Introduction

Docker containers have long been used to enable development of new applications leveraging modern application architectural patterns like microservices, but Docker containers are not just for new applications. Traditional or legacy applications can also be migrated to containers and Docker Enterprise Edition (EE) to take advantage of the benefits that Docker EE provides.

What You Will Learn

This reference architecture provides guidance and examples for modernizing traditional .NET Framework applications to Docker Enterprise Edition. You will learn to identify the types of .NET Framework applications that are good candidates for containerization, the "lift-and-shift" approach to containerization with little to no code changes, how to get started, and guidance around various .NET Framework applications and Windows Server containers, including handling Windows Integrated Authentication, networking, logging, and monitoring.

This document focuses primarily on custom .NET Framework applications. It does not cover commercial off-the-shelf (COTS) .NET Framework applications such as SharePoint and Sitecore. Although it may be possible to run these COTS applications in Docker EE, guidance on how to do so for these applications are beyond the scope of this reference architecture. Also, .NET Core is not covered. All references to .NET applications refer to .NET Framework applications and not .NET Core applications.

Refactoring to microservices architectures is also not covered in this document. At the end of the containerization process discussed in this reference architecture, your .NET Framework application will be ready should you decide to refactor parts of the application to microservices.

> Before continuing, please become familiar with the reference architecture Design Considerations and Best Practices to Modernize Traditional Apps (https://success.docker.com/article/Docker_Reference_Architecture-_Design_Considerations_and_Best_Practices_to_Modernize_Traditional_Apps_(MTA)_with_Docker_EE).

Application Selection

Before diving in, it's important to understand there are different types of .NET Framework applications. Although not intended to be exhaustive, this section describes the most common types of .NET Framework applications and considerations that need to be made for these applications before proceeding with containerization.

Application Type	Considerations
ASP.NET Framework Applications	• Web-based applications and web services • Built with ASP.NET MVC, ASP.NET Web Forms, ASP.NET Web Services, or Web API • Commonly hosted on Internet Information Services (IIS) • Good candidate for containerization
WCF Services	• Service-oriented applications built with Windows Communication Framework

	• Often hosted in IIS as well but can also be hosted inside other applications and services (for example, inside another Web application, Windows Forms app, a Windows service, etc.) • Should be a good candidate for containerization, except for services hosted in a desktop application
Windows Services	• Applications that run as background services in Windows • Can be containerized but since services run in the background, a foreground process needs to be created to keep the container running
Desktop Applications	• Desktop applications built as Windows Forms or Windows Presentation Foundation (WPF) apps • Desktop based apps with graphical user interfaces (GUIs) cannot yet be containerized
Console Applications	• Applications that run from the command line • Should be easy to containerize and is a good candidate for containerization
COTS Applications	• Short for Commercial Off the Shelf Applications • Examples include SharePoint, Sitecore, and DNN • Generally best to avoid containerizing COTS applications until the vendor officially supports running these applications in containers

Application Dependencies

When initially getting started with the app containerization process, avoid applications that have many dependencies, components, and/or many tiers. Begin with a 2-3 tier application first until you are comfortable with the containerization process before moving to more complex applications.

Additionally, for applications that have component dependencies, ensure that the components can be installed without interaction (i.e., unattended installation or scripted). Components that require interaction during installation can't be added to the Dockerfile.

Lastly, for applications that have dependencies to services or external systems (e.g. databases, file shares, web services, etc.) ensure that the addresses/endpoints for those services are stored in configuration files and are resolvable from the Docker Swarm Windows Server hosts. Any hard-coded service references will need to be refactored prior to containerization.

Application Containerization

When containerizing an application it might be tempting to refactor, re-architect, or upgrade it at the same time. However, Docker recommends a “lift and shift” approach initially, where the application is first containerized exactly “as is” with the minimal amount of changes possible. Once the application has been containerized, it can then be regression tested and operationalized with the Docker EE deployment pipeline.

