Brent Robinson

Follow 114 Followers About

Containerised CI/CD pipelines with Azure DevOps

Brent Robinson May 23, 2019 · 15 min read ★

It starts simple — one application, one technology, one build server — and CI/CD is working flawlessly. Time and technology move on, complexity grows, and your build server, while stable, is loaded with numerous versions of different tools. One day, a new tool is required, which cannot co-exist with others, and before long, build servers begin to exist for specific projects which match their precise needs. Perhaps you're not as fortunate, and new projects are held back, forced to use legacy tools because the effort to upgrade all other projects is too much.

With polyglot architecture and rapid adoption of new technology, it has become increasingly challenging to maintain build servers that meet broad technical requirements. CI/CD has brought increased importance on the infrastructure supporting delivery, in aspects such as high availability, disaster recovery, and security best practice. A standalone server that "just works" no longer meets the brief.

Container Jobs on Azure DevOps can reduce the complexity of managing the vast requirements of build servers. In this article, we'll examine container jobs, see how they address real-world problems, and compare them to alternative solutions. We'll use .NET Core in examples, but the same concepts apply to Node.js, Java, and other technologies. Experience with Azure DevOps (Pipelines and Service Connections), Docker, Ubuntu, Azure Container Registry, and the YAML format will help to make sense of the examples.

Container Jobs

agent (build server). When using the **Hosted** pools of Azure DevOps, this agent is automatically provisioned and cleaned up for us. When using **Private** pools, these are a carefully curated configuration of servers we manage ourselves.

Figure 1: A pipeline job running on an agent.

When a pipeline runs, it has the tools installed on the agent available to it. Azure DevOps provides tasks for installing specific versions of common tools for a job. This capability overcomes some challenges with managing multiple tool versions.

Container jobs take this one step further by running the pipeline inside a container provisioned on the agent. Azure DevOps first provisions the container on an agent, then orchestrates the setup of that container to execute the tasks of the pipeline. The pipeline then runs within the container and can leverage the tools available inside the container, regardless of the capabilities of the underlying agent.

Figure 2: A pipeline job running within a container on an agent (i.e. a container job).

SDK by using a different container, both on the same agent. The agent doesn't need to have any version of .NET Core installed, resulting in one less component to maintain or be concerned about compatibility.

The tasks in the pipeline are unaffected by running in a container. The tasks run as they would directly on the agent. However, if we're building a Docker image, we won't be able to do that while inside the container. We review the solution to that later on.

A container job is different from running a container during a pipeline job. The primary difference is the context of the pipeline: where it is executing, and what it has available to it. A container launched by a pipeline job (not a container job) does not have access to the capabilities of Azure DevOps, such as tasks and service connections. Any resources must be explicitly provided to the container by some other means.

Figure 3: A container running within a pipeline job on an agent.

However, launching a container in a pipeline job can be quite useful, and we'll explore how we can use this alongside container jobs shortly.

The problem

Let's take another look at the problem we're trying to solve with container jobs, and how we'll do it with an example.

We start with two projects, **Project A** and **Project B**, both of which build on a single agent pool.

Figure 4: Two projects which can run on the same agent pool.

Then, **Project C** comes along eager to use the latest preview version of .NET Core, 3.0 at the time of writing. The first solution is to install the preview SDK on the agent pool.

Figure 5: Three projects which can run on the same agent pool.

Initially, this works well, maybe with a few updates to the old projects to include a **global.json** to force an SDK version. Then challenges arise: what's the security risks of running preview software on our agents, is it reliable enough, are there compatibility issues with other software, how frequently do we need to upgrade, and how can we compare preview versions side-by-side?

The next solution is to dedicate a pool to **Project C** to meet their changing requirements without impeding on **Project A** and **Project B**.

Figure 6: Projects split across two agent pools.

problems compound as our agent pools shard further.

If we use container jobs, we can pull in containers with specific versions of the .NET Core SDK to match our projects requirements. Projects can change their SDK versions independently of the build agent capabilities. We'll have one agent pool which can service all builds, and can accommodate new requirements without any changes. We can patch the agents confidently without the risk of introducing build-time compatibility issues.

