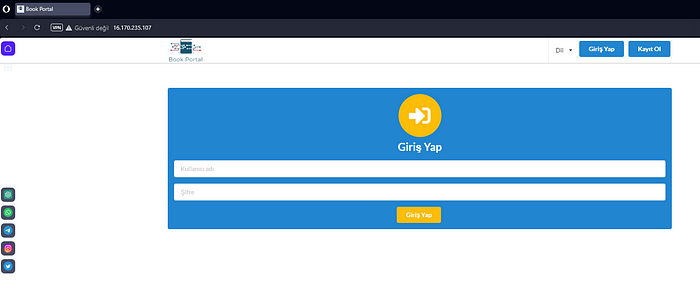


It’s working! Now let’s try to access this instance with the public IPv4 address:

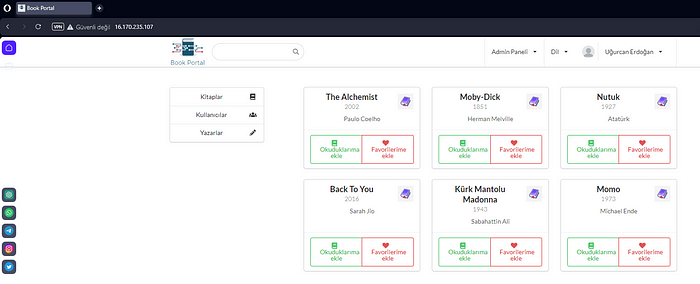
**[Book Portal](http://16.170.235.107/?source=post_page-----c0a206b51413---------------------------------------" \t "_blank)**

[Edit description](http://16.170.235.107/?source=post_page-----c0a206b51413---------------------------------------" \t "_blank)

[16.170.235.107](http://16.170.235.107/?source=post_page-----c0a206b51413---------------------------------------" \t "_blank)



[🐓](https://emojipedia.org/rooster/)



Here we go! Now we can access the project via a specific URL. Thus, we will be able to access the project through the instance without the need to run it in our local.

**Conclusion:**

In this part, we pushed our previously dockerized images to DockerHub and pulled these images to our AWS EC2 instance using the Container Registry. After modifying the docker-compose.yml file, we containerized the images and were able to access our project through the public IPv4 address of the EC2 instance where we made networking configurations.

In the next part, we will see how we can automatically reflect the changes created by the code we pushed to GitHub onto our EC2 instance. This way, we will have implemented CI/CD. See you in the next part!

*Our project has grown up and flown out of the nest. :’)*