Introduction

Microsoft Azure DevOps (ADO) is a fully managed suite of tooling that empowers developers and operators to implement DevOps techniques. The Docker Enterprise Platform (https://www.docker.com/products/docker-enterprise) integrates with ADO through the use of Azure Pipelines for automated building and deploying of container-based workloads.

This guide does not attempt to recreate the excellent Azure DevOps documentation (https://docs.microsoft.com/en-us/azure/devops) available online, but will focus on integration points between the two systems.

Setting up Azure DevOps

An Azure DevOps tenant is necessary to use the service. These accounts are available from a myriad of different sources from Microsoft Developer Network (MSDN) subscriptions, to simply signing up (https://azure.microsoft.com/en-us/services/devops/) for a free account.

Creating a Project

Once a tenant is secured, create an Organization (https://docs.microsoft.com/en-us/azure/devops/organizations/accounts/create-organization) and a Project to hold code and pipeline definitions.

Configuring Source Control Management

Azure DevOps is capable of working with git repositories hosted in ADO itself, or in a variety of other locations such as GitHub, or on-premises servers. Create a new git repository in the ADO Project, or link to an existing repository to begin the enablement of build automation.

Automated triggers can be used between the git repository and ADO, meaning that when a git commit push occurs then an ADO Pipeline can be automatically initiate. This effectively established a continuous integration (CI) capability for source code.

Once the git repository is linked with ADO, ensure that all application code has been committed.

Building a Pipeline


Pipelines can be generated via two techniques. The Classic experience is a GUI-driven wizard where boxes and dropdowns are completed with pipeline steps. This system has been largely replaced with a YAML-based system more in line with other offerings in the DevOps market.

The YAML-based pipelines offer "pipelines as code" benefits, as they are committed to source control and able to versioned and shared like any other file. This guide will focus on YAML-based pipelines.

Triggers
Pipelines are initiated via [Triggers](https://docs.microsoft.com/en-us/azure/devops/pipelines/build/triggers), which contain the logic that determines how and when a pipeline begins execution. Triggers can be configured in a variety of ways:

- **Manual** will require that an operator initiate a new build
- **Automatic** will initiate a build any time that a commit is added to the designated source control system. To avoid builds when non-application changes are made to the repository, specify a path and/or branch to specifically target application code changes.

```
trigger:
  branches:
    include:
      - master
  paths:
    include:
      - app/src/*
```

### Pools

When a build is triggered it is added to a queue for a given agent pool ([https://docs.microsoft.com/en-us/azure/devops/pipelines/agents/pools-queues](https://docs.microsoft.com/en-us/azure/devops/pipelines/agents/pools-queues)). The next available agent will then have the build assigned, and it will execute the pipeline steps.

By default ADO uses a [hosted agent pool](https://docs.microsoft.com/en-us/azure/devops/pipelines/agents/hosted?view=azure-devops) where all servers are maintained by Microsoft. Alternatively, a pool of [custom agents](https://docs.microsoft.com/en-us/azure/devops/pipelines/agents/v2-linux?view=azure-devops) may also be used. Please see the Agents section for more detailed information on build agent setup.

Using the Ubuntu-based hosted agent (which includes a Moby-based container runtime):

```
pool:
  vmImage: 'ubuntu-latest'
```

Using a pool of custom build agents:

```
pool:
  name: 'UCP Agents - Linux'
```

### Steps

The steps within a pipeline define which actions are done to the source code. These actions may be defined as custom scripts or configured via pre-built tasks.

Script blocks are used to run shell code as a pipeline step. For small scripts it is fine to place these inline within the YAML. For larger scripts consider creating a `scripts` directory within the code repository and creating dedicated `.sh`, `.ps1`, etc. files. These files may then be called from the pipeline step without cluttering the pipeline file.

Build a Docker Image with a shell script:
Build a Docker Image with a PowerShell script:

```powershell
- powershell: |
  docker build '  
  --tag ${DOCKER_REGISTRY_IMAGE}:"$(git rev-parse --short HEAD)" '  
  ./moby/pipeline
```

displayName: 'Build Docker Image'

Tasks are pre-built actions that can easily be integrated into a pipeline. A series of tasks are available out of the box from Microsoft; however, the system is also extensible through the Visual Studio Marketplace community.