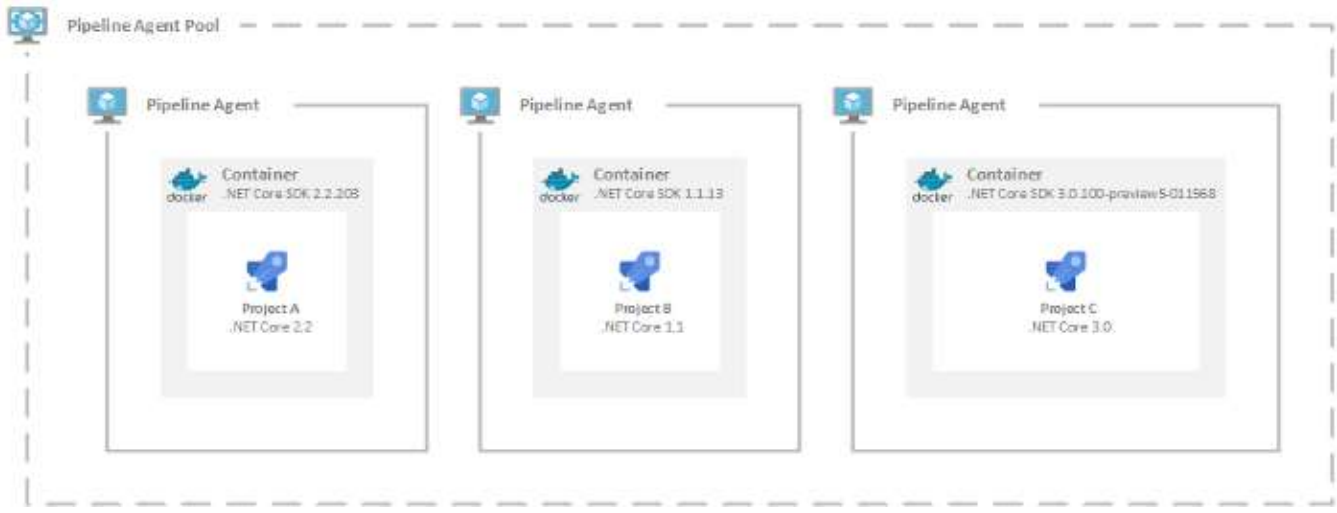

Figure 7: All projects running on one agent pool, and using containers to satisfy their dependencies.

Each project pulls in the tool versions it needs just for the lifetime of the pipeline job. This approach empowers projects to manage their dependencies, including the upgrade lifecycle.

Creating a container job

A container job can be created using the Azure Pipelines YAML syntax. We need to specify both an agent pool which can run containers, as well as the name of the container itself. For example:

```
pool:
  vmImage: 'ubuntu-16.04'

container: mcr.microsoft.com/dotnet/core/sdk:2.2
```

inside the container.

Container jobs can run on most Windows and Linux agents. At the time of writing, neither macOS or Red Hat Enterprise Linux 6 can run container jobs.

Example 1: A simple container job pipeline

Let's walk through a simple container job pipeline. We'll start with a .NET Core project which you can find on Github, along with all the pipeline definitions in this article. The project is a .NET Core 2.2 console application with unit tests and a .NET Standard 2.0 class library.

Here's our first pipeline definition which compiles the application and publishes it to Azure DevOps as an artifact.

```yaml
pool:
  vmImage: 'ubuntu-16.04'

container: mcr.microsoft.com/dotnet/core/sdk:2.2

variables:
  DOTNET_SKIP_FIRST_TIME_EXPERIENCE: true

steps:
- task: DotNetCoreCLI@2
  displayName: Build
  inputs:
    command: build
    projects: '**/*.csproj'
    arguments: '--configuration release'

- task: DotNetCoreCLI@2
  displayName: Publish
  inputs:
    command: publish
    projects: 'MyProject/MyProject.csproj'
    publishWebProjects: false
    zipAfterPublish: false
    arguments: '--configuration release'

- task: PublishPipelineArtifact@0
  displayName: Store artefact
  inputs:
```

azure-pipelines-1.yml hosted with ♡ by GitHub view raw

https://github.com/brent-robinson/azure-devops-container-jobs-example/blob/master/azure-pipelines-1.yml

At the top, we're declaring in the **pool** property that we need an Ubuntu 16.04 virtual machine to run our pipeline, followed by the container to run the pipeline in the **container** property. For each build, we create a new container instance, disposing of the container after the build. For this reason, we set the **DOTNET_SKIP_FIRST_TIME_EXPERIENCE** environment variable to skip the optimisations .NET Core performs on the first run. The optimisations speed up subsequent builds, so are of no value when we replace our environment each time. If you're using tools other than .NET Core, there may be other performance tweaks that speed up the pipeline.

The remainder of the pipeline contains typical tasks. If you haven't seen it before, the **PublishPipelineArtifact** task is a new task which replaces the **PublishBuildArtifact** task, so expect all new features to be incorporated into the new task type.