With a "lift and shift" approach, some rules of thumb are:

- Keep the .NET Framework version the same
- Keep the existing application architecture

- Keep the same versions of components and application dependencies
- Keep the deployment simple: static and not elastic

Once the application is successfully containerized it should then be easier and faster to change, for example:

- Upgrade to a newer version of application server
- Gradually refactor to microservices
- Make dynamically scalable or use elastic deployment

The following sections discuss the application containerization process.

Creating the Dockerfile

The first step in a lift and shift approach is to create the Dockerfile, and the first step in creating the Dockerfile is choosing the right base Docker image to use. All containerized .NET Framework applications use an image that is based on Microsoft's Windows Server Core base OS image (https://store.docker.com/images/windowsservercore).

Microsoft Base Images

Depending on the type of .NET Framework application, consider using the following as base images to start:

Application Type	Image	
ASP.NET Applications	microsoft/aspnet (https://store.docker.com/images/aspnet)	IIS and ASP.NET Framework preinstalled
WCF Services	microsoft/iis (https://store.docker.com/community/images/microsoft/iis)	Assumes the WCF service is hosted in IIS. If hosted in another application, another base image may be more appropriate.
Windows Services	microsoft/dotnet-framework (https://store.docker.com/community/images/microsoft/dotnet-framework)	.NET Framework preinstalled
Console Applications	microsoft/dotnet-framework (https://store.docker.com/community/images/microsoft/dotnet-framework)	.NET Framework preinstalled

Image2Docker

If the application being containerized is an ASP.NET-based application, the Image2Docker (https://success.docker.com/api/asset/.%2Fmodernizing-traditional-dot-net-applications%2FI2D) (https://github.com/docker/communitytools-image2docker-win) Powershell module is a good resource to use for generating a Dockerfile that can be used as a starting point. The primary cmdlet available in the I2D module is the `ConvertTo-Dockerfile` and can be used to inspect the local machine or an image file (VHDX or WIM).

For example, to create a Dockerfile for an app hosted in IIS called `MyApp`:

```
ConvertTo-Dockerfile -Local -Artifact IIS -ArtifactParam MyApp -IncludeWindowsFeatures -OutputPath
E:\i2d-out\MyApp
```

The `-IncludeWindowsFeatures` flag is used to tell Image2Docker to inspect the list of Windows features that have been turned on in the target machine and include instructions to install those features in the outputted Dockerfile. The resulting Dockerfile is:

```
# escape=`
FROM microsoft/aspnet:3.5-windowsservercore-10.0.14393.1715
SHELL ["powershell", "-Command", "$ErrorActionPreference = 'Stop'; $ProgressPreference =
'SilentlyContinue';"]

RUN Remove-Website 'Default Web Site';

RUN Enable-WindowsOptionalFeature -Online -FeatureName IIS-ApplicationDevelopment,IIS-ASPNET,IIS-
ASPNET45,IIS-BasicAuthentication,IIS-ClientCertificateMappingAuthentication,IIS-CommonHttpFeatures,IIS-
DefaultDocument,IIS-DigestAuthentication,IIS-DirectoryBrowsing,IIS-HealthAndDiagnostics,IIS-
HttpCompressionStatic,IIS-HttpErrors,IIS-HttpLogging,IIS-IISCertificateMappingAuthentication,IIS-
IPSecurity,IIS-ISAPIExtensions,IIS-ISAPIFilter,IIS-NetFxExtensibility,IIS-NetFxExtensibility45,IIS-
Performance,IIS-RequestFiltering,IIS-Security,IIS-StaticContent,IIS-URLAuthorization,IIS-WebServer,IIS-
WebServerRole,IIS-WindowsAuthentication,NetFx4Extended-ASPNET45

# Set up website: MyApp
RUN New-Item -Path 'C:\MyApp' -Type Directory -Force;

RUN New-Website -Name 'MyApp' -PhysicalPath 'C:\MyApp' -Port 8001 -ApplicationPool '.NET v2.0' -Force;

EXPOSE 8001

COPY ["MyApp", "/MyApp"]

RUN $path='C:\MyApp'; `
    $acl = Get-Acl $path; `
    $newOwner = [System.Security.Principal.NTAccount]
(https://success.docker.com/api/asset/.%2Fmodernizing-traditional-dot-net-
applications%2F'BUILTIN\IIS_IUSRS'); `
    $acl.SetOwner($newOwner); `
    dir -r $path | Set-Acl -aclobject  $acl
```

