Build and deploy a multi-container application in Azure Service Fabric

By Paolo Salvatori

Azure Customer Advisory Team (AzureCAT)

March 2018

Contents

List of figures

Authored by Paolo Salvatori. Edited by Nanette Ray. Reviewed by Matthew Snider.

Overview

This guide shows how to create a sample multi-container application using ASP.NET Core and Docker and deploy it on an Azure Service Fabric cluster. There are several choices you have to make before running multi-container applications on Azure. For example, the cluster can run on Linux or Windows and pull images from Docker Hub or Azure Container Registry. When describing your application, you can use Docker Compose or Service Fabric's application and service manifests. This guide documents several of these possible combinations so you can understand your options. Sample configuration files for these scenarios are provided on GitHub.

In containerized deployments, it's important to understand that Docker containers offer isolation, not virtualization. The operating system (OS) of the host computer and container image must be the same. At the time of this writing, you cannot use a Linux container on a Windows computer or a Windows container on a Linux computer. Consequently, you cannot use a Docker image for Linux to deploy a Window container to a Service Fabric cluster on Windows. Likewise, you cannot use a Docker image for Windows to deploy a Linux container to a Service Fabric cluster on Linux. There are exceptions—for example, you can run a Hyper-V virtual machine with a Linux OS on a physical host running Windows Server 2016. Inside the virtual machine, you can run containers built with Linux.

This guide discusses both Windows and Linux scenarios. The code base is the same for the Service Fabric clusters on Linux and Windows, but keep in mind that you need to build OS-specific images to deploy the multi-container application to a Service Fabric cluster. Windows and Linux environments also differ in the format of certificates they use as discussed later in this guide.

Service Fabric and microservices

Service Fabric is a distributed systems platform that makes it easy to package, deploy, and manage scalable and reliable microservices and containers. Microservice architectures are an important trend within the container world. The sample multi-container application for this project adopts a microservices architecture that consists of small, autonomous services. Each service is self-contained and implements a single business capability. For details, see Contents of this project later in this section.

Microservices have the following characteristics:

- In a microservices architecture, services are small, independent, and loosely coupled.
- Each microservice is a defined by a separate code base, configuration data, and data package that can be managed by a small development team.
- Each microservice can be built using different programming languages, technology stacks, libraries, and frameworks.
- Microservices communicate with each other by using well-defined application programming interfaces (APIs). Internal implementation details of each service are hidden from other services.
- Microservices can be versioned and deployed independently. A team can update an existing service without rebuilding and redeploying an entire application.

- Services are responsible for persisting their own data or external state. This approach differs from the traditional model, where a separate data layer handles data persistence.

For detailed guidance about building a microservices architecture on Azure, see Designing, building, and operating microservices on Azure.

Secrets and configuration data

This guide shows two ways to store configuration data, including the use of Azure Key Vault for storing secrets such as connection strings, credentials, and keys.

The recommended way to manage service configuration settings (including secrets) is through service configuration packages. Configuration packages are versioned and updatable through managed rolling upgrades with health validation and automatic rollback. This makes it easy and safe to roll out configuration changes for your services, including changes to secrets. To help with securing secrets, the Service Fabric SDK has built-in secret encryption and decryption functions. Secret values can be encrypted at build time, deployed via the same rolling configuration upgrade mechanism, and then decrypted and read programmatically in the service code.

Rather than using Service Fabric's encryption and decryption utilities and configuration deployment mechanisms for managing secrets, you can also manually encrypt secrets and pass the encrypted values to your application as environment variables via the application parameters file or the Docker Compose YAML files used to deploy the application to a Service Fabric cluster. Commonly this encryption uses a certificate, but there are other mechanisms. During runtime, you can then pass the certificate (or other information necessary for decryption) to the containerized services, and use code inside the services to decrypt the value of parameters or environment variables. For more information, see Managing secrets in Service Fabric applications.

A third model, and one of the models demonstrated in this guide, is to store regular configuration properties in Service Fabric configuration files or to pass them to the services via environment variables and application parameters, while storing secrets specifically within Key Vault. The services can then pull the secrets directly from Key Vault at runtime. This model is chosen because it simplifies the deployment and security requirements for the services and the application. However, with this model, you must take care if and when any secrets are updated or rotated as such changes do not easily benefit from Service Fabric's rolling upgrade model.