Example 2: Using resources

In our first example, we're not running any of the tests within our project. In this specific project, we require an instance of Redis to run the tests. Conventionally, we may have installed Redis on the build agent, set up a dedicated Redis server to service tests, or leveraged an underutilised instance in a development environment. In Azure Pipelines, we can run **service containers** side-by-side **container jobs**. Service containers are containers that run alongside our pipeline and can be utilised by the pipeline tasks. For example, we can run a Redis container which can service our tests.

```yaml
pool:
  vmImage: 'ubuntu-16.04'

container: mcr.microsoft.com/dotnet/core/sdk:2.2

resources:
  containers:
  - container: redis
    image: redis

services:
  redis: redis
```

```
16
17  steps:
18  - task: DotNetCoreCLI@2
19    displayName: Build
20    inputs:
21      command: build
22      projects: '**/*.csproj'
23      arguments: '--configuration release'
24
25  - task: DotNetCoreCLI@2
26    displayName: Test
27    inputs:
28      command: test
29      projects: '**/*Tests.csproj'
30      arguments: '--configuration release'
31    env:
32      CONNECTIONSTRINGS_REDIS: redis:6379
33
34  - task: DotNetCoreCLI@2
35    displayName: Publish
36    inputs:
37      command: publish
38      projects: 'MyProject/MyProject.csproj'
39      publishWebProjects: false
40      zipAfterPublish: false
41      arguments: '--configuration release'
42
43  - task: PublishPipelineArtifact@0
44    displayName: Store artefact
45    inputs:
46      artifactName: 'MyProject'
47      targetPath: 'MyProject/bin/release/netcoreapp2.2/publish/'
```

azure-pipelines-2.yml hosted with ♡ by GitHub view raw

https://github.com/brent-robinson/azure-devops-container-jobs-example/blob/master/azure-pipelines-2.yml

In our updated pipeline definition, we now have the **resources** and **services** properties. The **resources** property defines what is available to the pipeline, and the **services** property creates them. These are two distinct declarations as a pipeline may have multiple jobs. In that case, we declare the **resources** for the pipeline once but specify the **services** required for each job. This syntax adds a couple of lines for small

In the **resources** property, we list all the containers we require, in this case, just Redis. The **container** property declares the name we'll use to refer to this resource later in the pipeline. The **image** property declares the name of the image to pull from Docker Hub. If an image tag is omitted, the "latest" tag is assumed.

```
resources:
  containers:
  - container: redis
    image: redis
```

In the **services** property, we give a name to the services we require available to this job. For a container, this is the hostname we can use to communicate with the container. On the right side of the expression, we're specifying the name of the container resource. For simplicity, both are the same name.

```
services:
  redis: redis
```

In the task which executes the test, we are providing an environment variable that informs the test how to connect to the Redis instance. The test project is designed to search for this environment variable. We specify "redis" as the hostname, matching the name of the service we defined earlier. We use 6379 as the port number as this is the default port the container launches Redis on. Both the container job and the Redis service container run within the same Docker network. As they share a Docker network, they can communicate on all ports the containers use without explicitly defining these ports elsewhere in the configuration. We'll look at this in more detail in another example.

Note: if the pipeline was not running in a container job, it would not be able to connect to Redis without explicitly defining the ports to open to the container from the host.

Example 3: Multi-job matrix strategy

Perhaps for this project, we're eager to test early against new versions of the .NET Core 2.2 SDK. The Azure Pipelines syntax allows for this by defining a **strategy**.

```yaml
strategy:
  matrix:
    DotNetCore22:
      containerImage: mcr.microsoft.com/dotnet/core/sdk:2.2
    DotNetCore22Nightly:
      containerImage: mcr.microsoft.com/dotnet/core-nightly/sdk:2.2

container: $[ variables['containerImage'] ]

resources:
  containers:
  - container: redis
    image: redis

services:
  redis: redis

variables:
  DOTNET_SKIP_FIRST_TIME_EXPERIENCE: true

steps:
- task: DotNetCoreCLI@2
  displayName: Build
  inputs:
    command: build
    projects: '**/*.csproj'
    arguments: '--configuration release'

- task: DotNetCoreCLI@2
  displayName: Test
  inputs:
    command: test
    projects: '**/*Tests.csproj'
    arguments: '--configuration release'
  env:
    CONNECTIONSTRINGS_REDIS: redis:6379

- task: DotNetCoreCLI@2
  displayName: Publish
  inputs:
    command: publish
    projects: 'MyProject/MyProject.csproj'
    publishWebProjects: false
    zipAfterPublish: false
    arguments: '--configuration release'
```

```
    displayName: Store artefact
    inputs:
      artifactName: 'MyProject'
      targetPath: 'MyProject/bin/release/netcoreapp2.2/publish/'
    condition: and(succeeded(), endsWith(variables['Agent.JobName'], 'DotNetCore22'))
```

https://github.com/brent-robinson/azure-devops-container-jobs-example/blob/master/azure-pipelines-3.yml

