Building Containers with Azure DevOps Using the DevTest Pattern

In this documents, we highlight learnings from applying the DevTest pattern to container development in Azure DevOps through pipelines.

The pattern enabled as to build container for development, testing and releasing the container for further reuse (production ready).

We will dive into tools needed to build, test and push a container, our environment and go through each step separately.

Follow this link to dive deeper or revisit the DevTest pattern.

Build the Container

The first step in container development, after creating the necessary Dockerfiles and source code, is building the container. Even the Dockerfile itself can include some basic testing. Code tests are performed when pushing the code to the repository origin, where it is then used to build the container.

The first step in our pipeline is to run the `docker build` command with a temporary tag and the required build arguments:

```
- task: Bash@3
  name: BuildImage
  displayName: 'Build the image via docker'
  inputs:
    workingDirectory: "$(System.DefaultWorkingDirectory)${{
parameters.buildDirectory }}"
    targetType: 'inline'
    script: |
      docker build -t ${{ parameters.imageName }} --build-arg YOUR_BUILD_ARG -f
${{ parameters.dockerfileName }} .
  env:
    PredefinedPassword: $(Password)
    NewVariable: "newVariableValue"
```

This task includes the parameters `buildDirectory`, `imageName` and `dockerfileName`, which have to be set beforehand. This task can for example be used in a template for multiple containers to improve code reuse.

It is also possible to pass environment variables directly to the Dockerfile through the `env` section of the task.

If this task succeeds, the Dockerfile was build without errors and we can continue to testing the container itself.

Test the Container

To test the container, we are using the tox environment. For more details on tox please visit the tox section of this repository or visit the official tox documentation page.

Before we test the container, we are checking for exposed credentials in the docker image history. If known passwords, used to access our internal resources, are exposed here, the build step will fail:

```
- task: Bash@3
  name: CheckIfPasswordInDockerHistory
  displayName: 'Check for password in docker history'
  inputs:
    workingDirectory: "$(System.DefaultWorkingDirectory)"
    targetType: 'inline'
    failOnStdErr: true
    script: |
      if docker image history --no-trunc ${{ parameters.imageName }} | grep -qF
$PredefinedPassword; then
        exit 1;
      fi
      exit 0;
  env:
    PredefinedPassword: $(Password)
```

After the credential test, the container is tested through the pytest extension testinfra. Testinfra is a Python-based tool which can be used to start a container, gather prerequisites, test the container and shut it down again, without any effort besides writing the tests. These tests can for example include:

- if files exist
- if environment variables are set correctly
- if certain processes are running

- if the correct host environment is used

For a complete collection of capabilities and requirements, please visit the testinfra project on GitHub.

A few methods of a Linux-based container test can look like this:

```
def test_dependencies(host):
    '''
    Check all files needed to run the container properly.
    '''
    env_file = "/app/environment.sh.env"
    assert host.file(env_file).exists

    activate_sh_path = "/app/start.sh"
    assert host.file(activate_sh_path).exists


def test_container_running(host):
    process = host.process.get(comm="start.sh")
    assert process.user == "root"


def test_host_system(host):
    system_type = 'linux'
    distribution = 'ubuntu'
    release = '18.04'

    assert system_type == host.system_info.type
    assert distribution == host.system_info.distribution
    assert release == host.system_info.release


def extract_env_var(file_content):
    import re

    regex = r"ENV_VAR=\"(?P<s>[^\"]*)\""

    match = re.match(regex, file_content)
    return match.group('s')


def test_ports_exposed(host):
    port1 = "9010"
    st1 = f"grep -q {port1} /app/Dockerfile && echo 'true' || echo 'false'"
    cmd1 = host.run(st1)
    assert cmd1.stdout


def test_listening_simserver_sockets(host):
```

```
    assert host.socket("tcp://0.0.0.0:32512").is_listening
    assert host.socket("tcp://0.0.0.0:32513").is_listening
```

To start the test, a pytest command is executed through tox.

A task containing the tox command can look like this:

```
- task: Bash@3
  name: RunTestCommands
  displayName: "Test - Run test commands"
  inputs:
    workingDirectory: "$(System.DefaultWorkingDirectory)"
    targetType: 'inline'
    script: |
      tox -e testinfra-${{ parameters.makeTarget }} -- ${{ parameters.imageName
}}
    failOnStderr: true
```

Which could trigger the following pytest code, which is contained in the tox.ini file:

```
pytest -vv tests/{env:CONTEXT} --container-image={posargs:{env:IMAGE_TAG}} --
volume={env:VOLUME}
```

As a last task of this pipeline to build and test the container, we set a variable called `testsPassed` which is only `true`, if the previous tasks succeeded:

```
- task: Bash@3
  name: UpdateTestResultVariable
  condition: succeeded()
  inputs:
    targetType: 'inline'
    script: |
      echo '##vso[task.setvariable variable=testsPassed]true'
```

Push the Container

After building and testing, if our container runs as expected, we want to release it to our Azure Container Registry (ACR) to be used by our larger application. Before that, we want to automate the push behavior and define a meaningful tag.