ⓘ **IMPORTANT:** For testing purposes, the deployment scripts in this guide let you store sensitive secret configuration data in cleartext in the application manifest, application parameters file, and Compose YAML files. This approach is not recommended in a production environment, because it potentially exposes sensitive information to unauthorized users.

In this guide

This guide is intended for software developers who have some familiarity with multi-container applications. Included is a sample application, Todolist, created with ASP.NET Core. This guide explains various options for deploying Todolist on Azure.

The table below shows the combinations implemented in this solution. For brevity, some configurations are not documented—for example, the scenario in which Service Fabric clusters on Windows pull Docker

images from Container Registry. More details about handling sensitive and non-sensitive configuration data are discussed later in this guide.

Summary of solution contents

Deployment option	Service Fabric cluster	Image store	Configuration data
Application and service manifests are used to deploy the multi-container application using Linux-specific Docker images	Linux	Docker images are pulled from Container Registry	Sensitive and non-sensitive configuration data is stored in cleartext in the Cloud.xml application parameters file
		Docker images are pulled from Docker Hub	Sensitive configuration data is stored in Key Vault, non-sensitive configuration data is stored in the Cloud.xml application parameters file
Application and service manifests are used to deploy the multi-container application using Windows-specific Docker images	Windows	Docker images are pulled from Docker Hub	Sensitive configuration data is stored in Key Vault, non-sensitive configuration data is stored in the Cloud.xml application parameters file
Docker compose is used to deploy the multi-container application using Linux-specific Docker images	Linux	Docker images are pulled from Container Registry	Sensitive and non-sensitive configuration data is defined in cleartext in the servicefabric-docker-compose-from-azure-container-registry.yml file
		Docker images are pulled from Docker Hub	Sensitive and non-sensitive configuration data is defined in cleartext in the servicefabric-docker-compose-from-docker-hub.yml file

This guide also shows how to:

- Use environment variables in a Dockerfile or Docker Compose file to specify application settings.
- Set up the deployment for monitoring using Azure Application Insights.
- Store keys and other secrets used by various components during configuration in Key Vault.
- Delegate a public domain to Azure DNS.

The sample application, scripts, and command files are included in a GitHub repository so you can experiment.

ⓘ **NOTE:** This project requires a cluster to be running on Azure. Consider trying Party Clusters, the free, time-limited (one hour) Service Fabric clusters hosted on Azure and run by the Service Fabric team.

Prerequisites for the development computer

- Install Microsoft Visual Studio 2017 with .NET Core workload. For more information, see Visual Studio Tools for Docker.
- Install Docker for Windows and configure it to use Linux containers.
- Set up the developer environment for Service Fabric.
- Create a Service Fabric cluster on Linux or Windows with a minimum of five nodes on Azure, and enable the DNS Service.
- Clone or download this multi-container solution into a directory on your local machine.
- Replace the values of the placeholder parameters as described throughout this guide.

Contents of this project

The GitHub repository for this project contains the command files, scripts, and other resources used to build and deploy the Todolist multi-container application using Service Fabric. Additional projects and scripts in the repository are intended to be used to deploy the multi-container application to Azure Container Service for Kubernetes.

The following contents of the project repository are used in the architecture based on Service Fabric:

- **TodoWeb**: This project is an ASP.NET Core 2.0 Web application that represents the front end of the solution. The user interface is composed of a set of Razor pages that can be used to browse, create, delete, update, and see the details about a collection of to-do items stored in an Azure Cosmos DB collection. The front-end service is configured to send logs, events, traces, requests, dependencies, and exceptions to Application Insights.
- **TodoApi**: This project contains an ASP.NET Core Web API service that is invoked by the **TodoWeb** front-end service to access the data stored in an Azure Cosmos DB SQL API database. Each time a CRUD operation is performed by any of the methods exposed by the **TodoController**, the back-end service sends a notification message to a Service Bus queue. You can use Service Bus Explorer to read messages from the queue. The back-end service is configured to send logs, events, traces, requests, dependencies, and exceptions to Application Insights. The back-end service adopts Swagger to expose a machine-readable representation of its RESTful API.
- **TodoAppForWindowsFromDockerHub:** This project contains a Service Fabric application that describes the multi-container application targeting Windows with images stored in Docker Hub. When deployed, Service Fabric pulls the Docker images for the Windows containers from Docker Hub. This project makes use of an application manifest and two service manifests, one for the front-end service and one for the back-end service, to deploy the application to a Service Fabric Windows cluster in Azure. Before the deployment, make sure to configure the value of the parameters used by the front-end and back-end services in the /ApplicationParameters/Cloud.xml file.
- **TodoAppForLinuxFromACR**: This project contains a Service Fabric application that describes the multi-container application targeting Linux, with images stored in Container Registry. When deployed, Service Fabric pulls the Docker images for the Linux containers from Container Registry. This project makes use of an application manifest and two service manifests, one for the front-end service and one

