PrestoChango Project

**\*Precursor\***

First, I want to say I made a conscious decision to attempt to complete the project with technologies I virtually have no work experience with. I wanted to challenge myself to see how quickly I could learn both Docker and AWS platforms along with also trying to automate against them. Sorry to say I was not able to put together a fully working build and deployment “pipeline” in the time that is recommended for this exercise. With that said, I was able to create some infrastructure, and learn the commands in order to enable me to automate them in a process (more on that in my “What if Jim had more time” section) and was able to hit my presto container via public IP. What I created was an EC2 instance instead of an ECS Container instance to complete the project. I started with an ECS instance and had a container running but struggled to automate that process. I needed more time to learn how to automate against the ECS Container instance to better create the pipeline I had in mind.

With the EC2 instance created, I was able to install Docker and login to me ECR repo which contains the Presto image I was able to build and push up to AWS.

**\*Initial Task for Pipeline Creation\***

As stated early, I created an EC2 instance following this guideline to create a linux host and install docker: (Not ideal, but accomplishes my “MVP” pipeline)

<https://www.youtube.com/watch?v=IDUyROEXnNA>

I wanted to take a different approach from selecting a particular branch to initiate a build, and instead just allow the developer to perform a commit which would trigger “build” steps to create the docker image. The docker file itself contains the version of the image that would be ultimately built. The build process would inspect the docker file for the version number and execute a command to build the image and tag it with either the version number itself or the version number plus branch name (ex. – tag=312-e.7-mybranch). Once built, the image is then pushed to the ECR repo, where then any image could be deployed by specify the “version” tag.

I didn’t want to do a lot of manipulation at the code repo level and rather use webhooks or triggers to initiate the docker image build, push to repo, then deploy based on image tag, which could be an input variable value. This would constitute an “MVP” (minimal viable product) for the developers to leverage by just having to push a commit, and then inputting what version image to deploy. The deployment and build would be handled by an orchestrator where variables can be stored and substituted in the scripts to select the “version” along with type environment specific values(preprod and prod).

My “MVP” state is a manual code pull from the repo to a directory from which I then execute the tasks to build, push, and then deploy the Presto Container to AWS ECR and EC2. I didn’t have enough time to complete the automate Octopus tasks, but I have all code that the tasks would run uploaded in my repo for review and listed here in this document.

Prerequirements:

* Docker client installed on local workstation
* PowerShell Module “Posh-SSH” installed for connection to EC2 instance

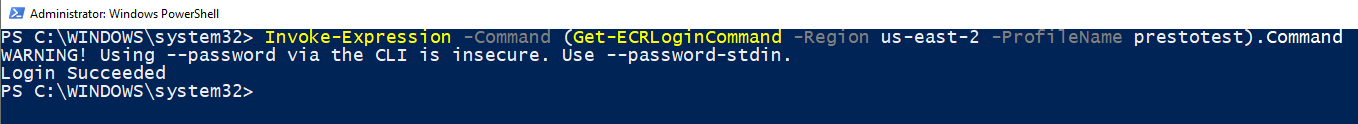
## Quick Overview of Steps:

1. Clone Presto repo to local workstation
2. Trigger Octopus Deploy to build and deploy the Presto Container.

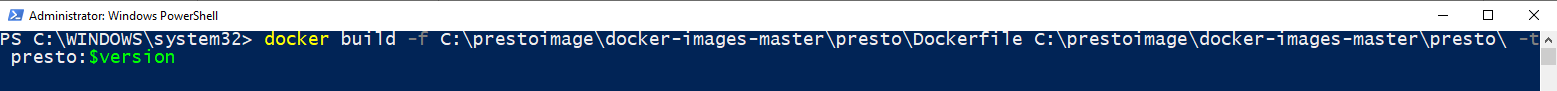
* Octopus runs command to log into AWS ECR registry – Task 1
* Task runs to build docker image from local repo path – Task 2
* Task runs to tag the image with variable substitute – Task 3
* Task runs to push to AWS ECR repo with variable tag – Task 4
* Task runs to log into AWS ECR repo from EC2 instance – Task 5
* Task runs to pull presto image with variable version tag – Task 6
* Task runs to run the docker container specifying ports and variable version – Task 7
* Task runs “Invoke-WebRequest” against EC2 Container instance expecting statuscode “200” for successful deployment. If not “200” then deployment fails. (Health-check test) – Task 8