It's important to treat the output of Image2Docker only as a starting point. Rarely will the output of the tool be the final results that are needed for the app to run properly in a Windows Server container.

The following sections discuss what to look for in the I2D file.

Windows Features

I2D picks up every Windows Server feature that's been enabled when the `-IncludeWindowsFeature` flag is on. However, in many cases, not every one of the features that's been enabled on a VM is actually needed to run the application being containerized. In the above Dockerfile, for example, `IIS-BasicAuthentication` was included even though the application doesn't use Basic Authentication at all.

To optimize your image, remove any unnecessary Windows features that aren't being used.

Application Pools

I2D does not pick up any custom application pool you may have created for your application. It assigns an out of the box application pool to your application. The app pool will either be `DefaultAppPool` or `.NET v2.0` depending on if ASP.NET v3.5 is installed on the machine.

In general, it's best to explicitly create and use your own application pool for your web app. Note that if you use a domain account or service account for your application pool identity, you cannot just specify a domain account in your Dockerfile. You need to set the identity to one of the built-in types and then use a Group Managed Service Account (gMSA) via a Credential Spec when running the container. See the section Integrated Windows Authentication (https://success.docker.com/api/asset/.%2Fmodernizing-traditional-dot-net-applications%2F#integratedwindowsauthentication) for more details.

Web Configuration Settings

I2D also does not add any web configuration details to the Dockerfile. Any settings that have been configured manually for the web application through IIS (e.g. authentication settings, etc.) must be added to your Dockerfile manually.

Final Dockerfile

The following Dockerfile is an example of a final Dockerfile that has been modified from the initial version produced by Image2Docker:

```
# escape=`
FROM microsoft/aspnet:3.5-windowsservercore-10.0.14393.1715
SHELL ["powershell", "-Command", "$ErrorActionPreference = 'Stop'; $ProgressPreference =
'SilentlyContinue';"]

RUN Remove-Website 'Default Web Site';

RUN Enable-WindowsOptionalFeature -Online -FeatureName IIS-WindowsAuthentication

# Create the App Pool
RUN Import-Module WebAdministration; `
    New-Item -Path IIS:\AppPools\MyAppPool; `
    Set-ItemProperty -Path IIS:\AppPools\MyAppPool -Name managedRuntimeVersion -Value 'v4.0'; `
    Set-ItemProperty -Path IIS:\AppPools\MyAppPool -Name processModel -value
@{identitytype='ApplicationPoolIdentity'}

# Set up website: MyApp
RUN New-Item -Path 'C:\MyApp' -Type Directory -Force;

RUN New-Website -Name 'MyApp' -PhysicalPath 'C:\MyApp' -Port 80 -ApplicationPool 'MyAppPool' -Force;

# This disables Anonymous Authentication and enables Windows Authentication
RUN $siteName='MyApp'; `
    Set-WebConfigurationProperty -filter
/system.WebServer/security/authentication/AnonymousAuthentication -name enabled -value false -location
$sitename; `
    Set-WebConfigurationProperty -filter
/system.WebServer/security/authentication/windowsAuthentication -name enabled -value true -location
$sitename;

EXPOSE 80

COPY ["MyApp", "/MyApp"]

RUN $path='C:\MyApp'; `
    $acl = Get-Acl $path; `
    $newOwner = [System.Security.Principal.NTAccount]
(https://success.docker.com/api/asset/.%2Fmodernizing-traditional-dot-net-
applications%2F'BUILTIN\IIS_IUSRS'); `
    $acl.SetOwner($newOwner); `
    dir -r $path | Set-Acl -aclobject  $acl
```