We're using the **matrix** strategy in this case, which runs our job once for each item defined under the **matrix** property. In our case, we've created two jobs named **DotNetCore22** and **DotNetCore22Nightly**. These jobs can be called anything within the limitation of characters (alphanumeric and hyphens). Under these job declarations, we can define variables which change the behaviour of each job. These too can be given any name, and in our case, we've used **containerImage** to declare the different containers images to run our pipeline within. In the **container** property, we've applied a special syntax to use the variable we defined in the **strategy** jobs, in our case, the container image name.

```
strategy:
  matrix:
    DotNetCore22:
      containerImage: mcr.microsoft.com/dotnet/core/sdk:2.2
    DotNetCore22Nightly:
      containerImage: mcr.microsoft.com/dotnet/core-nightly/sdk:2.2

container: $[ variables['containerImage'] ]
```

When using strategies, we're effectively stating: run this job multiple times with different configurations. In our example, Azure Pipelines creates two jobs to run our pipeline, each with individual Redis instances to run the tests. If we have parallel agent licences, the jobs run in parallel. If not, they'll run sequentially. Each job is completely independent as if they are two separate pipelines altogether.

The other change to this pipeline definition is the condition on the **PublishPipelineArtifact** task. We're ensuring that we do not store the artefact from the nightly SDK job, only the stable SDK job — if the build and test succeeded.

Example 4: Multiple resources

latest Redis Docker hub image, which contains Redis version 5 at the time of writing.

```yaml
pool:
  vmImage: 'ubuntu-16.04'

strategy:
  matrix:
    DotNetCore22:
      containerImage: mcr.microsoft.com/dotnet/core/sdk:2.2
    DotNetCore22Nightly:
      containerImage: mcr.microsoft.com/dotnet/core-nightly/sdk:2.2

container: $[ variables['containerImage'] ]

resources:
  containers:
  - container: redis5
    image: redis:5
  - container: redis4
    image: redis:4

services:
  redis5: redis5
  redis4: redis4

variables:
  DOTNET_SKIP_FIRST_TIME_EXPERIENCE: true

steps:
- task: DotNetCoreCLI@2
  displayName: Build
  inputs:
    command: build
    projects: '**/*.csproj'
    arguments: '--configuration release'

- task: DotNetCoreCLI@2
  displayName: Test with Redis v5
  inputs:
    command: test
    projects: '**/*Tests.csproj'
    arguments: '--configuration release'
    testRunTitle: MyProject - Redis v5 - $(Agent.JobName)
  env:
```

```yaml
- task: DotNetCoreCLI@2
  displayName: Test with Redis v4
  inputs:
    command: test
    projects: '**/*Tests.csproj'
    arguments: '--configuration release'
    testRunTitle: MyProject - Redis v4 - $(Agent.JobName)
  env:
    CONNECTIONSTRINGS_REDIS: redis4:6379
  continueOnError: true

- task: DotNetCoreCLI@2
  displayName: Publish
  inputs:
    command: publish
    projects: 'MyProject/MyProject.csproj'
    publishWebProjects: false
    zipAfterPublish: false
    arguments: '--configuration release'

- task: PublishPipelineArtifact@0
  displayName: Store artefact
  inputs:
    artifactName: 'MyProject'
    targetPath: 'MyProject/bin/release/netcoreapp2.2/publish/'
  condition: and(eq(variables['Agent.JobStatus'], 'Succeeded'), endsWith(variables['Agent.JobNa
```

4.yml

In this example, we've defined multiple **resources** and multiple **services** using distinct names to differentiate between the two instances of Redis.

```yaml
resources:
  containers:
  - container: redis5
    image: redis:5
  - container: redis4
    image: redis:4

services:
  redis5: redis5
```

We're duplicating the test task, passing in a different connection string for each Redis version. We've given our test tasks a **testRunTitle** to differentiate between them in the test results, and also include the **Agent.JobName** to record which .NET Core SDK version it was run against (from our **strategy** declaration).

What's noteworthy here is the port number for the Redis instances. For both Redis 4 and Redis 5 we're using the same port number. Docker has created a private network on the host agent, which containers run as if they were individual hosts with their own ports. They're not using the ports of the host agent.