for the back-end service, to deploy the application to Service Fabric Linux cluster in Azure. Before the deployment, make sure to configure the value of the parameters used by the front-end and back-end services in the /ApplicationParameters/Cloud.xml file.

- **TodoAppForLinuxFromDockerHub**: This project contains a Service Fabric application that describes the multi-container application targeting Linux with images stored in Docker Hub. When deployed, Service Fabric pulls the Docker images for the Linux containers from Docker Hub. This project makes use of an application manifest and two service manifests, one for the front-end service and one for the back-end service, to deploy the application to a Service Fabric Linux cluster in Azure. Before the deployment, make sure to configure the value of the parameters used by the front-end and back-end services in the /ApplicationParameters/Cloud.xml file.

\Scripts

- **Azure-Container-Registry** contains create-azure-container-registry.cmd, which is used to create a Container Registry repository. Container Registry stores images for container deployments in Container Service, Azure Service Fabric, and other container orchestrators.

- **Push-Docker-Images-Scripts** contains push-images-to-azure-container-registry.cmd, which is used to push Docker images to a Container Registry. It also contains push-images-to-docker-hub.cmd used to push Docker images to a Docker Hub repository.

\Scripts\Service-Fabric-Docker-Compose

The following are used to deploy the multi-container application using Docker Compose. These deployments use Docker Compose to describe the application structure, rather than the native Service Fabric application and service manifests.

- **servicefabric-create-deployment-from-azure-container-registry.cmd**: Deploys the multi-container application targeting Linux and described via Docker Compose to a Service Fabric cluster. When deployed, Service Fabric digests the compose file and pulls the Docker images from Container Registry.
- **servicefabric-create-deployment-from-azure-container-registry.ps1**: Uses PowerShell to do the same thing as the previous command.
- **servicefabric-create-deployment-from-docker-hub.cmd**: Deploys the multi-container application targeting Linux and described via Docker Compose to a Service Fabric cluster. When deployed, Service Fabric digests the compose file and pulls the Docker images from Docker Hub.
- **servicefabric-create-deployment-from-docker-hub.ps1**: Uses PowerShell to do the same thing as the previous command.

\Scripts\Service-Fabric-Key-Vault

The following are used for storing keys and other secrets used by this project.

- **CreateKeyVault.cmd**: Used to create a key vault in the Azure Key Vault service.
- **AddSecretsToKeyVault.cmd**: Used to add secrets to the key vault used by the multi-container application.
- **CertificateCommands.cmd**: Contains commands to:
  - Create PEM and key files from a PFX certificate file.
  - Create a certificate (CER) containing only the public key from a PFX file using the OpenSSL tool.
- **CreateAADApplication.ps1**: Used to create an Azure Active Directory (Azure AD) application using a certificate as credentials. It then creates an Azure AD service principal for the application. The script associates the service principal with the key vault used by the application as a secrets repository.

ⓘ **NOTE:** Both the front-end (**TodoWeb**) and back-end (**TodoApi**) containerized services use the microsoft/aspnetcore:2.0 image as the base Docker image. For more information, see Official .NET Docker images.

For more information about secrets management, see the following resources:

- Manage Key Vault using CLI 2.0
- Authenticate with a certificate instead of a client secret

Architecture

Figure 1 provides a high-level look at the architecture of the multi-container Todolist application:

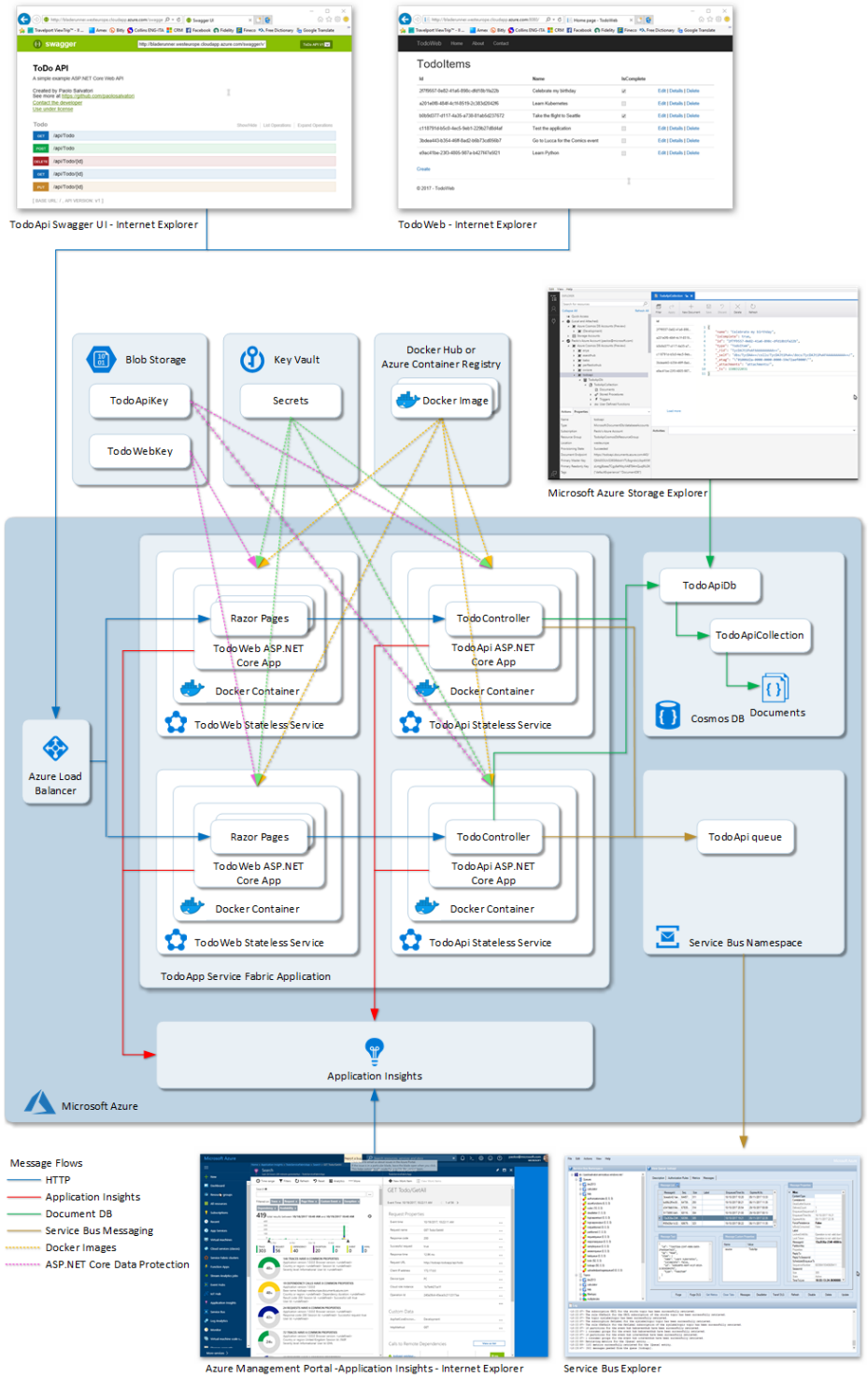

Figure 1. Architecture for deploying the multi-container application to a Service Fabric cluster.

In Service Fabric, both the **TodoApi** and **TodoWeb** containerized services are defined as stateless services. The instance count for both services is equal to **–1**. This means that a container for each service is created on each node of the Service Fabric cluster.

For the **TodoApi** service, the ApplicationManifest.xml file defines **ServiceDnsName** as *todoapi.todoapp*. The **TodoAPI**'s DNS name and assigned port are provided in the **TodoAPIService_EndpointUri** environment variable. This environment variable is passed to the **TodoWeb** front-end service, which uses it to create the HTTP address of the **TodoApi** back-end service.