In the above Dockerfile, a new app pool was explicitly created and configuration was added to disable Anonymous Authentication and enable Windows Authentication. This image can now be built and pushed to Docker Trusted Registry:

```
docker image build -t dtr.example.com/demos/myapp:1.0-10.0.14393.1715 .
docker image push dtr.example.com/demos/myapp:1.0-10.0.14393.1715
```

During the build and debugging process, for IIS-hosted applications such as the above, you may also want to build a second Dockerfile that enables remote IIS management:

```
# escape=`
FROM dtr.example.com/demos/myapp:1.0-10.0.14393.1715
SHELL ["powershell", "-Command", "$ErrorActionPreference = 'Stop'; $ProgressPreference =
'SilentlyContinue';"]

# Enable Remote IIS Management
RUN Install-WindowsFeature Web-Mgmt-Service; `
    NET USER dockertester 'Docker1234' /ADD; `
    NET LOCALGROUP 'Administrators' 'testing' /add; `
    Configure-SMRemoting.exe -enable; `
    sc.exe config WMSVC start=auto; `
    Set-ItemProperty -Path HKLM:\SOFTWARE\Microsoft\WebManagement\Server -Name EnableRemoteManagement -
Value 1

EXPOSE 80 5985
```

With the above Dockerfile, the container's IIS is available at `<container-ip>:5985` and can be reviewed remotely on another machine with IIS management console installed. The user is `dockertester` with a password of `Docker1234`. Note that IIS management console should not be used to apply changes to running containers. It should only be used to troubleshoot and determine if instructions in the Dockerfile have been properly applied.

The above Dockerfile also represents a typical Dockerfile created for .NET Framework applications. The high level steps in such a Dockerfile are:

1. Select a base image
2. Install Windows features and other dependencies
3. Install and configure your application
4. Expose ports

CMD and ENTRYPOINT

One step that is often in a Dockerfile but not in the above example is the use of `CMD` (https://docs.docker.com/engine/reference/builder/#cmd) or `ENTRYPOINT` (https://docs.docker.com/engine/reference/builder/#entrypoint).

The ASP.NET Framework base image used in the above example already contains an entrypoint that was sufficient for this application. You can choose to create your own entrypoint for your application so you can change or add additional functionality. One scenario to use an entrypoint for is when your application needs to wait for services that it requires. Typically, a Powershell script is created to handle the wait logic:

```
# PowerShell entrypoint.ps1
while ((Get-Service "MyWindowsService").Status -ne "Running") {
  Start-Sleep -Seconds 10;
}
```

and the Dockerfile contains an `ENTRYPOINT` entry that points to that Powershell file:

```
ENTRYPOINT ["powershell", ".\\entrypoint.ps1"]
```

Image Tags and Windows Versions

When using one of the previously mentioned base images (https://success.docker.com/api/asset/.%2Fmodernizing-traditional-dot-net-applications%2F#Microsoft-Base-Images), it is important to use the right tag. For containers running on Windows Server 2016, Microsoft only

supports containers whose base image version exactly matches the host's operating system version as described in **Windows container requirements** on docs.microsoft.com (https://docs.microsoft.com/en-us/virtualization/windowscontainers/deploy-containers/system-requirements#matching-container-host-version-with-container-image-versions). Although a container may start or even appear to work even if its base version doesn't match the host's version, Microsoft cannot guarantee full functionality so it's best to always match the versions.

To determine the Windows Server version of the Docker Windows Server host, use the following Powershell command:

```
Get-ItemProperty "HKLM:\SOFTWARE\Microsoft\Windows NT\CurrentVersion" | % {"{0}.{1}.{2}.{3}" -f
$_.CurrentMajorVersionNumber,$_.CurrentMinorVersionNumber,$_.CurrentBuildNumber,$_.UBR}
```

The output will be something like `10.0.14393.1715`. When using one of Microsoft's base images, use an image tagged with the full version number outputted by the above command. For example, a Dockerfile for an ASP.NET 3.5 web application would start with the following:

```
# escape=`
FROM microsoft/aspnet:3.5-windowsservercore-10.0.14393.1715
```

When using Image2Docker to create the initial Dockerfile, make sure to change the `FROM` line to match the appropriate Docker Windows Server host version.