Pre-built tasks for Docker and Kubernetes are available, however the typical brevity of the `docker` and `kubectl` command lines make the additional abstraction optional compared to use of simple `script` tasks.

**Variables**

Dockerfiles are often static assets, requiring a developer to commit code changes to adjust its behavior. Hard-coded values also impede the ability to reuse a Dockerfile across multiple contexts or image variations. The Azure DevOps platform offers variables to be defined for a given Pipeline, radically increasing the flexibility of Dockerfiles by not requiring code changes to reuse a given file.

While variables may be named any value, it is recommended to decide upon a naming convention that promotes consistency and predictability. `DOCKER_` is one such convention prefix that clearly denotes that a variable is related to Docker. For example, `DOCKER_REGISTRY_USERNAME` would denote first that a value is related to Docker, that it is used to interact with a Registry, and that it contains an account username.

When Azure DevOps needs to use a variable that contains sensitive or secret information, a Build Secret may be employed rather than a Build Variable. When creating a variable, simply select the lock icon to convert the value to a secret. Secrets are not echoed out in logs or allowed to be seen once set. Setting the password or token used to authenticate with a Docker Registry via a `DOCKER_REGISTRY_TOKEN` secret would be advisable instead of a variable.

Azure DevOps, Docker, and Kubernetes all have the notion of "Secrets" to handle sensitive information and can be used in tandem to protect values in a pipeline.

**Docker Enterprise Service Account**

Steps within an Azure DevOps Pipeline that require interaction with Docker Enterprise may use a service account model for clean separation between systems. In Universal Control Plane, a new user account may be created with a name such as `azure-devops` or similiar that will serve as a service account. If using LDAP,
Preparing a Dockerfile
A developer working with a Dockerfile in their local environment has different requirements than a build automation system using the same file. A series of adjustments can optimize a Dockerfile for build performance and enhance the flexibility of a file to be utilized in multiple build variations.

**Build Arguments**

The mechanism to dynamically pass a value into a Dockerfile at `docker build` time is the `-build-arg` flag. A variable or secret can be used with the flag to change a build outcome without committing a code change into the source control system. To utilize the flag, we add an `ARG` line to our Dockerfile for each variable to be passed.

For example, to dynamically expose a port with in the Dockerfile we would adjust:

```yaml
FROM mcr.microsoft.com/dotnet/core/sdk:2.1
EXPOSE 80
WORKDIR /app
```

to include an `ARG`

```yaml
FROM mcr.microsoft.com/dotnet/core/sdk:2.1
ARG DOCKER_IMAGE_PORT=80
EXPOSE ${DOCKER_IMAGE_PORT}
WORKDIR /app
```

Note that we set a default value by including `=80`; this value will be used if a dynamic value is not passed in at build time.

A base image may also be made into a dynamic value, however the `ARG` must be placed outside of the `FROM` statement. For example, to adjust the following Dockerfile:

```yaml
FROM mcr.microsoft.com/dotnet/core/sdk:2.1
EXPOSE 80
WORKDIR /app
```

Place the `ARG` at the beginning of the file and use the variable name in the `FROM` statement:

```yaml
ARG BASE_IMAGE='mcr.microsoft.com/dotnet/core/sdk:2.1'
FROM ${BASE_IMAGE}
EXPOSE 80
WORKDIR /app
```

Positioning the `ARG` outside of the `FROM` statement(s) places it in a higher scope than within any specific stage.

Using the `ARG` and `-build-arg` pattern is useful to easily patch a given image when improvements are made to its base image. Adjusting a build variable and initiating a build brings the newer base image tag in without requiring a formal code commit.