Configuration

In ASP.NET Core, the configuration API provides a way of configuring an app based on a list of name-value pairs. Configuration is read at runtime from multiple sources. The name-value pairs can be grouped hierarchically. ASP.NET Core includes configuration providers for:

- File formats (INI, JSON, and XML)
- Command-line arguments
- Environment variables
- In-memory .NET objects
- An encrypted user store
- Key Vault
- Custom providers that you install or create

TodoApi service configuration

The configuration of the **TodoApi** service is defined in the appsettings.json file using the following elements:

AzureKeyVault section

This section applies to keys and other secret values that you store in Key Vault.

- **CertificateEnvironmentVariable** indicates the name of the environment variable that contains the path to the certificate file used by the service to authenticate against Key Vault. When using Windows containers, the certificate file is a .pfx file. When using Linux containers, the certificate is a .pem file.
- **KeyEnvironmentVariable** indicates the name of the environment variable that contains the path to the key of the certificate used by the service to authenticate against Key Vault. When using Windows containers, this path points to a text file that contains the password for the .pfx file. When using Linux containers, a .key certificate contains the private key of the .pem certificate.
- **ClientId** contains the application ID of the Azure AD service principal used by the service to authenticate against Key Vault based on the certificate as credentials.
- **Name** contains the name of the key vault used by the application to store credentials.

Other sections

- The **RepositoryService** element contains the **CosmosDb** element, which in turn contains the **EndpointUri**, **PrimaryKey**, **DatabaseName,** and **CollectionName** of the Azure Cosmos DB database holding the data.
- The **NotificationService** element contains the **ServiceBus** element, which in turn contains the **ConnectionString** of the Service Bus namespace used by the notification service, and the **QueueName** setting, which holds the name of the queue where the back-end service sends a message any time a CRUD operation is performed on a document.
- The **DataProtection** element contains the **BlobStorage** element, which in turn contains the **ConnectionString** of the storage account and **ContainerName** settings, which define the name of the container. The ASP.NET data protection system uses these settings to determine where to store the key. For more information, see Data Protection in ASP.NET Core.
- The **ApplicationInsights** element contains the instrumentation key of the Application Insights resource used by the service for diagnostics, logging, performance monitoring, analytics, and alerting.
- The **Logging** element contains the log level for the various logging providers.

The \ToDoApi\appsettings.json code is shown below.

```
{
    "AzureKeyVault": {
      "Certificate": {
        "CertificateEnvironmentVariable": "",
        "KeyEnvironmentVariable": ""
      },
      "ClientId": "",
      "Name": ""
    },
    "RepositoryService": {
      "CosmosDb": {
        "EndpointUri": "",
        "PrimaryKey": "",
        "DatabaseName": "",
        "CollectionName": ""
      }
    },
    "NotificationService": {
      "ServiceBus": {
        "ConnectionString": "",
        "QueueName": ""
      }
    },
    "DataProtection": {
      "BlobStorage": {
        "ConnectionString": "",
        "ContainerName": ""
      }
    },
    "ApplicationInsights": {
```

```
    "InstrumentationKey": ""
  },
  "Logging": {
    "IncludeScopes": false,
    "Debug": {
      "LogLevel": {
        "Default": "Information"
      }
    },
    "Console": {
      "LogLevel": {
        "Default": "Information"
      }
    },
    "EventSource": {
      "LogLevel": {
        "Default": "Warning"
      }
    },
    "ApplicationInsights": {
      "LogLevel": {
        "Default": "Information"
      }
    }
  }
}
```

TodoWeb service configuration

The configuration of the **TodoWeb** service is defined in the \ToDoApi\appsettings.json file using the following elements.

AzureKeyVault section

This section applies to keys and other secret values that you store in Key Vault.

- **CertificateEnvironmentVariable** indicates the name of the environment variable that contains the path to the certificate file used by the service to authenticate against Key Vault. When using Windows containers, the certificate file is a .pfx file. When using Linux containers, the certificate is a .pem file.
- **KeyEnvironmentVariable** indicates the name of the environment variable that contains the path to the key of the certificate used by the service to authenticate against Key Vault. When using Windows containers, this path will point to a text file which contains the password for the .pfx file. When using Linux containers, a .key certificate contains the private key of the .pem certificate.
- **ClientId** contains the application ID of the Azure AD service principal used by the service to authenticate against Key Vault using the certificate as credentials.
- **Name** contains the name of the key vault used by the application to store credentials.