When tagging your own images, it's a good practice with Windows Server containers to also indicate the full Windows Server version number.

> Note that for containers started with Hyper-V isolation `--isolation=hyperv`, the version match requirement is not necessary.

Integrated Windows Authentication

One of the unique aspects often found in Windows-based applications is the use of Integrated Windows Authentication (IWA). It is often used with Windows-based applications to validate a client's identity, where the client's identity/account is maintained in Active Directory. A client, in this case, may be an end user, a computer, an application, or a service.

A common pattern is to use Integrated Windows Authentication for applications hosted in IIS to authenticate the application's end users. With this approach, the application authenticates with the credentials of the user currently logged in, eliminating the need for the application and the user to maintain another set of credentials for authentication purposes. Another common use of IWA is to use it for service-to-service authentication, such as the authentication that happens between an ASP.NET Framework application (more specifically, the application's process identity) and a backend service like a SQL Server service.

Because containers cannot currently be joined to an Active Directory domain as required for Integrated Windows Authentication to work, some additional configuration is required for applications that require IWA as these applications are migrated to containers. The following sections provide the configuration steps needed to enable IWA.

Group Managed Service Accounts

A **Group Managed Service Account (https://success.docker.com/api/asset/.%2Fmodernizing-traditional-dot-net-applications%2FgMSA)** (https://technet.microsoft.com/en-us/library/hh831782.aspx), introduced in Windows Server 2012, is similar to a Managed Service Account (MSA). Like a MSA, gMSAs are managed domain

accounts that can be used by applications and services as a specific user principal used to connect to and access network resources. Unlike MSAs, which can only be used by a single instance of a service, a gMSA can be used by multiple instances of a service running across multiple computers, such as in a server farm or in load-balanced services. Similarly, containerized applications and services use the gMSA when access to domain resources (file shares, databases, directory services, etc.) from the container are needed.

Prior to creating a Group Managed Service Account for a containerized application or service, ensure that Windows Server worker nodes that are part of your Docker Swarm cluster are joined to your Active Directory domain. This is required to access and use the gMSA. Additionally, it is highly recommended to create an Active Directory group specifically for managing the Windows Server hosts in your Docker Swarm cluster.

To create an Active Directory group called `Container Hosts`, the following Powershell command can be used:

```
New-ADGroup "Container Hosts" -GroupScope Global
```

To add your Windows Server worker nodes to this group:

```
$group = Get-ADGroup "Container Hosts";
$host = Get-ADComputer "Windows Worker Node Name";
Add-ADGroupMember $group -Members $host;
```

For the Active Directory domain controller (DC) to begin managing the passwords for Group Managed Service Accounts, a root key for the Key Distribution Service (KDS) is first needed. This step is only required once for the domain.

The Powershell cmdlet `Get-KDSRootKey` can be used to check if a root key already exists. If not, a new root key can be added with the following:

```
Add-KDSRootKey -EffectiveImmediately
```

Note that although the `-EffectiveImmediately` parameter is used, the key is not immediately replicated to all domain controllers. Additional information on creating KDS root keys that are effective immediately for test environments can be found at https://technet.microsoft.com/en-us/library/jj128430.aspx (https://technet.microsoft.com/en-us/library/jj128430.aspx).

