# Problem Statement

Earlier sections of this document have championed a microservices architecture where business applications are distributed among different containers each running small, focused services. That goal has many benefits. In new development, it’s strongly recommended. Enterprise critical applications will also gain enough benefits to justify the cost of a re-architecture and re-implementation. But not every application will gain enough benefits to justify the cost. That doesn’t mean they can’t be used in container scenarios, or gain some important benefits.

In this section, we’ll explore an internal application for eShopOnContainers. It’s a WebForms application used to browse [[and modify?]] the catalog entries. The WebForms dependency means this application won’t run on .NET Core, without being re-written on MVC. You’ll see how you can run applications like these in containers without changes. You’ll also see how you can make minimal changes to enable them to access some of the services that have been moved to a microservices architecture.

# Benefits

The Catalog.WebForms application worked fine as a stand alone web application accessing a high availability data store. Even so, there are benefits gained by running the application in a container. You create an image for the application. From that point forward, every deployment will be running in the same environment. Every container will use the same OS version, have the same version of dependencies installed, be using the same framework, and have been built using the same process.

In addition, developers will all be running the application in this consistent environment. Issues that only appear with certain versions will appear immediately for developers rather than surfacing in a staging or production environment. Differences between the development environments among the development team matter less once applications run in containers.

Finally, containerized applications have a flatter scale-out curve. You’ve learned how containerized enable more containers in a VM, or more containers in a physical machine. This translates to higher density and fewer needed resources.

Scenarios like these benefit from a “lift and shift” operation to enable running these applications in a Docker container. The phrase “lift and shift” describes the scope of the task: You *lift* the entire application from a physical or virtual machine, and *shift* it into a container. In ideal situations, you don’t need to make any changes to the application code to run it in a container.

# Path

The original application was self-contained, including data access libraries. The database ran on a separate high availability machine. That configuration is simulated in the sample code by using a mock catalog service: you can run the catalog.WebForms application against that fake data to simulate a pure ‘lift and shift’ scenario. This demonstrates the simplest path where you move existing assets to run in a container without the smallest amount of code changes. This path is appropriate for applications that are “done”, and have minimal interaction with functionality that you are moving to microservices.

You’ll also see a path where applications that you’ve migrated to a Windows based container can gain some benefits from a small refactoring. The catalog.WebForms application accesses the catalog data. A small set of code changes enables the WebForms application to access the catalog through the microservice. These changes demonstrate the continuum you work with for your own applications. You can do anything from moving an existing application into containers, to making small changes that enable existing applications to access some of the new microservices to completely rewriting an application to fully participate in a new microservice based architecture. The right path depends on both the cost of the migration and the benefits from any migration.

# Application Tour

You can load the Catalog.WebForms solution and run the application as a stand alone app. In this configuration, the startup code configures the DI container to use the fake catalog service. Run the application and you’ll see the WebForms application displaying the catalog data.

Most of the techniques used in this application should be very familiar to anyone that has used WebForms. The use of the catalog microservice has introduced two techniques that might be unfamiliar: dependency injrection, and working with asynchronous data stores in WebForms.

Dependency Injection inverts the typical object oriented strategy of writing classes that allocate all needed resources. Dependency injection means that classes request their dependencies from a service container. The advantages of Dependency Injection is that you can replace external services with fakes or mocks to support testing or other environments.

The DI container uses web.config appSettings to control whether to use the fake catalog data, or the live data from the running service. You can see this in the Global classes Application\_Start method:

// Register Containers:

var settings= WebConfigurationManager.AppSettings;

var useFake = settings["usefake"];

bool fake = useFake == "true";

var builder = new ContainerBuilder();

if (fake)

{

builder.RegisterType<CatalogMockService>()

.As<ICatalogService>();

} else {

builder.RegisterType<CatalogMockService>()

.As<ICatalogService>();

}

var container = builder.Build();

Application.Add("container", container);

The Default.aspx code behind asks for the catalog service from the DI container when the page loads. You can see that in the LoadDatalogDataAsync() method. The code asks the container for the catalog service. Because the service runs asynchronously, you need to use a different idiom in that method to call the service. You need to register an async task that loads the data for the page.

protected override void OnLoad(EventArgs e)

{

RegisterAsyncTask(new PageAsyncTask(LoadCatalogDataAsync));

base.OnLoad(e);

}

private async Task LoadCatalogDataAsync()

{

var container = Application.Get("container") as IContainer;

using (scope = container?.BeginLifetimeScope())

{

catalog = container?.Resolve<ICatalogService>();

var collection = await catalog?.GetCatalogAsync();

catalogList.DataSource = collection;

catalogList.DataBind();

}

}

This task loads the data asynchronously from the catalog data service. Once that task is completed, the code binds the returned collection to the web forms list control.