Other sections

- The **TodoApiService** element contains the **EndpointUri** of the **TodoApi** service. In Service Fabric, this setting is used for the DNS names assigned to the **TodoApi** service. For more information, see DNS Service in Azure Service Fabric.
- The **DataProtection** element contains the **BlobStorage** element, which in turn contains the **ConnectionString** of the storage account and **ContainerName** settings, which define the name of the container. The ASP.NET data protection system uses these settings to determine where to store the key. For more information, see Data Protection in ASP.NET Core.
- The **ApplicationInsights** element contains the instrumentation key of the Application Insights resource used by the service for diagnostics, logging, performance monitoring, analytics, and alerting.
- The **Logging** element contains the log level for the various logging providers.

The appsettings.json code for **TodoWeb** is shown below.

```
{
  "AzureKeyVault": {
    "Certificate": {
      "CertificateEnvironmentVariable": "",
      "KeyEnvironmentVariable": ""
    },
    "ClientId": "",
    "Name": ""
  },
  "TodoApiService": {
    "EndpointUri": ""
  },
  "DataProtection": {
    "BlobStorage": {
      "ConnectionString": "",
      "ContainerName": ""
    }
  },
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Warning"
    },
    "ApplicationInsights": {
      "LogLevel": {
        "Default": "Information"
      }
    }
  },
  "ApplicationInsights": {
    "InstrumentationKey": ""
  }
}
```

How configuration works in ASP.NET Core

The CreateDefaultBuilder extension method in an ASP.NET Core 2.*x* app adds configuration providers for reading JSON files and system configuration sources:

- appsettings.json
- appsettings.<EnvironmentName>.json
- environment variables

In ASP.NET Core, you can use additional configuration providers to read settings from a heterogeneous range of repositories. Later in this document, we'll see how to:

- Create a key vault using the Azure CLI.
- Create secrets in this repository using the Azure CLI.
- Use the Key Vault configuration provider in the application to read sensitive parameters from the vault.

Configuration consists of a hierarchical list of name-value pairs in which the nodes are separated by a colon. To retrieve a value, access the configuration indexer with the corresponding item's key. For example, if you want to retrieve the value of the **QueueName** setting from the configuration of the **TodoApi** service, you must use the following format:

```
var queueName = Configuration["NotificationService:ServiceBus:QueueName"];
```

If you want to create an environment variable to provide a value for a setting defined in the appsettings.json file, you can replace the colon (:) with a double underscore (__).

```
NotificationService__ServiceBus__QueueName=todoapi
```

The **CreateDefaultBuilder** helper method specifies environment variables last, so that the local environment can override anything set in the deployed configuration files. This allows you to define settings in the appsettings.json file but leave their values empty and specify their values using environment variables.

For more information about configuration, see the following resources:

- Configuration in ASP.NET Core
- ASP.NET Core and Docker environment variables

Define the Docker images and containers

For this solution, Visual Studio Tools for Docker was used to build an image based on the microsoft/aspnetcore:2.0 standard image. The tool creates the Dockerfiles that automate the steps to create an image for both the front-end and back-end services. For example, the Dockerfile instructions set up the environment inside your container, load the application you want to run, and map the ports. You can customize the Dockerfile as needed. Then you use it as the input to the *docker build* command, which creates the image.

For example, here is the Dockerfile of the **TodoApi** service:

```
FROM microsoft/aspnetcore:2.0
ARG source
WORKDIR /app
EXPOSE 80
COPY ${source:-obj/Docker/publish} .
ENTRYPOINT ["dotnet", "TodoApi.dll"]
```

The Dockerfile of the **TodoWeb** service shows the different entry point:

```
FROM microsoft/aspnetcore:2.0
ARG source
WORKDIR /app
EXPOSE 80
COPY ${source:-obj/Docker/publish} .
ENTRYPOINT ["dotnet", "TodoWeb.dll"]
```

Visual Studio Tools for Docker also creates the docker-compose.yml and docker-compose-override.yml files that you can use to test the application locally. For an introduction to building and deploying a multi-container application with Visual Studio, see Defining your multi-container application with docker-compose.yml.

