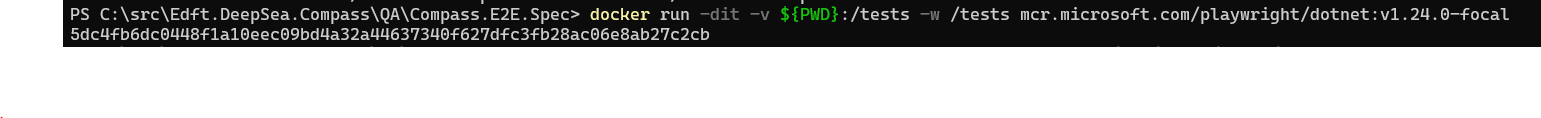
Docker

# Approach: Use shared data between host and container via volumes

Good for when we need to build something in the host outside of the container, then use the data for the container. E.g.: in this case, we build test solution in the host. The container then can use the build artifacts directly from the bond volume.

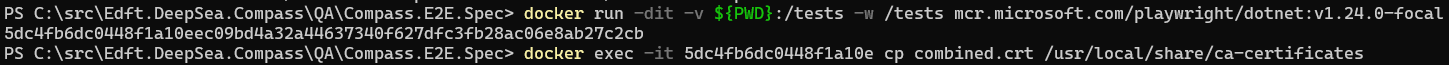
docker run -dit -v ${PWD}:/tests -w /tests mcr.microsoft.com/playwright/dotnet:v1.24.0-focal

docker run -v ${pwd}:/tests mounts the current host directory (i.e.: ${pwd}) into /tests in the container

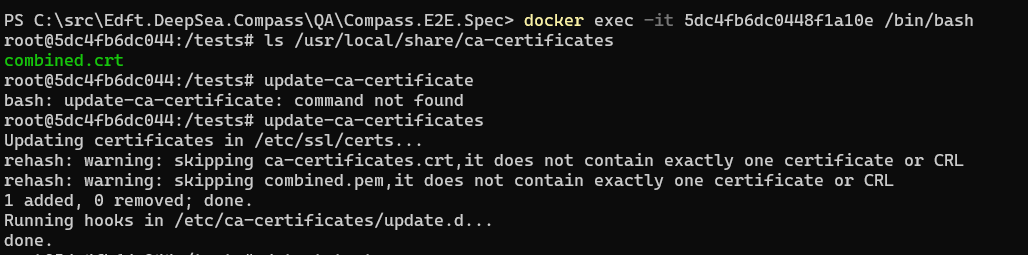
-w /tests sets the working folder as /tests, i.e.: the current host directory that was mounted

docker run -dit runs the container detached in the background, keeps input open and a suedo shell

Then can use docker exec to run shell commands inside the docker

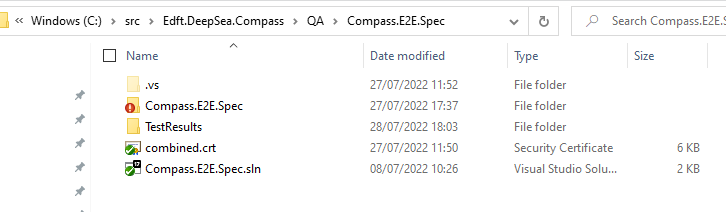


Get a shell into the container with docker exec -it {dockerid} /bin/bash



# To run playwright .net in a linux container:

Locally & manually

1. build the tests, navigate to the root folder where the solution file is: 

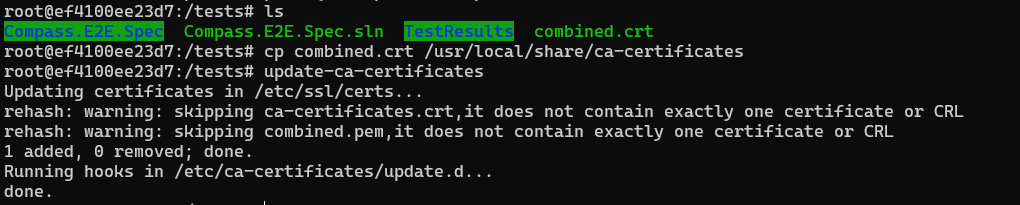
2. run the official playwright docker image in detached mode and bind current directory to the container volume