Once the KDS root key is created and the Windows Server worker nodes are joined to the domain, a Group Managed Service Account can then be created for use by the containerized application. The Powershell cmdlet `New-ADServiceAccount` (https://technet.microsoft.com/en-us/library/hh852236.aspx) is used to create a gMSA. At a minimum, to ensure that the gMSA will work properly in a container, the `-Name`, `-ServicePrincipalName`, and `-PrincipalsAllowedToRetrieveManagedPasswords` options should be used:

```
New-ADServiceAccount -Name mySvcAcct -DNSHostName myapp.example.com -ServicePrincipalNames
'HTTP/myapp.example.com' -PrincipalsAllowedToRetrieveManagedPassword 'Container Hosts'
```

- `Name` - the account name that is given to the gMSA in Active Directory.
- `DNSHostName` - the DNS host name of the service.
- `ServicePrincipalName` - the unique identifier(s) for the service that will be using the gMSA account.
- `PrincipalsAllowedToRetrieveManagedPasswords` - the principals that are allowed to use the gMSA. In this example, `Container Hosts` is the name of the Active Directory group where all Windows Server worker nodes in the Swarm have been been added to.

Once the Group Managed Service Account has been created, you can test to see if the gMSA can be used on the Windows Server worker node by executing the following Powershell commands on that node:

```
Add-WindowsFeature RSAT-AD-Powershell;
Import-Module ActiveDirectory;
Install-ADServiceAccount mySvcAcct;
Test-ADServiceAccount mySvcAcct;
```

Credential Specs

Once a Group Managed Service Account is created, the next step is to create a **credential spec**. A credential spec is a file that resides on the Windows Server worker node and stores information about a gMSA. When a container is created, you can specify a credential spec for a container to use, which then uses the associated gMSA to access network resources.

To create a credential spec, open a Powershell session on one of the Windows Server worker nodes in the Swarm and execute the following commands:

```
Invoke-WebRequest https://raw.githubusercontent.com/Microsoft/Virtualization-
Documentation/live/windows-server-container-tools/ServiceAccounts/CredentialSpec.psm1 -OutFile
CredentialSpec.psm1
Import-Module .\CredentialSpec.psm1;
New-CredentialSpec -Name myapp -AccountName mySvcAcct;
```

The first two lines simply downloads and imports into the session a Powershell module from Microsoft's virtualization team that contains Powershell functions for creating and managing credential specs.

The `New-CredentialSpec` function is used on the last line to create a credential spec. The `-Name` parameter indicates the name for the credential spec (and is used to name the credential spec JSON file), and the `-AccountName` parameter indicates the name of the Group Managed Service Account to use.

Credential specs are created and stored in the `C:\ProgramData\docker\CredentialSpecs\` directory by default. The `Get-CredentialSpec` Powershell function can be used to list all credential specs on the current system. For each credential spec file you create, copy the file to the same directory on the other Windows Server worker nodes that are part of the cluster.

The contents of a credential spec file should look similar to the following:

```
{
    "CmsPlugins":  [
        "ActiveDirectory"
    ],
    "DomainJoinConfig":  {
        "Sid":  "S-1-5-21-2718210484-3565342085-4281728074",
        "MachineAccountName":  "mySvcAcct",
        "Guid":  "274490ad-0f72-4bdd-af6b-d8283ca3fa69",
        "DnsTreeName":  "example.com",
        "DnsName":  "example.com",
        "NetBiosName":  "DCKR"
    },
    "ActiveDirectoryConfig":  {
        "GroupManagedServiceAccounts":  [
            {
                "Name":  "mySvcAcct",
                "Scope":  "example.com"
            },
            {
                "Name":  "mySvcAcct",
                "Scope":  "DCKR"
            }
        ]
    }
}
```

Once the credential spec file is created, it can be used by a container by specifying it as the value of the `--security-opt` parameter passed to the `docker run` command:

```
$ docker run --security-opt "credentialspec=file://myapp.json" -d -p 80:80 --hostname myapp.example.com
dtr.example.com/demos/myapp:1.0-10.0.14393.1715
```

Notice in the above example, the `--hostname` value specified matches the Service Principal Name that was assigned when the Group Managed Service Account was created. This is also required for Integrated Windows Authentication to function properly.