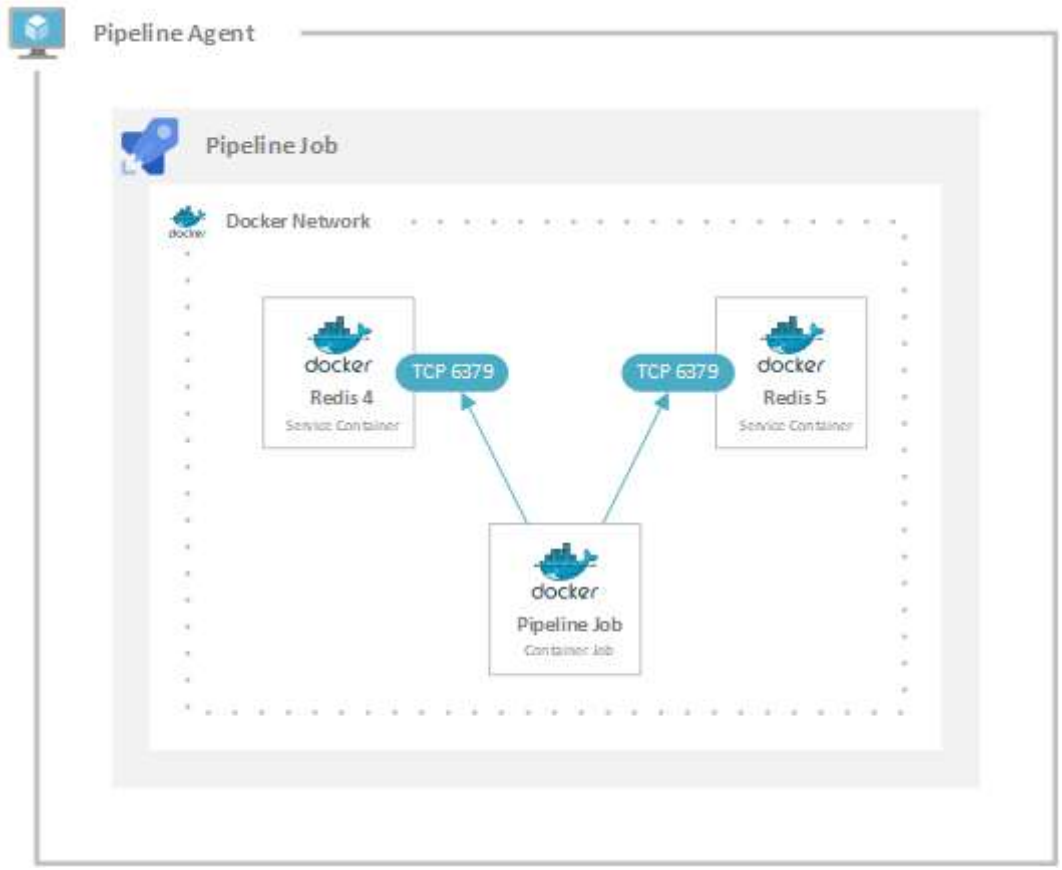

Figure 8: Docker network inside a pipeline job. Each container functions like a separate network host.

Finally, we've added the **continueOnError** property to the test tasks to ensure the pipeline doesn't immediately fail if the tests fail (mainly so we run both sets of tests, even if the first fail). To ensure we don't store the result of the build with tests that have failed, we've updated the **condition** on the **PublishPipelineArifact** task to look for a **Succeeded** job status explicitly. The **succeeded()** function we were previously using accepts both **Succeeded** and **SucceededWithIssues**. The pipeline would report as SucceededWithIssues if the only issue were a test failure.

container job pipeline? We split our pipeline in half: the first to build and test the project, the second to package it as a Docker image. We run the build and test steps inside a container job which contains the specific tools and resources we require. The packaging steps, which only require Docker, are run directly on the agent.