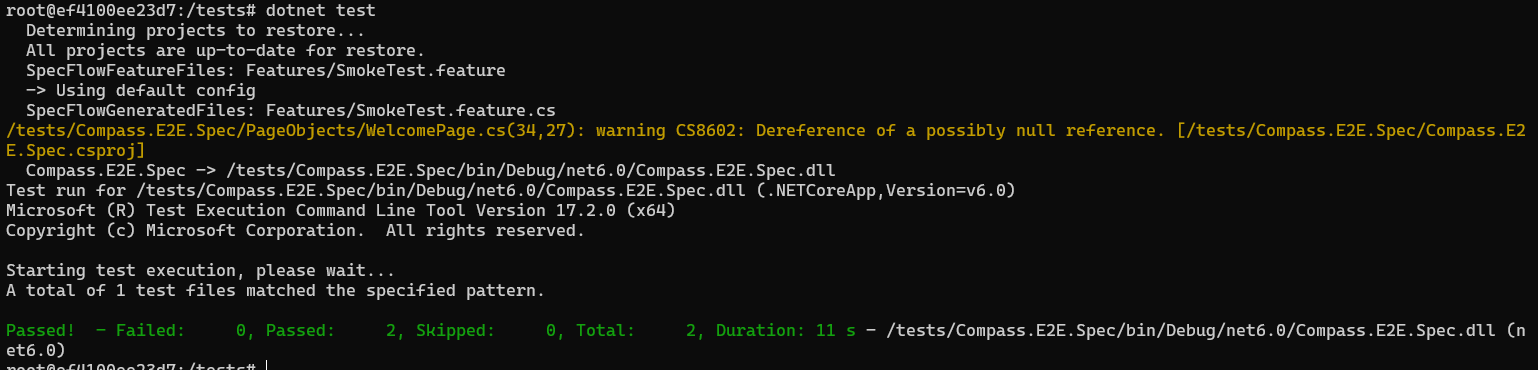
docker run -dit -v ${PWD}:/tests -w /tests mcr.microsoft.com/playwright/dotnet:v1.24.0-focal