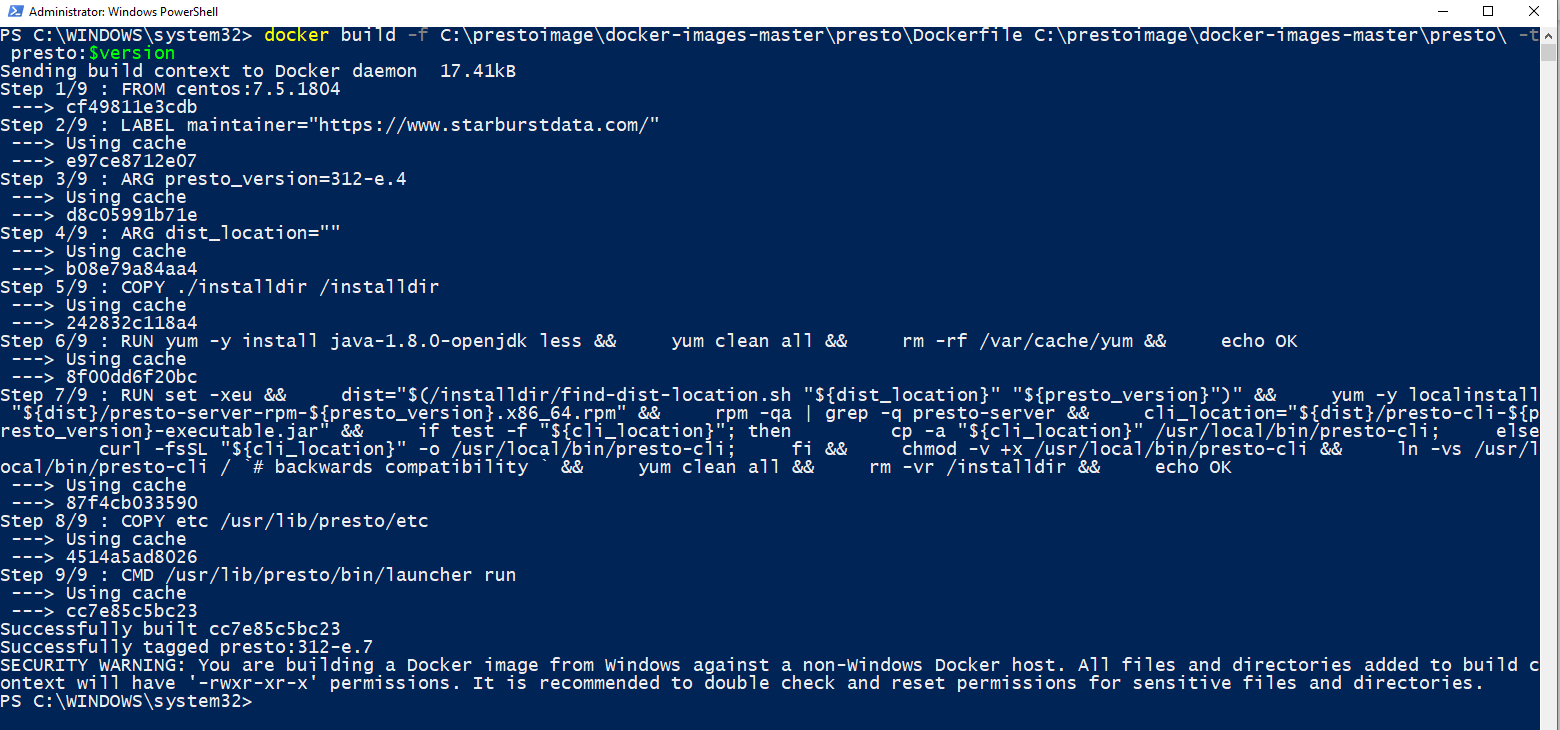
Screenshots of each task fun manually:

Task 1: (Aws Credential was built previously to this step)

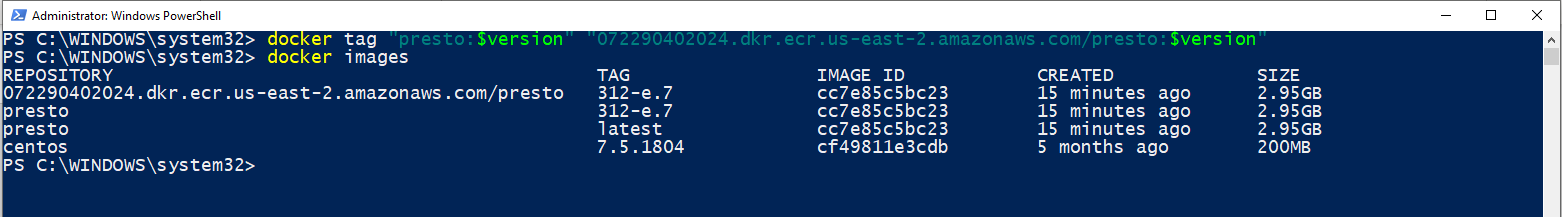


Task 2:

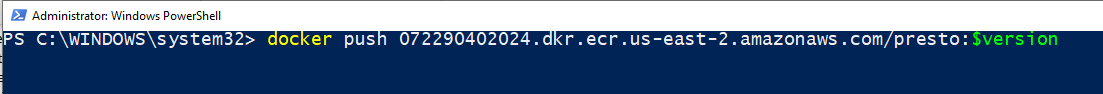




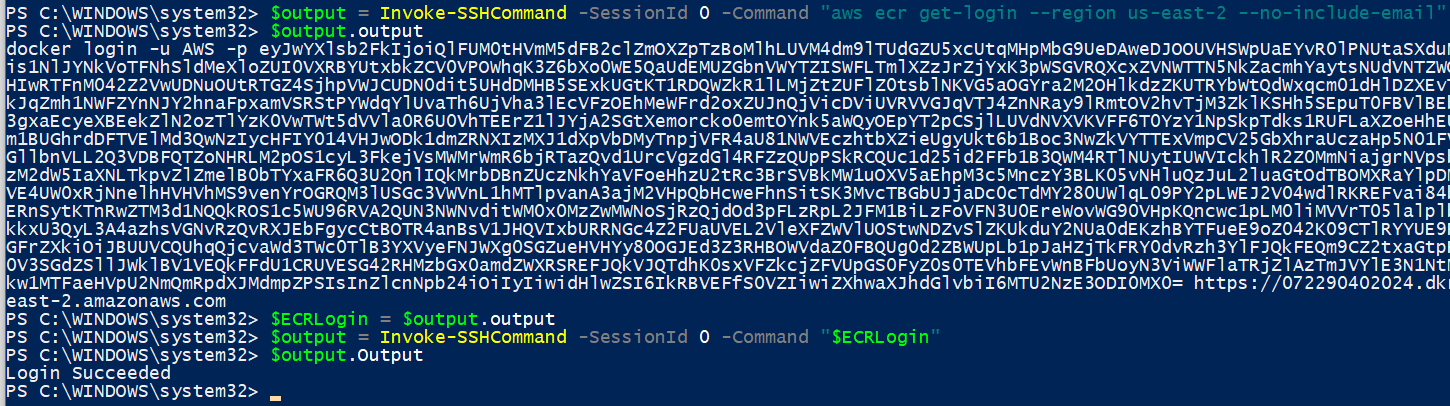
Task 3:



Task 4:



Task 5: This uses the Posh-SSH module and a session is established before running this task. Will elaborate more on this further in “extend further” section:



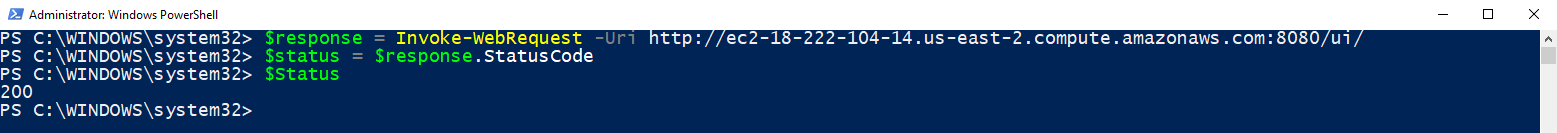
Task 6:



Task 7:

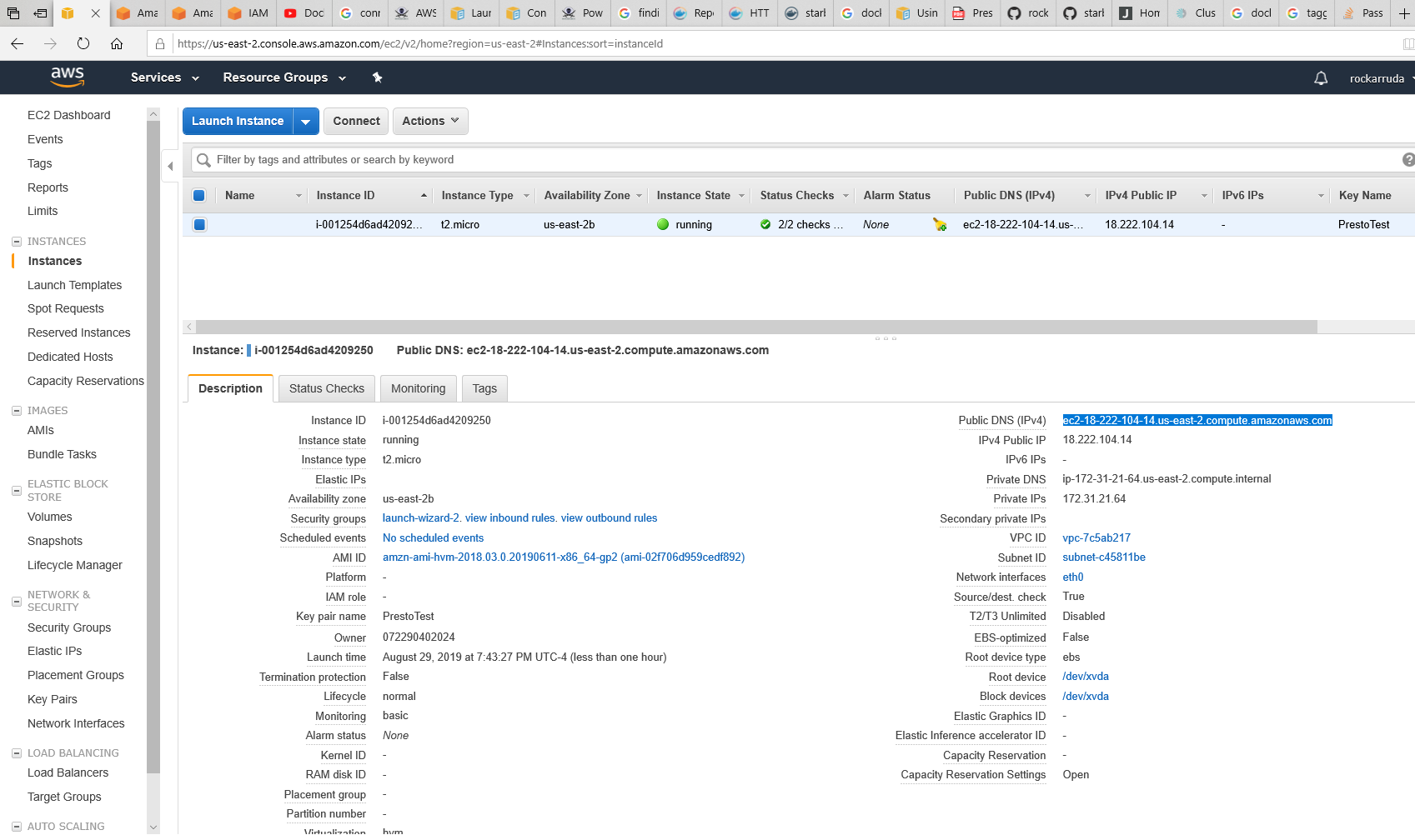


Task 8:

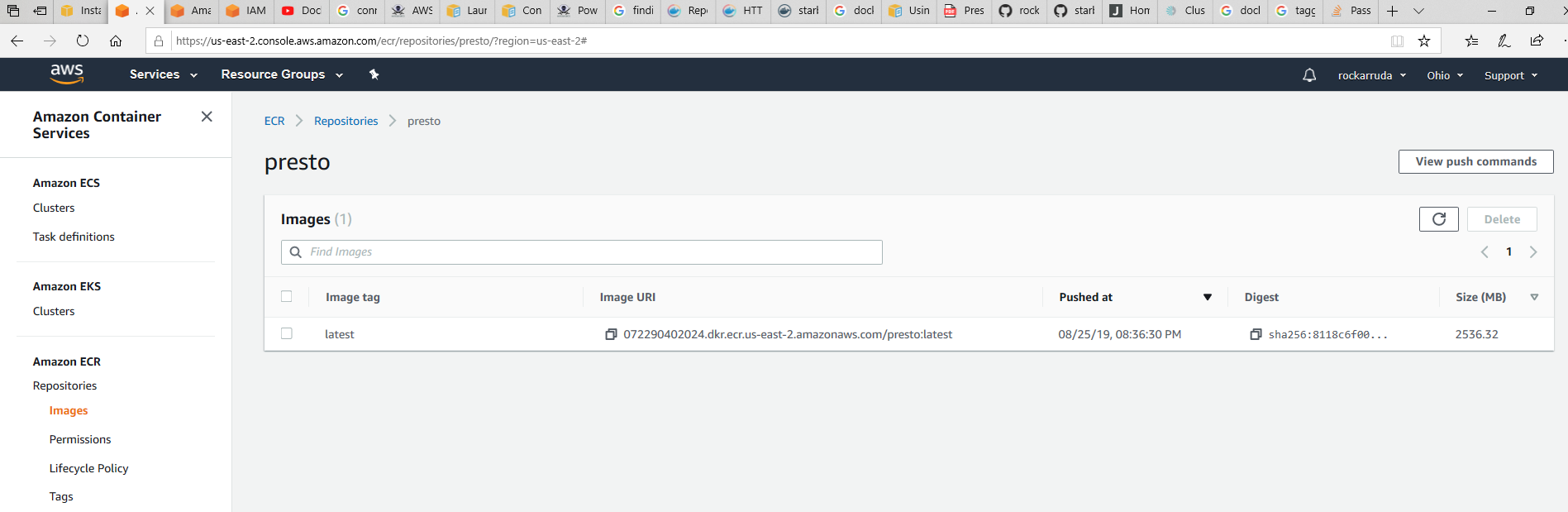


**\*Evidence\***

**AWS EC2 Instance:**

****

**ECR Repo:**

****

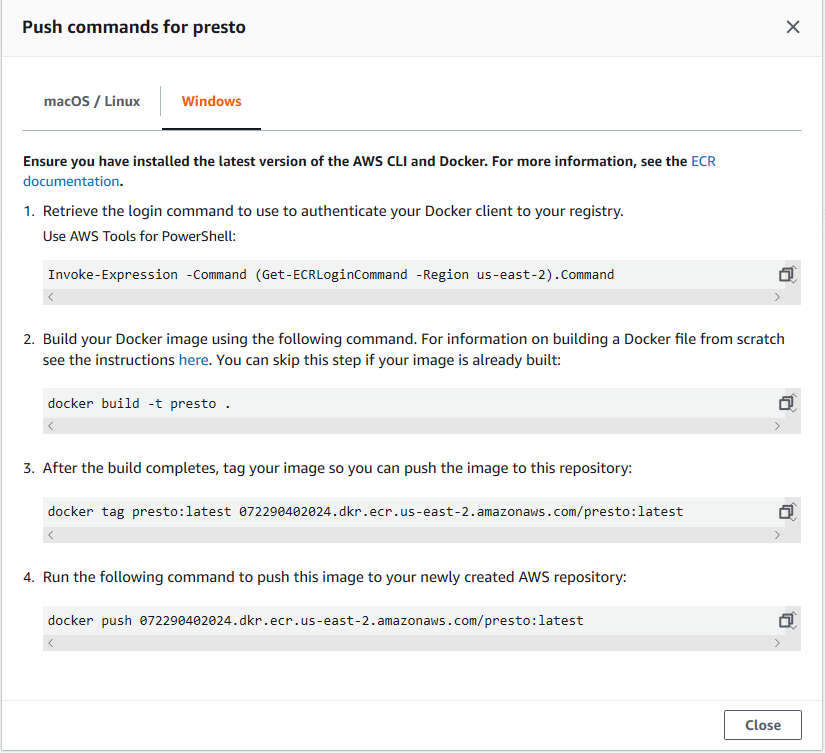
**Amazon Resources: (Basically A LOT of Amazon docs) Some are below**

[**https://docs.aws.amazon.com/AmazonECR/latest/userguide/ECR\_AWSCLI.html**](https://docs.aws.amazon.com/AmazonECR/latest/userguide/ECR_AWSCLI.html)

[**https://docs.aws.amazon.com/powershell/latest/userguide/pstools-appendix-sign-up.html**](https://docs.aws.amazon.com/powershell/latest/userguide/pstools-appendix-sign-up.html)

[**https://docs.aws.amazon.com/AmazonECR/latest/userguide/docker-basics.html**](https://docs.aws.amazon.com/AmazonECR/latest/userguide/docker-basics.html)

[**https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-network-security.html?icmpid=docs\_ec2\_console**](https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-network-security.html?icmpid=docs_ec2_console)

****