**Labels**

Metadata may be added to a Dockerfile via the `LABEL` keyword. Labels can help designate particular application owners, points of contact, or extraneous characteristics that would benefit from embedding within an image.
Some common settings in Dockerfiles include:

```plaintext
FROM mcr.microsoft.com/dotnet/core/sdk:2.1
LABEL IMAGE_OWNER='Moby Whale <mobyw@docker.com>'
LABEL IMAGE_DEPARTMENT='Digital Marketing'
LABEL IMAGE_BUILT_BY='Azure DevOps'
EXPOSE 80
WORKDIR /app
```

Combine `LABEL` with `ARG` to dynamically pass metadata values into an image at build time.

```plaintext
FROM mcr.microsoft.com/dotnet/core/sdk:2.1
ARG COMMIT_ID=''
LABEL GIT_COMMIT_ID=${COMMIT_ID}
```

docker build `--build-arg COMMIT_ID="$(git rev-parse --short HEAD)" `
```plaintext
--tag $(DOCKER_REGISTRY_IMAGE):"$(git rev-parse --short HEAD)"
```

Values for `LABEL` can be viewed in a container registry such as Docker Trusted Registry (DTR), or from the Docker CLI:

```plaintext
$ docker inspect moby:1
[...
  "Config": {
    ... "Labels": {
      "GIT_COMMIT_ID": "421b895"
    }
  }
  ...
}
```

**Multi-Stage Builds**

Dockerfiles originally functioned as a single "stage", where all steps took place in the same context. All libraries and frameworks necessary for the Dockerfile to build had to be loaded in, bloating the size of the resulting images. Much of this image size was used during the build phase, but was not necessary for the application to properly run; for example, after an application compiles it does not necessarily have to have the compiler or SDK within the image to run.

The introduction of "multi-stage builds (https://docs.docker.com/develop/develop-images/multistage-build/)" introduced splitting out of individual "stages" within one physical Dockerfile. In a build system such as Azure DevOps, we can define a "builder" stage with a base layer containing all necessary compilation components, and then a second, lightweight "runtime" stage devoid of hefty SDKs and compilation tooling. This last stage becomes the built image, with the builder stage serving only as a temporary intermediary.
# Stage 1: Use the larger SDK image to compile .NET code

FROM mcr.microsoft.com/dotnet/core/sdk:2.1 AS build

WORKDIR /app

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

# copy everything else and build app
COPY aspnetapp/*. /aspnetapp/
WORKDIR /app/aspnetapp
RUN dotnet publish -c Release -o out

# Stage 2: Copy built artifact into the slim runtime image

FROM mcr.microsoft.com/dotnet/core/aspnet:2.1 AS runtime

EXPOSE 80
WORKDIR /app
COPY --from=build /app/aspnetapp/out ./
ENTRYPOINT ["dotnet", "aspnetapp.dll"]

Using multi-stage builds in your pipelines has considerable impact to the speed and efficiency of builds, and the resulting file size of the built container image. For a .NET Core application, the *aspnet* base layer without the *sdk* components is substantially smaller:

```
$ docker image ls \
  --format 'table {{.Repository}}|{{.Tag}}|{{.Size}}' \
  --filter=reference=mcr.microsoft.com/dotnet/core/*:*

<table>
<thead>
<tr>
<th>REPOSITORY</th>
<th>TAG</th>
<th>SIZE</th>
</tr>
</thead>
<tbody>
<tr>
<td>mcr.microsoft.com/dotnet/core/sdk</td>
<td>2.1</td>
<td>1.74GB</td>
</tr>
<tr>
<td>mcr.microsoft.com/dotnet/core/sdk</td>
<td>2.2</td>
<td>1.74GB</td>
</tr>
<tr>
<td>mcr.microsoft.com/dotnet/core/aspnet</td>
<td>2.1</td>
<td>253MB</td>
</tr>
<tr>
<td>mcr.microsoft.com/dotnet/core/runtime</td>
<td>2.1</td>
<td>180MB</td>
</tr>
</tbody>
</table>
```

For more information please see the Docker Documentation for Multi-Stage Builds (https://success.docker.com/api/asset/%2Fautomation-with-azure-devops%2FTODO).

## Agents

When a pipeline is triggered, Azure DevOps initiates a build by adding the build job to an agent queue. These queues are sets of build agents, which are the actual environments that will execute the steps defined in the pipeline.