When configuring for use in a Docker stack, the `credential_spec` and `hostname` keys can be used in the Docker Compose YAML file as in the following example:

```
version: "3.3"
services:
  web:
    image: dtr.example.com/demos/myapp:1.0-10.0.14393.1715
    credential_spec:
      file: myapp.json
    hostname: myapp.example.com
```

Networking

Networking is another aspect to consider when containerizing your Windows application's services and components. For services that need to be available outside the swarm, Linux containers are able to use Docker swarm's ingress routing mesh (https://docs.docker.com/engine/swarm/ingress/). However, Windows Server 2016 does not currently support the ingress routing mesh. Therefore Docker services scheduled for

Windows Server 2016 nodes that need to be accessed outside of swarm need to be configured to bypass Docker's routing mesh. This is done by publishing ports using `host` mode which publishes the service's port directly on the node where it is running.

Additionally, Docker's DNS Round Robin is the only load balancing strategy supported by Windows Server 2016 today; therefore, for every Docker service scheduled to these nodes, the `--endpoint-mode` parameter must also be specified with a value of `dnsrr`. For example:

```
$ docker service create \
  --publish mode=host,target=80,port=80 \
  --endpoint-mode dnsrr \
  --constraint "node.os.platform == windows" \
  dtr.example.com/demos/myapp:1.0-10.0.14393.1715
```

Because ingress routing mesh is not being used, an error could occur should a client attempt to access the service on a node where the service isn't currently deployed. One approach to ensure the service is accessible from multiple nodes is to deploy the service in `global` mode which places a single instance of the service on each node:

```
$ docker service create \
  --publish mode=host,target=80,port=80 \
  --endpoint-mode dnsrr \
  --mode global \
  --constraint "node.os.platform == windows" \
  dtr.example.com/demos/myapp:1.0-10.0.14393.1715
```

Creating a global service ensures that one and only one instance of that service runs on each node. However, if `replicated` deployment mode is what is desired, additional considerations and configurations need to be made to properly handle load balancing and service discovery. With `host` publishing mode, it is your responsibility to provide the list of IP addresses and ports to your load balancer. Doing so typically requires a custom registrator service on each Windows Server host that uses Docker events to monitor containers starting and stopping. Implementation of the custom registrator service is out of scope for this article.

Note that Docker's routing and service discovery for services on the same `overlay` network works without additional configuration.

For more details about swarm networking in general, see the Docker networking reference architecture (https://success.docker.com/article/Docker_Reference_Architecture-_Designing_Scalable,_Portable_Docker_Container_Networks).

HTTP Routing Mesh

Another option to consider for services available outside the swarm is Docker Universal Control Plane's (UCP) HTTP Routing Mesh (HRM). HRM works at the application layer (L7) and uses the `Host` HTTP request header found in HTTP requests to route incoming requests to the corresponding service. Docker services can participate in the HRM by adding a `com.docker.ucp.mesh.http` label and attaching it to an HRM network (`ucp-hrm` is a default network):

```
$ docker service create \
  --name aspnet_app \
  --port 80 \
  --network ucp-hrm \
  --label com.docker.ucp.mesh.http.demoappweb:
"external_route=http://mydemoapp.example.com,internal_port=80" \
  --placement "node.os.platform == windows" \
  dtr.example.com/demos/myapp:1.0-10.0.14393.1715
```

In the above example, because of the value for the `com.docker.ucp.mesh.http.demoappweb` label, inbound HTTP traffic received with `mydemoapp.example.com` `Host` HTTP request header will be routed to a container for this service on the container's port 80. More details on how to use HTTP Routing Mesh can be found in the Docker UCP Service Discovery and Loading Balancing reference architecture (https://success.docker.com/article/Docker_Reference_Architecture-_Universal_Control_Plane_2.0_Service_Discovery_and_Load_Balancing#thehttproutingmesh).