3. get into container shell

docker exec -it ef4100ee23d79faa62085867 /bin/bash

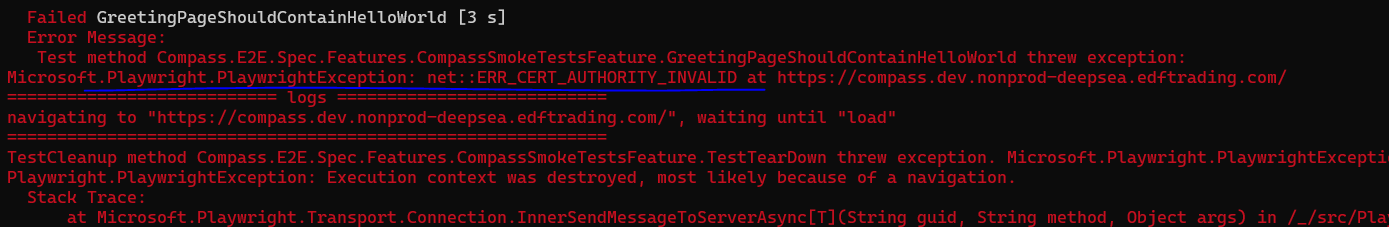
4. copy & update the CA cert



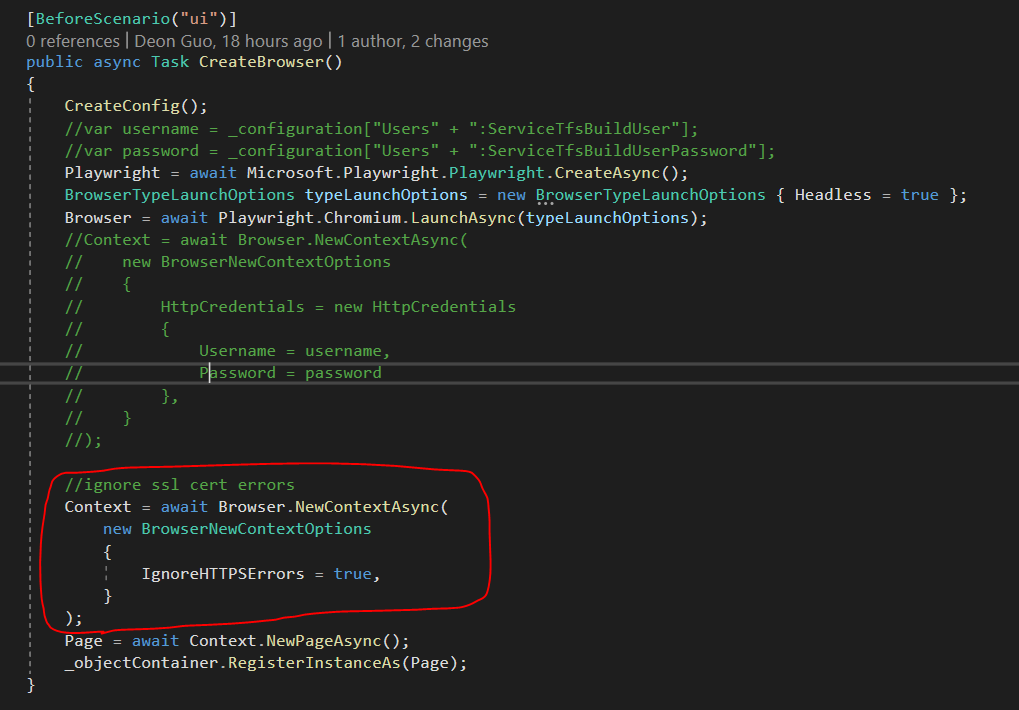
5. run tests

# Toubleshoot:

## Microsoft.Playwright.PlaywrightException: net::ERR\_CERT\_AUTHORITY\_INVALID at <https://compass.dev.nonprod-deepsea.edftrading.com/>



Solution 1:

Configure Playwright to bypass https errors

# Create Dockerfile – to build image

FROM – base image

WORKDIR – container’s working directory

COPY – copy project files needed

Do this in stages so that it accelerates building in later runs, e.g.:

Copy .sln, .csproj files first, do dotnet restore

Then copy all other project files

RUN – the state after running the command will be saved into the image

ENTRYPOINT – the main command the container will execute when it’s run

CMD – supply arguments to ENTRYPOINT. Can be overwritten by user input run-time arguments

# WORKDIR

The **WORKDIR command** is used to define the *working directory* of a Docker container at any given time. The command is specified in the Dockerfile.

Any RUN, CMD, ADD, COPY, or ENTRYPOINT command will be executed in the specified working directory.

If the WORKDIR command is not written in the Dockerfile, it will automatically be created by the Docker compiler. Hence, it can be said that the command performs mkdir and cd implicitly.

Here’s a sample Dockerfile in which the working directory is set to /project:

FROM ubuntu:16.04

WORKDIR /project

RUN npm install

If the project directory does not exist, it will be created. The RUN command will be executed inside project.

COPY

COPY has two forms:

COPY <src>... <dest>

COPY ["<src>",... "<dest>"] (this form is required for paths containing whitespace)

The COPY instruction copies new files or directories from <src> and adds them to the filesystem of the container at the path <dest>.

Multiple <src> resource may be specified but they must be relative to the source directory that is being built (the context of the build).

If multiple <src> resources are specified, either directly or due to the use of a wildcard, then <dest> must be a directory, and it must end with a slash /.

