Executing Pipelines Locally

Abstract

Having the ability to execute pipeline activities locally has been identified as an opportunity to promote positive developer experience. In this document we will explore a solution which will allow us to have the local CI experience to be as similar as possible to the remote process in the CI server.

Using the suggested method will allow us to:

- Build
- Lint
- Unit test
- E2E test
- Run Solution
- Be OS and environment agnostic.

Enter Docker Compose

Docker Compose allows you to build push or run multi-container Docker applications.

Method of Work

1. Dockerize your application(s), including a build step if possible.

2. Add a step in your docker file to execute unit tests.
3. Add a step in the docker file for linting.
4. Create a new dockerfile, possibly in a different folder, which executes end-to-end tests against the cluster. Make sure the default endpoints are configurable (This will become handy in your remote CI server, where you will be able to test against a live environment, if you choose to).
5. Create a docker-compose file which allows you to choose which of the services to run. The default will run all applications and tests, and an optional parameter can run specific services, for example only the application without the tests.

Prerequisites

1. Docker
2. Optional: if you clone the sample app, you need to have dotnet core installed.

Step by Step with Examples

For this tutorial we are going to use a sample dotnet core api application. Here is the docker file for the sample app:

```
# https://hub.docker.com/_/microsoft-dotnet
FROM mcr.microsoft.com/dotnet/sdk:5.0 AS build
WORKDIR /app

# copy csproj and restore as distinct layers
COPY ./ ./
RUN dotnet restore

RUN dotnet test
```

```
# copy everything else and build app
COPY SampleApp/. ./
RUN dotnet publish -c release -o out --no-restore

# final stage/image
FROM mcr.microsoft.com/dotnet/aspnet:5.0
WORKDIR /app
COPY --from=build /app/out .
ENTRYPOINT ["dotnet", "SampleNetApi.dll"]
```

This script restores all dependencies, builds and runs tests. The dotnet app includes `stylecop` which fails the build in case of linting issues.

Next we will also create a dockerfile to perform an end-to-end test. Usually this will look like a set of scripts, or a dedicated app which performs actual HTTP calls to a running application. For the sake of simplicity the dockerfile itself will run a simple curl command:

```
FROM alpine:3.7
RUN apk --no-cache add curl
ENTRYPOINT ["curl","0.0.0.0:8080/weatherforecast"]
```

Now we are ready to combine both of the dockerfiles in a docker-compose script:

```
version: '3'
services:
  app:
    image: app:0.01
    build:
      context: .
    ports:
      - "8080:80"
  e2e:
    image: e2e:0.01
    build:
      context: ./E2E
```

The docker-compose script will launch the 2 dockerfiles, and it will build them if they were not built before. The following command will run docker compose:

```
docker-compose up --build -d
```

Once the images are up, you can make calls to the service. The e2e image will perform the set of e2e tests. If you want to skip the tests, you can simply tell compose to run a specific service by appending the name of the service, as follows:

```
docker-compose up --build -d app
```

Now you have a local script which builds and tests you application. The next step would be make your CI run the docker-compose script.

Here is an example of a yaml file used by Azure DevOps pipelines:

```
trigger:
- master

pool:
  vmImage: 'ubuntu-latest'

variables:
  solution: '**/*.sln'
  buildPlatform: 'Any CPU'
  buildConfiguration: 'Release'

steps:
- task: DockerCompose@0
  displayName: Build, Test, E2E
  inputs:
    action: Run services
    dockerComposeFile: docker-compose.yml
- script: dotnet restore SampleApp
- script: dotnet build --configuration $(buildConfiguration)
SampleApp
  displayName: 'dotnet build $(buildConfiguration)'
```

In this script the first step is docker-compose, which uses the same file we created the previous steps. The next steps, do the same using scripts, and are here for comparison. By the end of this step, your CI effectively runs the same build and test commands you run locally.

Last update: August 22, 2024