Here is docker-compose.yml:

```
version: '3'

services:
  todoapi:
    image: todoapi
    build:
      context: ./TodoApi
      dockerfile: Dockerfile

  todoweb:
    image: todoweb
    build:
      context: ./TodoWeb
      dockerfile: Dockerfile
  dockerfile: Dockerfile
```

Here is docker-compose-override.yml:

```
version: '3'

services:
  todoapi:
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - RepositoryService__CosmosDb__EndpointUri=COSMOS_DB_ENDPOINT_URI
      - RepositoryService__CosmosDb__PrimaryKey=DOCUMENT_DB_PRIMARY_KEY
      - RepositoryService__CosmosDb__DatabaseName=TodoApiDb
      - RepositoryService__CosmosDb__CollectionName=TodoApiCollection
      - NotificationService__ServiceBus__ConnectionString=SERVICE_BUS_CONNECTIONSTRING
      - NotificationService__ServiceBus__QueueName=todoapi
      - DataProtection__BlobStorage__ConnectionString=STORAGE_ACCOUNT_CONNECTION_STRING
      - DataProtection__BlobStorage__ContainerName=todoapi
      - ApplicationInsights__InstrumentationKey=APPLICATION_INSIGHTS_INSTRUMENTATION_KEY

    ports:
      - "80"

  todoweb:
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - TodoApiService__EndpointUri=todoapi
      - DataProtection__BlobStorage__ConnectionString=STORAGE_ACCOUNT_CONNECTION_STRING
      - DataProtection__BlobStorage__ContainerName=todoweb
      - ApplicationInsights__InstrumentationKey=APPLICATION_INSIGHTS_INSTRUMENTATION_KEY
    ports:
      - "80"
```

Before debugging the application in Visual Studio, make the following changes to the docker-compose-override.yml file:

- Replace COSMOS_DB_ENDPOINT_URI with the endpoint URI of your Azure Cosmos DB.
- Replace COSMOS_DB_PRIMARY_KEY with the primary key of your Azure Cosmos DB.
- Replace SERVICE_BUS_CONNECTION_STRING with the connection string of your Service Bus Messaging namespace.
- Replace STORAGE_ACCOUNT_CONNECTION_STRING with the connection string of the Storage Account used by ASP.NET Core Data Protection.
- Replace APPLICATION_INSIGHTS_INSTRUMENTATION_KEY with the instrumentation key of the Application Insights resource used to monitor the multi-container application.

ⓘ **NOTE:** The Docker Compose file used to deploy and test the multi-container application locally in the Visual Studio solution containing the sample requires that Docker on the local computer is configured to use Linux containers.

You can debug the application locally by specifying sensitive and non-sensitive configuration data in the docker-compose-override.yml in plain text. Alternatively, you can store and read sensitive configuration data from Key Vault. When using Key Vault to store secrets, and Linux containers to run the front-end and back-end services, a few configuration steps are needed. You must also make changes to the docker-compose-override.yml file before debugging the application locally in Visual Studio.

Do the following operations before debugging the application on the local machine. For more details about steps 1 to 5, see Store secret parameters in Azure Key Vault later in this document.

1. In \Scripts\Service-Fabric-Key-Vault, run CreateKeyVault.cmd to create key vault used by the application. For more information, see Manage Key Vault using CLI 2.0.
2. Open \Scripts\Service-Fabric-Key-Vault\AddSecretsToKeyVault.cmd and replace the placeholders for the secrets parameters such as SERVICE_BUS_CONNECTION_STRING with your values.
3. Run \Scripts\Service-Fabric-Key-Vault\AddSecretsToKeyVault.cmd to add secrets to the key vault created in step 1.
4. Run \Scripts\Service-Fabric-Key-Vault\CreateAADApplication.ps1 to create an Azure AD application using the KeyVaultCertificate.cer certificate as credentials. It also creates the service principal for the application and associates the service principal with the key vault used by the application as repository for secrets.
5. Create a local folder (in **C:** for example), then copy the KeyVaultCertificate.pem and KeyVaultCertificate.key files there. These contain the public and private key of the certificate used by the front-end and back-end services for authenticating against Key Vault.
6. Open the docker-compose-override.yml file and use this version instead:

```
version: '3'

services:
  todoapi:
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - NotificationService__ServiceBus__QueueName=todoapi
      - DataProtection__BlobStorage__ContainerName=todoapi
      -
AzureKeyVault__Certificate__CertificateEnvironmentVariable=Certificates_TodoApiPkg_Code_To
doListCert_PEM
      -
AzureKeyVault__Certificate__KeyEnvironmentVariable=Certificates_TodoApiPkg_Code_TodoListCe
rt_PrivateKey
      - AzureKeyVault__ClientId=AZURE_AD_APPLICATION_ID
      - AzureKeyVault__Name=AZURE_KEY_VAULT_NAME
      - Certificates_TodoApiPkg_Code_TodoListCert_PEM=/pem/KeyVaultCertificate.pem
      - Certificates_TodoApiPkg_Code_TodoListCert_PrivateKey=/pem/KeyVaultCertificate.key
    ports:
      - "80"
    volumes:
      - C:\Temp\Pem:/pem

  todoweb:
```

```
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - TodoApiService__EndpointUri=todoapi
      - DataProtection__BlobStorage__ContainerName=todoweb
      -
AzureKeyVault__Certificate__CertificateEnvironmentVariable=Certificates_TodoApiPkg_Code_To
doListCert_PEM
      -
AzureKeyVault__Certificate__KeyEnvironmentVariable=Certificates_TodoApiPkg_Code_TodoListCe
rt_PrivateKey
      - AzureKeyVault__ClientId=AZURE_AD_APPLICATION_ID
      - AzureKeyVault__Name=AZURE_KEY_VAULT_NAME
      - Certificates_TodoApiPkg_Code_TodoListCert_PEM=/pem/KeyVaultCertificate.pem
      - Certificates_TodoApiPkg_Code_TodoListCert_PrivateKey=/pem/KeyVaultCertificate.key
    ports:
      - "80"
    volumes:
      - C:\Temp\Pem:/pem
```