## RUN, CMD and ENTRYPOINT command in Dockerfile

[RUN](https://docs.docker.com/engine/reference/builder/#run) is an image build step, the state of the container after a RUN command will be committed to the container image. A Dockerfile can have many RUN steps that layer on top of one another to build the image.

[CMD](https://docs.docker.com/engine/reference/builder/#cmd) is the command the container executes by default when you launch the built image. A Dockerfile will only use the final CMD defined. The CMD can be overridden when starting a container with docker run $image $other\_command.

[ENTRYPOINT](https://docs.docker.com/engine/reference/builder/#entrypoint) is also closely related to CMD and can modify the way a container is started from an image.

**RUN** - RUN instruction allows you to install your application and packages required for it. It executes any commands on top of the current image and creates a new layer by committing the results. Often you will find multiple RUN instructions in a Dockerfile.

**CMD** - CMD instruction allows you to set a default command, which will be executed only when you run container without specifying a command. If Docker container runs with a command, the default command will be ignored. If Dockerfile has more than one CMD instruction, all but last  
CMD instructions are ignored.

The ENTRYPOINT specifies a command that will always be executed when the container starts.

The CMD specifies arguments that will be fed to the ENTRYPOINT.

If you want to make an image dedicated to a specific command you will use ENTRYPOINT ["/path/dedicated\_command"]

Otherwise, if you want to make an image for general purpose, you can leave ENTRYPOINT unspecified and use CMD ["/path/dedicated\_command"] as you will be able to override the setting by supplying arguments to docker run.

For example, if your Dockerfile is:

FROM debian:wheezy

ENTRYPOINT ["/bin/ping"]

CMD ["localhost"]

Running the image without any argument will ping the localhost:

$ docker run -it test

PING localhost (127.0.0.1): 48 data bytes

56 bytes from 127.0.0.1: icmp\_seq=0 ttl=64 time=0.096 ms

56 bytes from 127.0.0.1: icmp\_seq=1 ttl=64 time=0.088 ms

56 bytes from 127.0.0.1: icmp\_seq=2 ttl=64 time=0.088 ms

^C--- localhost ping statistics ---

3 packets transmitted, 3 packets received, 0% packet loss

round-trip min/avg/max/stddev = 0.088/0.091/0.096/0.000 ms

Now, running the image with an argument will ping the argument:

$ docker run -it test google.com

PING google.com (173.194.45.70): 48 data bytes

56 bytes from 173.194.45.70: icmp\_seq=0 ttl=55 time=32.583 ms

56 bytes from 173.194.45.70: icmp\_seq=2 ttl=55 time=30.327 ms

56 bytes from 173.194.45.70: icmp\_seq=4 ttl=55 time=46.379 ms

^C--- google.com ping statistics ---

5 packets transmitted, 3 packets received, 40% packet loss

round-trip min/avg/max/stddev = 30.327/36.430/46.379/7.095 ms

For comparison, if your Dockerfile is:

FROM debian:wheezy

CMD ["/bin/ping", "localhost"]

Running the image without any argument will ping the localhost:

$ docker run -it test

PING localhost (127.0.0.1): 48 data bytes

56 bytes from 127.0.0.1: icmp\_seq=0 ttl=64 time=0.076 ms

56 bytes from 127.0.0.1: icmp\_seq=1 ttl=64 time=0.087 ms

56 bytes from 127.0.0.1: icmp\_seq=2 ttl=64 time=0.090 ms

^C--- localhost ping statistics ---

3 packets transmitted, 3 packets received, 0% packet loss

round-trip min/avg/max/stddev = 0.076/0.084/0.090/0.000 ms

But running the image with an argument will run the argument:

docker run -it test bash

root@e8bb7249b843:/#

FROM mcr.microsoft.com/playwright/dotnet:v1.24.0-focal AS base

WORKDIR /tests

# copy root cert

# error: COPY ["../../SslCertificates/\*.crt", "/usr/local/share/ca-certificates/"]

# why "../../SslCertificates/\*.crt" refered to "/var/lib/docker/tmp/buildkit-mount1479438691/SslCertificates"?

# because it can only access subfolders of the host system?

COPY ["\*.crt", "/usr/local/share/ca-certificates/"]

# trust the cert

RUN update-ca-certificates

# restore the packages first

COPY ["\*.sln", "."]