One or more agents may be added into a pool. Using a pool allows multiple agents to be used in parallel, decreasing the time that a job sits in the queue awaiting an available agent. Microsoft maintains "hosted" agents, or you can define and run your own "self-hosted" agents within a physical server, virtual machine, or even a Docker container.
Hosted Agents

The easiest way to execute builds on Azure DevOps is by using the hosted build agents. These VM-based agent pools come in a variety of operating systems and have Docker available. Pipeline steps such as `docker build` are available without additional configuration.

Being a hosted environment, there is minimal ability to customize the SDKs, tooling, and other components within the virtual machine. A pipeline step could be used to install needed software via approaches such as `apt-get` or `chocolatey`, however doing so may add substantial time to each build considering the VMs are wiped after the pipeline completes.

The use of multi-stage builds decreases this dependency on the hosted build environment, as all necessary components for a container image should be embedded into the “builder” Dockerfile stage. To adjust the container runtime itself, for example to use a supported Docker Enterprise engine, a self-hosted agent is necessary.

Self-Hosted Agents

Running a self-hosted agent provides the ability to customize every facet of the build environment. Container runtimes can be customized, specific SDKs and compilers added, and for scenarios where project requirements restrict cloud-based agents a self-hosted agent can enable on-premises builds.

The downside is that infrastructure must be deployed and maintained to host the agents. These servers or virtual machines install a software agent, which connects to a specific Azure DevOps tenant.

The software agent can also run within a Docker container. Deploying containerized agents provides an array of benefits compared to traditional VM-based build agents including:

- Utilize existing container clusters, such as Docker Enterprise, rather than dedicated infrastructure
- Easily scale up and down the number of build agent instances with the Kubernetes or Swarm orchestrators
- Minimal assets to maintain when multi-stage builds are used as the containerized agent only needs access to the host's Docker daemon

If building traditional software that does not run in a container, then simply install the Azure DevOps agent and connect to a tenant. For containerized applications that require a `docker build` pipeline step, the Docker CLI is installed within the container so that it may connect to the host's Docker Daemon. This connection executes the build on the host but controlled from a containerized build agent.

To make this connection on Linux, a Docker Volume is used to mount the host's daemon:

```
docker run --volume /var/run/docker.sock:/var/run/docker.sock ...
```

And on Windows a Named Pipe is used via a similar approach:

```
docker run --volume \\.\pipe\docker_engine:\.\pipe\docker_engine ...
```

Note that the ability to use Named Pipes was introduced in Windows Server 1709 (SAC), and Windows Server 2019 (LTSC)

These volume mounts allow the Docker CLI to execute commands such as `docker build` within the container, and have the action executed on the host.

Linux Container Build Agent
Microsoft provides documentation (https://docs.microsoft.com/en-us/azure/devops/pipelines/agents/docker?view=azure-devops#linux) on running a build agent within a container. The base instructions may be extended as-needed, for example to add binaries such as **kubectl** and **helm**:

```bash
#=========================================================
# Stage 1: Download Docker CLI binary
#=========================================================
FROM alpine:latest AS dockercli

ARG DOCKER_BRANCH=test
ARG DOCKER_VERSION=19.03.0-beta4

RUN wget --output docker.tgz https://download.docker.com/linux/static/$\{DOCKER_BRANCH\}/x86_64/docker-$\{DOCKER_VERSION\}.tgz && \
    tar -zxvf docker.tgz & & \n    chmod +x docker/docker

#=========================================================
# Stage 2: Download kubectl binary
#=========================================================
FROM alpine:latest AS kubectl

ARG KUBECTL_VERSION=v1.14.1

RUN wget --output ./kubectl https://storage.googleapis.com/kubernetes-release/release/$\{KUBECTL_VERSION\}/bin/linux/amd64/kubectl & & \n    chmod +x ./kubectl

#=========================================================
# Stage 3: Download Helm binary
#=========================================================
FROM alpine:latest AS helm

ARG HELM_VERSION=v2.13.1

RUN wget --output helm.tar.gz https://storage.googleapis.com/kubernetes-helm/helm-$\{HELM_VERSION\}-linux-amd64.tar.gz & & \n    tar -zxvf helm.tar.gz & & \n    chmod +x linux-amd64/helm

#=========================================================
# Stage 4: Setup Azure Pipelines remote agent
# Documented at https://docs.microsoft.com/en-us/azure/devops/pipelines/agents/docker?view=azure-devops#linux
#=========================================================
FROM ubuntu:16.04

RUN apt-get update \ 
    & & apt-get install -y --no-install-recommends \ 
    ca-certificates \ 
    curl \ 
    jq \ 
    git \ 
    iputils-ping \ 
    libcurl3 \ 
    libicu55 \ 
    zip \ 
```
unzip \
&& rm -rf /var/lib/apt/lists/*

WORKDIR /azp

COPY ./start.sh .