Logging

There are many different approaches to logging in traditional .NET Framework applications. Simpler applications log to the console (standard out or standard error), if available. Some applications will output logs to the file system or will log to Windows Event logs. Other applications will send its logs to a centralized location, such as a database or a logging service.

In Docker, logs are captured by default to a JSON file. The log entries in the file are usually whatever the console output is of the application or service. For .NET Framework applications that already write to standard output or standard error, these messages will appear in the JSON log file as well when the Docker command `docker container logs <containerid>` is issued. Some refactoring of your application may be required if your application does not currently send messages to standard out or standard error.

For .NET Framework applications that write to a log file, the entries in the log file can be relayed or redirected to the console in order to output them into Docker's logs. This approach is outlined in this post (https://blog.sixeyed.com/relay-iis-log-entries-to-read-them-in-docker/) from my colleague, Elton Stoneman, who uses a Powershell script and the `Get-Content ...` PowerShell cmdlet to relay IIS logs to Docker. This same approach can be taken with your own application's custom log files.

For applications that centralize its logs to a database, no refactoring should be necessary as long as the application in the container continues to have access to the logging database that's used. You may, however, want to do at least some refactoring to capture container-specific information in the logging DB such as container IDs, host, etc.

For applications that are sending logs to a centralized logging service, there may or may not be some refactoring required, depending on the service that is used. Additionally, Docker has several logging drivers available for Windows Server, including drivers that work with centralized logging services such as Amazon or Splunk. You can configure the logging driver that is used for each container or at the host level.

The logging drivers available for Windows Server are:

Driver	Descrption
json-file	Logs are formatted as JSON. Default logging driver for Docker.
awslogs	Writes log messages to Amazon CloudWatch logs.
etwlogs	Writes log messages as Event Tracing for Windows (ETW) events.
fluentd	Writes log messages to fluentd (forward input). The fluentd daemon must be running on the host machine.
logentries	Writes log messages to Rapid7 Logentries.
splunk	Writes log messages to splunk using the HTTP Event Collector.
syslog	Writes logging messages to the syslog facility. The syslog daemon must be running on the host machine.

More information about the logging drivers above can be found in the Docker docs (https://docs.docker.com/engine/admin/logging/overview/).

If you are not already using a centralized logging service, consider running a container-based centralized logging service running in Docker UCP. One logging service stack that is often used with Docker is ELK (Elasticsearch, Logstash and Kibana). Each component of the ELK stack can be run in a Linux container. Various Beats (https://www.elastic.co/products/beats) can then be used on the Windows Server hosts/containers to ship the appropriate logs to ELK services. A Beat, such as Winlogbeat Filebeat, can be installed on the Docker Windows Server host and configured to monitor and ship different log files. The Beat may even be containerized and run as a global service on each Windows Server host. An example of Filebeat running in Windows Server containers and shipping container logs on the host to a Docker UCP hosted ELK service can be found at https://github.com/bxtp4p/docker-logging-win (https://github.com/bxtp4p/docker-logging-win).

Monitoring

Like logging, monitoring is another aspect of .NET Framework applications where different approaches can be used, though most applications use a monitoring service such as AppDynamics, New Relic, or Microsoft Operations Management Suite (OMS). Like centralized logging services, depending on the monitoring service used, some refactoring or application configuration changes may be necessary when moving your application to a container.

If a monitoring solution isn't currently in place or you are just looking to get started and experiment with .NET Framework container monitoring, Prometheus (https://prometheus.io/) may be worth considering. Prometheus is an open source monitoring solution that can be run in a container. An example of running Prometheus in a container and monitoring an ASP.NET Framework application can be found at https://github.com/dockersamples/aspnet-monitoring (https://github.com/dockersamples/aspnet-monitoring).

Summary

This document provided an approach and guidance on how to containerize legacy .NET Framework applications. It covers how to start the containerization process, introduces tools like Image2Docker that can be used to assist in the process, and identifies key points to consider and directions on how to properly run .NET Framework applications on Docker. Follow the items outlined in this document to effectively migrate your .NET Framework applications to Docker.