COPY ["Compass.E2E.Spec/\*.csproj", "./Compass.E2E.Spec/"]

RUN dotnet restore

# copy all project files into container

COPY . .

RUN dotnet test -c Release --no-restore

# set the command for the container to run when started

# this command will always be run

ENTRYPOINT [ "dotnet" ]

# CMD feeds arguments into ENTRYPOINT command

# this command can be overwritten when starting the container,e.g: docker run -it containerName debug?

CMD ["test --no-restore --logger:trx -r ./results/"]

# to build the container locally: docker build -t compasse2e

# to run the container: docker run compasse2e

# to keep it up for shell: docker run -dit compasse2e

# container shell: docker exec -it [id] /bin/bash

# however the container exits as soon as the cmd was executed... can't shell in to check test result file

# next steps:

# how to keep container running even after cmd? so I can shell in

# integrate into pipeline

# publish test results

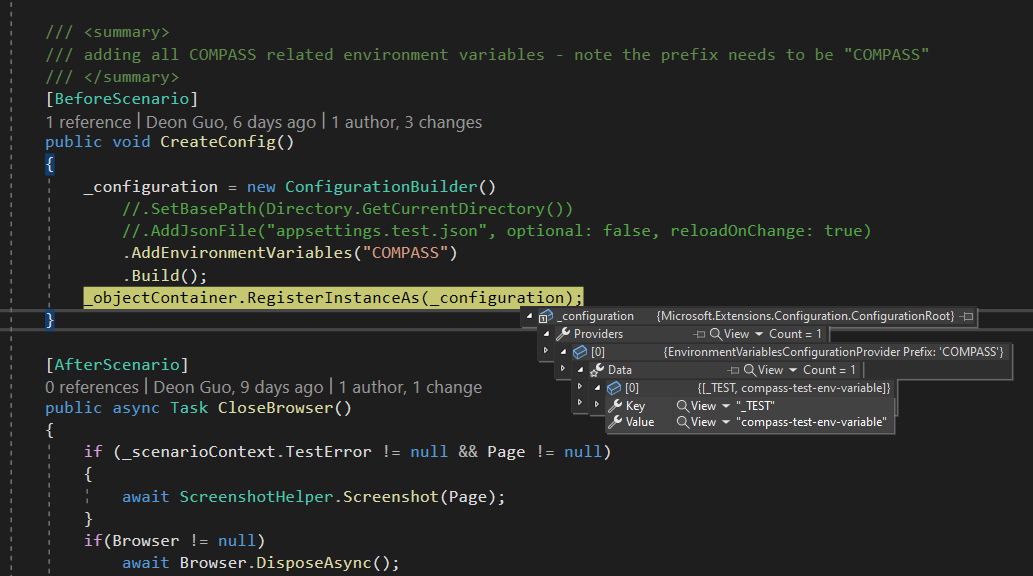
# Configure to use environment variables with Docker:

## 1. Set environment variables

### a. on a local machine

With Linux it’s easier. Below is how to Set environment variables in Windows:

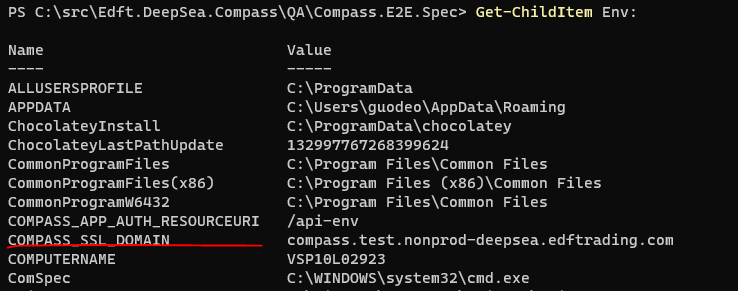
**setx COMPASS\_TEST 'compass-test-env-variable'**