# User is required for the Docker Socket
USER root

# Copy binaries from earlier build stages
COPY --from=dockercli docker/docker /usr/local/bin/docker
COPY --from=kubectl ./kubectl /usr/local/bin/kubectl
COPY --from=helm /linux-amd64/helm /usr/local/bin/helm

CMD ["./start.sh"]

Windows Container Build Agent

Microsoft does not maintain specific instructions for running the cross-platform build agent software within a Windows Container, but a similar approach to the Linux Container build agent may be taken:

# escape=

#========================================================
# Stage 1: Download Docker CLI binary
#========================================================
FROM mcr.microsoft.com/windows/servercore:ltsc2019 AS dockercli

SHELL ["powershell", ",-Command", "$ErrorActionPreference = 'Stop'; $ProgressPreference = 'SilentlyContinue';"]

RUN Invoke-WebRequest `  
   -OutFile docker.zip `  
   -Uri https://download.docker.com/components/engine/windows-server/18.09/docker-18.09.6.zip `  
   -UseBasicParsing; `  
   Expand-Archive `  
      -DestinationPath 'C:\' `  
      -Force `  
      -Path docker.zip;

#========================================================
# Stage 2: Download Azure DevOps Pipelines Agent
#========================================================
FROM mcr.microsoft.com/windows/servercore:ltsc2019 AS adoagent

SHELL ["powershell", ",-Command", "$ErrorActionPreference = 'Stop'; $ProgressPreference = 'SilentlyContinue';"]

ENV ADO_AGENT_URL='https://vstsagentpackage.azureedge.net/agent/2.148.2/vsts-agent-win-x64-2.148.2.zip'

RUN Invoke-WebRequest `  
   -OutFile C:\agent.zip `  
   -Uri $env:ADO_AGENT_URL `  
   -UseBasicParsing; `  
   Expand-Archive `  
      -Destination C:\agent `  
      -Force `  
      -Path agent.zip;

#========================================================
# Stage 3: Download ServiceMonitor
FROM mcr.microsoft.com/windows/servercore:ltsc2019 AS servicemonitor
SHELL ["powershell", "-Command", "$ErrorActionPreference = 'Stop'; $ProgressPreference = 'SilentlyContinue';"]
ENV SERVICE_MONITOR_VERSION='2.0.1.3'
WORKDIR C:\agent  
COPY --from=adoagent C:\agent C:\agent  
COPY --from=dockercli C:\docker C:\docker  
COPY --from=servicemonitor C:\ServiceMonitor.exe C:\ServiceMonitor.exe  
RUN $env:PATH = 'C:\docker;' + $env:PATH; `  
     Set-ItemProperty `     -Path 'HKLM:SYSTEM\CurrentControlSet\Control\Session Manager\Environment' `     -Name Path `     -Value $env:PATH;  
COPY start.ps1 .  
RUN .\start.ps1  
CMD .\start.ps1

PowerShell used as the CMD .\start.ps1 command:
# https://docs.microsoft.com/en-us/azure/devops/pipelines/agents/v2-windows

# Check if required variables are present
# ==============================================================
if (!$env:AZP_URL) {
    Write-Host "The AZP_URL environment variable is null. Please adjust before continuing";
    exit 1;
}
if (!$env:AZP_TOKEN) {
    Write-Host "The AZP_TOKEN environment variable is null. Please adjust before continuing";
    exit 1;
}
if (!$env:AZP_POOL) {
    $env:AZP_POOL = 'Default'
}