This same technique is used on other pages that require data from the catalog service.

The default configuration for the catalog web forms application uses a mock implementation of the catlog.api service. This mock users a hard coded dataset for its data. This simplifies some tasks by removing the dependency on the catalog.api service in development environments.

<<< TBD: The catalog service only supports Read (GET) operations on the catalog, the brands, and the pictures. Should write operations be added, or do we want to keep this super simple? >>>

# Lifting and Shifting

Visual Studio provides great support for containerizing an application. You right-click on the project node, and select “Add”, and “Docker Support”. This template adds a new project to the solution called “docker-compose”. This project contains the Docker assets that compose the images you need, and start the necessary containers. In the simplest lift and shift scenarios, this will be the single service that you use for the web forms application. The template has also changes your startup project to point to the docker-compose project. That means Ctrl+F5 and F5 will now create the Docker image and launch the Docker container.

Before you press Ctrl-F5, make sure you configure Docker to use Windows containers. This will restart Docker. When you build, you’ll build the application, and the Docker image for the WebForms project. The first time you do this, it takes considerable time. You’ll pull down the base Windows Server image and the additional image for ASP.NET. Subsequent build and run cycles will be much faster.

The wizard creates several files for you. Visual studio uses these files to create the Docker image and launch a container. You can also use those same files from the CLI to run Docker commands manually.

This Dockerfile shows the basics for building a Docker image based on the Windows ASP.NET image and built to run an ASP.NET site:

FROM microsoft/aspnet

ARG source

WORKDIR /inetpub/wwwroot

COPY ${source:-obj/Docker/publish} .

The most important difference here is that the base image is “microsoft/aspnet”, which is the current Windows server image that includes the .NET framework. Other differences are that the directories copied from your source directory are different.

The other files in the docker-compose project are the docker assets needed to build and configure the containers. Visual Studio puts the various docker-compos yml files under one node to highlight how they are used. The base docker-compose file contains the directives that are common to all configurations. The docker-compose.override.yml file contains environment variables and related overrides for a developer configuration. The variants with .vs.debug and .vs.release provide environment settings that enable Visual Studio to attach to and manage the running container.

While Visual Studio integration is part of adding Docker support to your solution, you can also build and run from the command line, using ‘docker-compose up’ as you saw in previous sections.

# Using the Catalog Microservice

You can reconfigure the webforms application to use the catalog microservice instead of the fake data. Edit the web.config file and set the value of the ‘useFake’ key to false. The DI container will use the class that access the live catalog microservice instead of the class that returns the hard coded data. No other code changes are needed.

Accessing the live catalog service does mean you need to update the docker-compose project to build the catalog service image and launch the catalog service container. Docker for Windows supports both linux containers and Windows containers, but not at the same time. So, in order to run the catalog microservice, you need to build an image that runs it on top of a windows based container. That requires a different Dockerfile for the microservices project than you’ve seen in earlier sections.

<< Changes to the dockerfile described here >>

The catalog microservice relies on the SQL server database. You’ll also need to build a windows based SQL server image.

The docker-compose file now builds a new windows-based image to run the SQL database and the catalog microservice using the new Dockerfiles that build Windows based images. Then, it launches an instance of all three containers in the same Docker host. Using this configuration, you are running a Docker host with all three services: the SQL data store, the catalog microservice, and the Web Forms application.

# Development and Production Environments

There are a couple differences between this configuration and a production configuration. In the development environment, you’ll be running the WebForms application, the catalog microservice and the SQL database in Windows containers, as part of the same Docker Host. In earlier sections, you’ve seen them deployed in the same Docker host as the other .NET core based services on a Linux based Docker host. The advantage of running the multiple microservices in the same Docker host (or cluster) is that you get lower latency from the network communication.

In the development environment, you must run all the containers in the same OS. Docker for Windows does not support running Windows and Linux based containers at the same time. In production, you can decide if you want to run the catalog microservice in a windows container in the same Docker host (or cluster), or have the web forms application communicate with an instance of the catalog microservice running in a Linux container on a different Docker host. It depends on where you want the greater network latency. In most cases, you’ll want the microservices your applications depend upon running in the same Docker host (or swarm) for ease of deployment, and lower communication latency. In those configurations, the only costly communications is between the microservice instances and the high-availability servers for the persistent data storage.

# Conclusion

The Lift and Shift scenario provides you with the benefits of moving to containerized deployments for existing .NET applications. You can run applications that have taken dependencies on Windows OS features. You can run applications that rely on features in the .NET Framework that are not available in .NET Core.

This scenario can be the correct long term solution for some applications. In others, it may be a short term solution on the path to a more complete migration to microservices. The possible benefits can continue to be measured, and the cost of a further migration can be estimated to determine when or if further investment is justified.