Note:

Setting up environment variable this way doesn’t work



Even though ps get-child returned the varialble

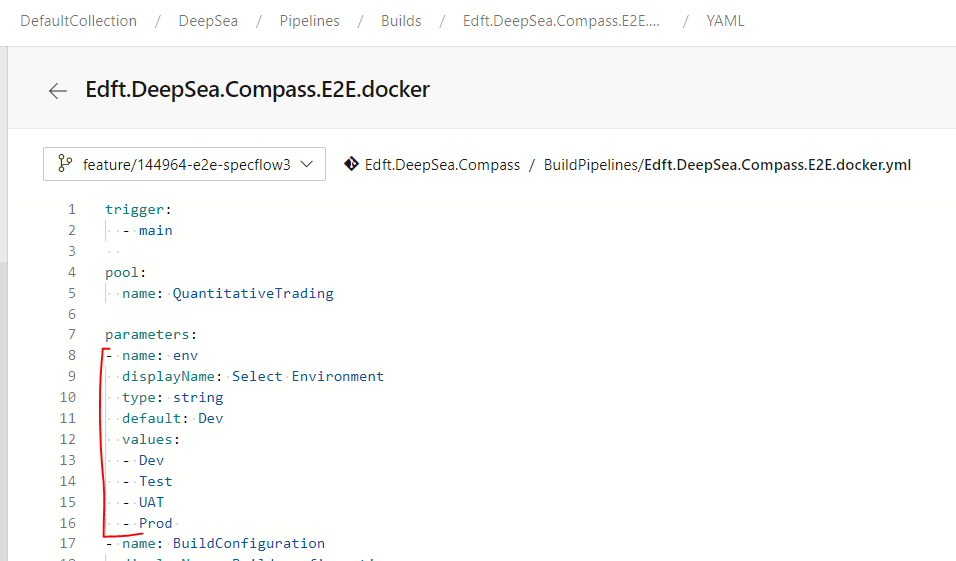
.net does not recognise it as environment variable, i.e.: the following config will not retrieve the variables set this way

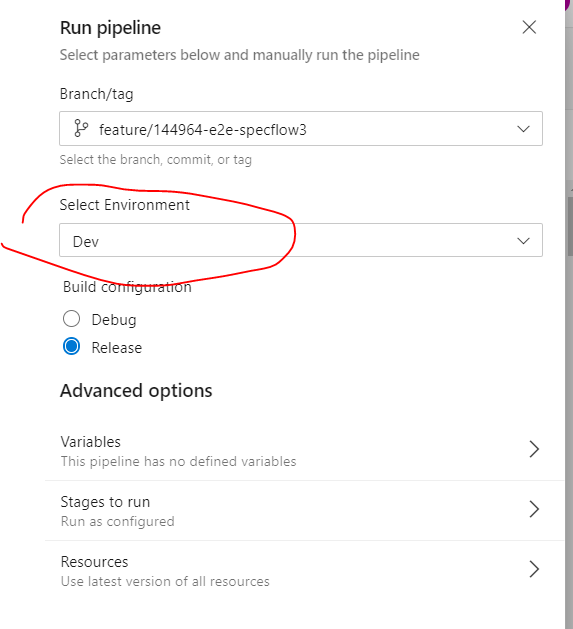
\_configuration = new ConfigurationBuilder()

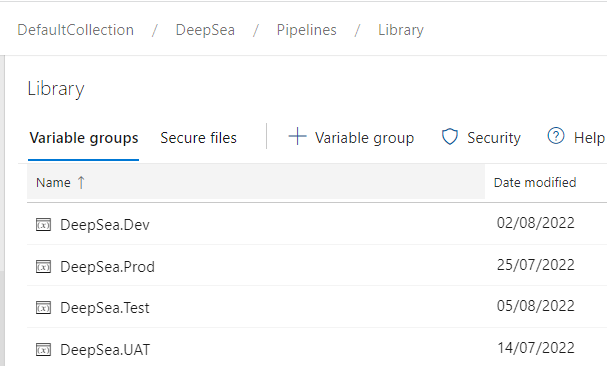
.AddEnvironmentVariables("COMPASS")