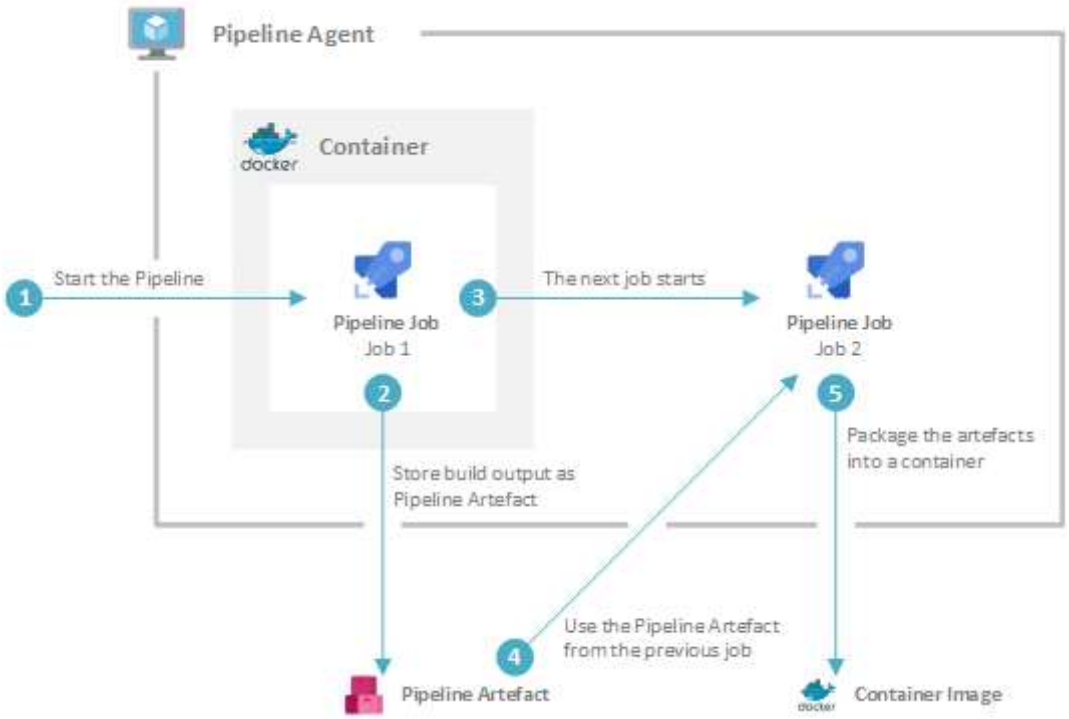

Figure 9: Producing a container as an output of a pipeline job while using container jobs.

There's no guarantee the second job runs on the same agent, and even if it does, it won't share any context with the previous job. We need to store any artefacts, and in the second job, fetch the artefacts the first job produced. The results of the second job is a container image we can publish to a registry.

```
resources:
  containers:
  - container: redis5
    image: redis:5
  - container: redis4
    image: redis:4

jobs:
- job: BuildAndTest
  displayName: Build and Test

  strategy:
    matrix:
```

```
        containerImage: mcr.microsoft.com/dotnet/core-nightly/sdk:2.2

    pool:
      vmImage: 'ubuntu-16.04'

    container: $[ variables['containerImage'] ]

    services:
      redis5: redis5
      redis4: redis4

    variables:
      DOTNET_SKIP_FIRST_TIME_EXPERIENCE: true

    steps:
    - task: DotNetCoreCLI@2
      displayName: Build
      inputs:
        command: build
        projects: '**/*.csproj'
        arguments: '--configuration release'

    - task: DotNetCoreCLI@2
      displayName: Test with Redis v5
      inputs:
        command: test
        projects: '**/*Tests.csproj'
        arguments: '--configuration release'
        testRunTitle: MyProject - Redis v5 - $(Agent.JobName)
      env:
        CONNECTIONSTRINGS_REDIS: redis5:6379
      continueOnError: true

    - task: DotNetCoreCLI@2
      displayName: Test with Redis v4
      inputs:
        command: test
        projects: '**/*Tests.csproj'
        arguments: '--configuration release'
        testRunTitle: MyProject - Redis v4 - $(Agent.JobName)
      env:
        CONNECTIONSTRINGS_REDIS: redis4:6379
      continueOnError: true
```

```
        command: publish
        projects: 'MyProject/MyProject.csproj'
        publishWebProjects: false
        zipAfterPublish: false
        arguments: '--configuration release'

    - task: PublishPipelineArtifact@0
      displayName: Store artefact
      inputs:
        artifactName: 'MyProject'
        targetPath: 'MyProject/bin/release/netcoreapp2.2/publish/'
      condition: and(eq(variables['Agent.JobStatus'], 'Succeeded'), endsWith(variables['Agent.Job

  - job: Package
    dependsOn: BuildAndTest
    condition: succeeded()

    pool:
      vmImage: 'ubuntu-16.04'

    steps:
    - checkout: none

    - task: DownloadPipelineArtifact@1
      inputs:
        artifactName: 'MyProject'
        targetPath: '$(System.DefaultWorkingDirectory)'

    - task: Docker@2
      displayName: Docker build and Push
      inputs:
        command: buildAndPush
        containerRegistry: 'MyACR'
        repository: myproject
        tags: $(Build.BuildNumber)
```

5.yml

The **jobs** property is new, and defines two **job** definitions. The **resources** are defined outside of the **jobs** property and created when we specify **services** that use them. The first job, **BuildAndTest** is almost identical to our previous example. The second job, **Package**, needs a little more explanation.

BuildAndTest succeeded, hence the **condition** of **succeeded()**. We specify the first step as **checkout: none** to inform the job to skip retrieving the source code from Git — something the pipeline does implicitly if not specified. Instead of the source code, we download the artefacts produced by the **BuildAndTest** job. Finally, we use the **Docker** task to build the image and publish it to a container registry. In this case, we're using an Azure Container Registry instance configured in Azure DevOps as a Service Connection with a name of **MyACR**. This task uses the Service Connection to authenticate with the registry and push the image.