As a developer it is often helpful to have containers pushed to ACR, even if they are failing. This can be done by checking for the `testsPassed` variable we introduced at the end of our testing.

If the test failed, we want to add a failed suffix at the end of the tag:

```
- task: Bash@3
  name: SetFailedSuffixTag
  displayName: "Set failed suffix, if the tests failed."
  condition: and(eq(variables['testsPassed'], false),
ne(variables['Build.SourceBranchName'], 'main'))
  # if this is not a release and failed -> retag the image to add failedSuffix
  inputs:
    targetType: inline
    script: |
      docker tag ${{ parameters.containerRegistry }}/${{
parameters.imageRepository }}:${{ parameters.imageTag }} ${{
parameters.containerRegistry }}/${{ parameters.imageRepository }}:${{
parameters.imageTag }}$(failedSuffix)
```

The condition checks, if the value of `testsPassed` is `false` and also if we are not on the *main* branch, as we don't want to push failed containers from main. This helps us to keep our production environment clean.

The value for imageRepository was defined in another template, along with the `failedSuffix` and `testsPassed`:

```
parameters:
  - name: component

variables:
  testsPassed: false
  failedSuffix: "-failed"
  # the imageRepo will changed based on dev or release
  ${{ if eq( variables['Build.SourceBranchName'], 'main' ) }}:
    imageRepository: 'stable/${{ parameters.component }}'
  ${{ if ne( variables['Build.SourceBranchName'], 'main' ) }}:
    imageRepository: 'dev/${{ parameters.component }}'
```

The imageTag is open to discussion, as it depends highly on how your team wants to use the container. We went for `Build.SourceVersion` which is the commit ID of the branch the container was developed in. This allows you to easily track the origin of the container and aids debugging.

A link to Azure DevOps predefined variables can be found in the Azure Docs on Azure DevOps

After a tag was added to the container, the image must be pushed. This can be done with the following task:

```
- task: Docker@1
  name: pushFailedDockerImage
  displayName: 'Pushes failed image via Docker'
  condition: and(eq(variables['testsPassed'], false),
ne(variables['Build.SourceBranchName'], 'main'))
  # if this is not a release and failed -> push the image with the failed tag
```

```
  inputs:
    containerregistrytype: 'Azure Container Registry'
    azureSubscriptionEndpoint: ${{ parameters.serviceConnection }}
    azureContainerRegistry: ${{ parameters.containerRegistry }}
    command: 'Push an image'
    imageName: '${{ parameters.imageRepository }}:${{ parameters.imageTag
}}$(failedSuffix)'
```

Similarly, these are the steps to publish the container to the ACR, if the tests succeeded:

```
- task: Bash@3
  name: SetLatestSuffixTag
  displayName: "Set latest suffix, if the tests succeed."
  condition:  eq(variables['testsPassed'], true)
  inputs:
    targetType: inline
    script: |
      docker tag ${{ parameters.containerRegistry }}/${{
parameters.imageRepository }}:${{ parameters.imageTag }} ${{
parameters.containerRegistry }}/${{ parameters.imageRepository }}:latest
- task: Docker@1
  name: pushSuccessfulDockerImageSha
  displayName: 'Pushes successful image via Docker'
  condition: eq(variables['testsPassed'], true)
  inputs:
    containerregistrytype: 'Azure Container Registry'
    azureSubscriptionEndpoint: ${{ parameters.serviceConnection }}
    azureContainerRegistry: ${{ parameters.containerRegistry }}
    command: 'Push an image'
    imageName: '${{ parameters.imageRepository }}:${{ parameters.imageTag }}'
- task: Docker@1
  name: pushSuccessfulDockerImageLatest
  displayName: 'Pushes successful image as latest'
  condition: eq(variables['testsPassed'], true)
  inputs:
    containerregistrytype: 'Azure Container Registry'
    azureSubscriptionEndpoint: ${{ parameters.serviceConnection }}
    azureContainerRegistry: ${{ parameters.containerRegistry }}
    command: 'Push an image'
    imageName: '${{ parameters.imageRepository }}:latest'
```

If you don't want to include the `latest` tag, you can also remove the steps involving latest (SetLatestSuffixTag & pushSuccessfulDockerImageLatest).

Resources

- DevTest pattern
- Azure Docs on Azure DevOps

- official tox documentation page
- Testinfra
- Testinfra project on GitHub
- pytest

Last update: August 22, 2024