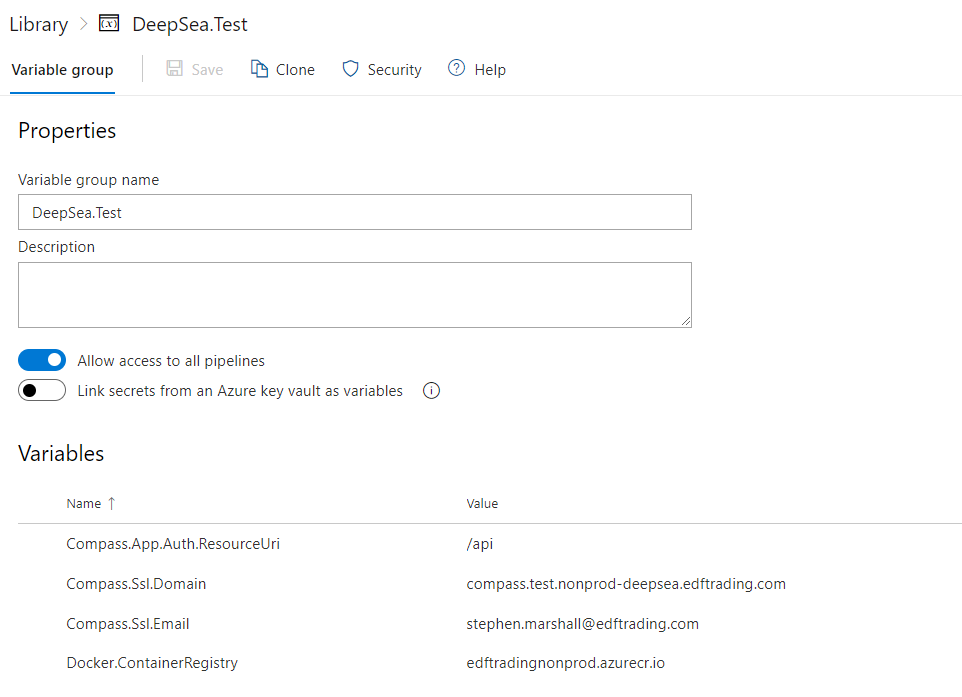
.Build();

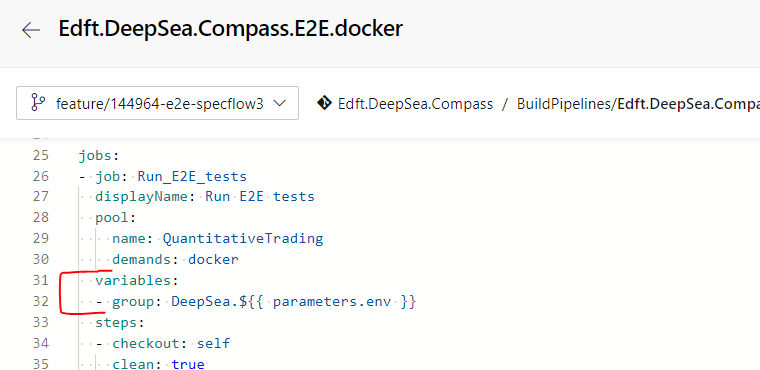
### b. in AzureDevOps pipeline

In the pipeline, use a parameter to set which environment to run 

When you run the pipeline in AzureDevOps, you’ll need to choose the value for the parameter