# ==============================================================
# Configure Azure Pipelines Agent
# ==============================================================
if(!(Test-Path -Path C:\agent\_work )) {  
    Write-Output "No previous agent configuration detected. Configuring agent."
    .\config.cmd 
    --acceptTeeEula 
    --auth PAT 
    --pool "$env:AZP_POOL" 
    --replace 
    --runAsService 
    --token "$env:AZP_TOKEN" 
    --unattended 
    --url "$env:AZP_URL" 
    --windowsLogonAccount "NT AUTHORITY\SYSTEM"
}

# ==============================================================
# Run Azure Pipelines Agent with ServiceMonitor
# ==============================================================
C:\ServiceMonitor.exe (Get-Service vstsagent*).Name

This Windows Container Dockerfile executes a variety of activities through the use of multiple stages:

- **Downloads the Docker CLI binary so that the container can communicate with the host’s Docker Daemon**

- **Downloads the Azure DevOps Pipeline Agent that will facilitate communications between the container and the tenant**

- **Downloads the ServiceMonitor executable, which will monitor and manage the background service used by the agent**

- **Combines all downloaded binaries into the final stage and sets path variables to be used by the startup process**

- **Sets a PowerShell file to run as the container’s startup CMD process**

Using [Docker Secrets](https://docs.docker.com/engine/swarm/secrets/) rather than an environment variable for the Azure DevOps Personal Access Token (PAT) would increase security
Deploying

Azure DevOps Pipelines may be used for both continuous integration, and for continuous delivery processes. In the integration stage a Docker image was built and pushed into Docker Trusted Registry (DTR). Delivery takes the next step to schedule the image from DTR onto a cluster of servers managed by Universal Control Plane.

Deploying with Helm

In the Kubernetes world, Helm (https://helm.sh/) is a popular tool for deploying and managing the life cycle of container workloads. A Helm "Chart" is created via the helm create command, and then values are adjusted to match a given application's needs. Docker Enterprise is a CNCF Certified distribution of Kubernetes and works seamlessly with Helm.

Azure DevOps Pipelines can interface with Docker Enterprise via Helm by having the kubectl binary installed in the build agent. This command line tool is then further configured to work with UCP through a Docker Client Bundle, which establishes a secure connection context between kubectl and UCP. Once established, standard Helm commands may be issued to update a running Helm workload.

In the following example, a formal deploy stage (https://docs.microsoft.com/en-us/azure/devops/pipelines/process/stages) is created in the Azure DevOps Pipeline that depends on the successful completion of a build stage. A Client Bundle is then downloaded from the Azure DevOps Secure File Library (https://docs.microsoft.com/en-us/azure/devops/pipelines/library/secure-files), unzipped, and sourced. The helm upgrade command then updates the declared image tag to the recently build tag from DTR with --set "image.tag=$(git rev-parse --short HEAD)". Helm then gracefully handles the upgrade process for the running image.
- stage: deploy
displayName: Deploy to Cluster
dependsOn:
  - build
cnjobs:
  - job: helm
displayName: 'Deploy Container with Helm'
pool:
  name: 'Shared SE - Linux'
demands:
  - agent.os equals Linux
  - docker
steps:
  - task: DownloadSecureFile@1
    inputs:
      secureFile: 'ucp-bundle-azure-devops.zip'

- script: |
  # Unzip Docker Client Bundle from UCP
displayName: 'Setup Docker Client Bundle'

- script: |
  # Connect Helm to cluster via Docker Client Bundle
  cd $(Agent.TempDirectory)/bundle
eval "$(<env.sh)"

  # Update deployment with Helm
  cd $(Build.SourcesDirectory)
helm upgrade \\
  ./pipelines/helm/techorama \\
  web \\
  --set "image.tag=$(git rev-parse --short HEAD)" \\
  --tiller-namespace se-stevenfollis
displayName: 'Update application with Helm'

Conclusion

Azure DevOps is an end-to-end suite of tools that enable an organization to successfully implement modern best-practices for building and releasing software. Organizations that have invested in the Docker Enterprise platform can easily utilize such tooling to build, share, and run containerized workloads wherever their Docker Enterprise clusters are operating. Simple connection points between the two systems such as Docker Client Bundles and Helm facilitate the movement of workloads and allow users to experience the benefits of both enterprise-grade systems.