One last thing to note here. In our .NET Core project, we've added the Dockerfile as a resource which copies to the build output directory when the project is built. The build output directory becomes the working directory on the second job. The copying of the Dockerfile is why the **Docker** command knows how to build the image: it's finding a Dockerfile in the working directory.

Using a customised image for a container job

We've now explored some complicated, but real-world uses of container jobs. However, what if an out-of-the-box container image doesn't contain everything we need to build and test our project? What if we have an uncommon (or proprietary) tool that needs to be installed, or maybe a combination of tools that aren't usually packaged together (as containers are usually specific).

There are many motivations to use a customised image, such as

· Pre-populating package caches to improve pipeline speed (e.g. NuGet, NPM)

· Installing tools or supporting software

· Configuring and optimising tools or supporting software

The benefit of making these changes in a custom image is that they only need to happen once — when creating the image — and all containers using the image immediately get the benefits. In many cases, we'll want to regularly update the image as software and optimisations change, as well as general security patching (container images need patching too!).

at runtime also enables us to rotate the credentials without replacing the image.

Building customised container images using Azure Pipelines

Let's look at an example of creating a customised container image. Perhaps we're developing an application that requires PowerShell Core available during the build and test phase — but we also need the .NET Core SDK. We can easily find these as two images, but it's more challenging to find them packaged together. We'll create a custom image to use as part of a container job.

Note: As this is only an example, our .NET Core project won't actually have a technical dependency on PowerShell Core.

Let's start with a Dockerfile that installs PowerShell on top of the .NET Core 2.2 SDK image. We're using the **2.2-bionic** tag as the version of PowerShell is specific to the underlying version of Ubuntu/Debian.

```
FROM mcr.microsoft.com/dotnet/core/runtime:2.2-bionic
RUN apt-get update \
    && apt-get install apt-transport-https -y --no-install-recommends \
    && curl -LO https://packages.microsoft.com/config/ubuntu/18.04/packages-microsoft-prod.deb \
    && dpkg -i packages-microsoft-prod.deb \
    && apt-get update \
    && apt-get install powershell -f -y \
    && pwsh --version
```

Dockerfile hosted with ♡ by GitHub view raw

https://github.com/brent-robinson/azure-devops-container-jobs-environment-example/blob/master/Dockerfile

In the Dockerfile, we're installing HTTPS support for apt (a Linux package manager), adding the Microsoft package source, then using apt to install PowerShell. Before we finish, we output the PowerShell version as a quick test.

Then, it's a matter of creating a pipeline to build the image and publish it to our private registry. Again, we're using an Azure Container Registry configured in Azure DevOps as a Service Connection named **MyACR**.

```
trigger:
  branches:
```

```yaml
pool:
  vmImage: 'ubuntu-16.04'

steps:
- task: Docker@2
  displayName: Docker build and Push
  inputs:
    dockerfile: MyProject.PipelineEnvironment/Dockerfile
    command: buildAndPush
    containerRegistry: 'MyACR'
    repository: my-project-pipeline-environment
    tags: |
      $(Build.BuildNumber)
      latest
```

azure-pipelines.yml hosted with ♡ by GitHub view raw

https://github.com/brent-robinson/azure-devops-container-jobs-environment-example/blob/master/azure-pipelines.yml

Each time we update the pipeline definition in the **master** branch, a pipeline job runs, updating the container image in our registry. Each new image receives the **latest** tag, as well as tag the image with the specific build number.