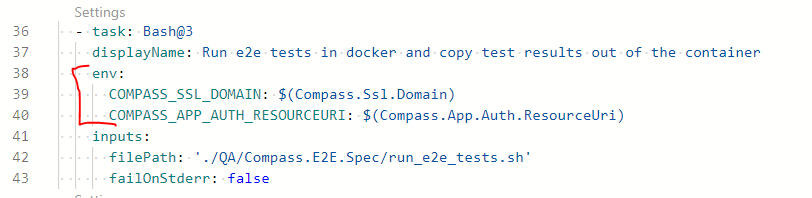
In Pipelines/Library, set up variable groups for each environment 

Each variable group should contain the corresponding values of the environment variables for the environment 

In pipeline YAML, use the env parameter value to select the right variable group: 

This selects all the variables in the group

and now we need to set the ones we need to be environment variables for the build process

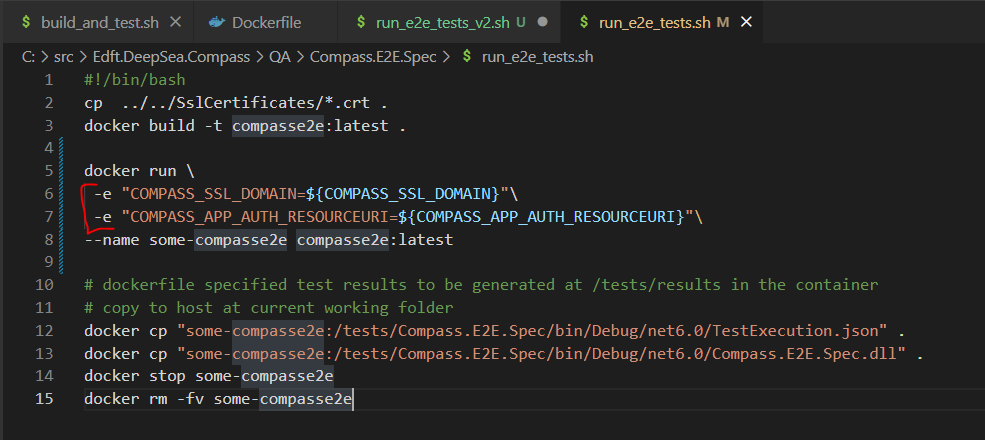
In the YAML file, in the step where you issue run docker command (or in this case, runs the script to invoke the docker run command), map docker’s env variables from the host build machine’s env

So now the build process has got the 2 env variable: COMPASS\_SSL\_DOMAIN and COMPASS\_APP\_AUTH\_RESOURCEURI

With these steps, the environment varialbes are set up for the DevOps build machine host environment.

## 2. Map the host environment variables into docker’s environment variables

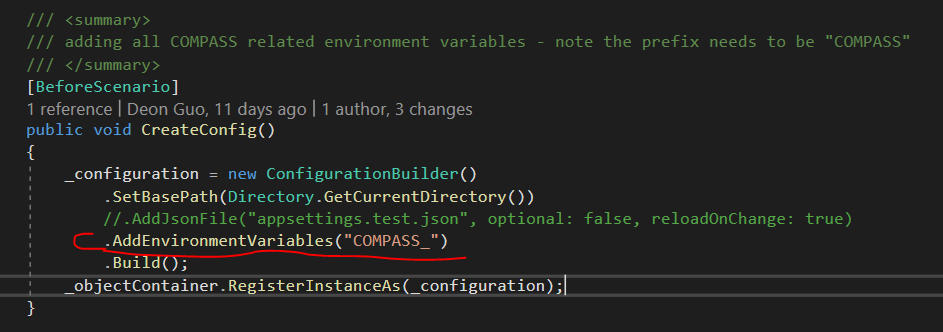
When running docker, map the environment variables from host into docker



The docker container now got these 2 env variables from the host: COMPASS\_SSL\_DOMAIN and COMPASS\_APP\_AUTH\_RESOURCEURI

## 3. Code uses the configuration to read and use the environment varialbes

In Hooks, read all the env variables with the specified prefix



Note: the prefix is then removed from the env name, e.g.: COMPASS\_SSL\_DOMAIN will be stored as SSL\_DOMAIN

Use the configuration variable 