Using customised container images in an Azure Pipelines container job

There's not too much difference between using an image from Docker Hub and an image from a private container registry.

```yaml
resources:
  containers:
  - container: redis5
    image: redis:5
  - container: redis4
    image: redis:4

jobs:
- job: BuildAndTest
  displayName: Build and Test

  pool:
    vmImage: 'ubuntu-16.04'

  container:
```

```yaml
services:
  redis5: redis5
  redis4: redis4

variables:
  DOTNET_SKIP_FIRST_TIME_EXPERIENCE: true

steps:
- task: DotNetCoreCLI@2
  displayName: Build
  inputs:
    command: build
    projects: '**/*.csproj'
    arguments: '--configuration release'

- task: DotNetCoreCLI@2
  displayName: Test with Redis v5
  inputs:
    command: test
    projects: '**/*Tests.csproj'
    arguments: '--configuration release'
    testRunTitle: MyProject - Redis v5 - $(Agent.JobName)
  env:
    CONNECTIONSTRINGS_REDIS: redis5:6379
  continueOnError: true

- task: DotNetCoreCLI@2
  displayName: Test with Redis v4
  inputs:
    command: test
    projects: '**/*Tests.csproj'
    arguments: '--configuration release'
    testRunTitle: MyProject - Redis v4 - $(Agent.JobName)
  env:
    CONNECTIONSTRINGS_REDIS: redis4:6379
  continueOnError: true

- task: DotNetCoreCLI@2
  displayName: Publish
  inputs:
    command: publish
    projects: 'MyProject/MyProject.csproj'
    publishWebProjects: false
    zipAfterPublish: false
    arguments: '--configuration release'
```

```
    inputs:
      artifactName: 'MyProject'
      targetPath: 'MyProject/bin/release/netcoreapp2.2/publish/'
    condition: eq(variables['Agent.JobStatus'], 'Succeeded')

- job: Package
  dependsOn: BuildAndTest
  condition: succeeded()

  pool:
    vmImage: 'ubuntu-16.04'

  steps:
  - checkout: none

  - task: DownloadPipelineArtifact@1
    inputs:
      artifactName: 'MyProject'
      targetPath: '$(System.DefaultWorkingDirectory)'

  - task: Docker@2
    displayName: Docker build and Push
    inputs:
      command: buildAndPush
      containerRegistry: 'MyACR'
      repository: myproject
      tags: $(Build.BuildNumber)
```

6.yml

To simplify the example, we've removed the **strategy**, as one container job is sufficient. We can use a custom image as part of a **strategy**.

The **container** property now has two child properties, **image** and **endpoint**. The **image** property is the name of the image in our private repository. The image name should include the name of the registry (i.e. the part before the "/"). The **endpoint** property is the name of the Service Connection configured in Azure DevOps for the private container registry — in this case, named **MyACR**.

The **endpoint** property is all that's required to use a private, customised image for our container job. If we were publishing our customised image to Docker Hub, we wouldn't

Comparing Docker multi-stage builds and Azure Pipelines container jobs

Docker supports multi-stage builds in which commands execute within multiple containers that can consume the outputs of one another. The result is a container image, without any of the containers/images used to build it. We can run Docker multi-stage builds on Azure Pipelines agents (not as a container job).

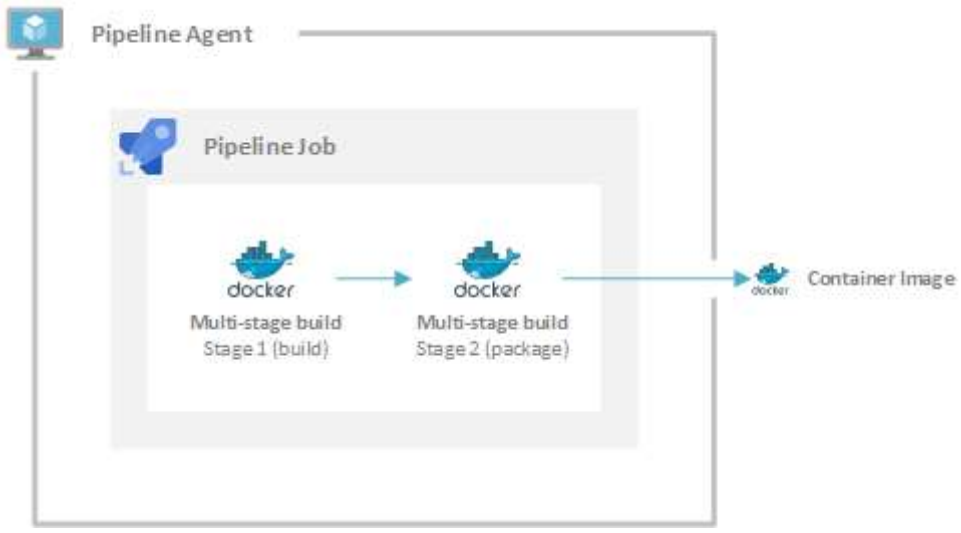

Figure 10: Multi-stage builds with Docker inside a pipeline job.

Docker's multi-stage builds seem remarkedly similar to container jobs. The distinction is that with container jobs, we can leverage the full capability of Azure DevOps and Azure Pipelines by using pipeline tasks, service connections, and many other capabilities Azure DevOps provides. Without the Azure DevOps capabilities, every action must be manually scripted, including acquiring and configuring credentials for resources such as container registries, NuGet feeds, and Azure.

Multi-stage builds are incredibly useful, but unless the process is mostly self-contained, and the sole objective is to create a container image, Azure Pipelines container jobs are a better choice.

Docker Azure Devops Net Core Azure Pipelines Containers