7. Make the following changes to the docker-compose-override.yml file:
   a. Replace COSMOS_DB_ENDPOINT_URI with your Azure Cosmos DB endpoint URI.
   b. Replace COSMOS_DB_PRIMARY_KEY with your Azure Cosmos DB primary key.
   c. Replace SERVICE_BUS_CONNECTION_STRING with the connection string of your Service Bus Messaging namespace.
   d. Replace STORAGE_ACCOUNT_CONNECTION_STRING with the connection string of the storage account used by ASP.NET Core Data Protection
   e. Replace APPLICATION_INSIGHTS_INSTRUMENTATION_KEY with the instrumentation key of the Application Insights resource used to monitor the multi-container application.
   f. Replace AZURE_AD_APPLICATION_ID with the Application ID of the service principal used by the front-end and back end services to authenticate against Key Vault.
   g. Replace AZURE_KEY_VAULT_NAME with the name of the key vault used by the application.

Push the Docker images to Docker Hub or Container Registry

You can register and deploy your Docker images using repositories from either Docker Hub or Container Registry. This solution includes the scripts for both services.

Push Docker images to Docker Hub

To push to Docker Hub, execute the push-images-to-docker-hub.cmd command shown below. Make sure to replace the DOCKER_HUB_REPOSITORY and DOCKER_HUB_PASSWORD placeholders with your Docker Hub username and password.

```
REM login to docker hub
docker login -u DOCKER_HUB_REPOSITORY -p DOCKER_HUB_PASSWORD

REM tag the local todoapi:v1 image with the name of the DOCKER_HUB_REPOSITORY
docker tag todoapi:latest DOCKER_HUB_REPOSITORY/todoapi:v1

REM push the image DOCKER_HUB_REPOSITORY/todoapi:v1 to the DOCKER_HUB_REPOSITORY
docker push DOCKER_HUB_REPOSITORY/todoapi:v1

REM tag the local todoweb:v1 image with the name of the DOCKER_HUB_REPOSITORY
docker tag todoweb:latest DOCKER_HUB_REPOSITORY/todoweb:v1

REM push the image DOCKER_HUB_REPOSITORY/todoweb:v1 to the DOCKER_HUB_REPOSITORY
docker push DOCKER_HUB_REPOSITORY/todoweb:v1

REM browse to https://hub.docker.com/r/DOCKER_HUB_REPOSITORY/
start chrome https://hub.docker.com/r/DOCKER_HUB_REPOSITORY/
```

Create and push to Container Registry

Container Registry is a managed Docker registry service based on the open-source Docker Registry 2.0. This repository can store images for container deployments in Container Service, Azure App Service, Azure Batch, Service Fabric, and others. For more information about Container Registry, see Introduction to private Docker container registries in Azure.

To create a Container Registry, run create-azure-container-registry.cmd.

```
REM Create a resource group for the Azure Container Registry
az group create --name ContainerRegistryResourceGroup --location westus2 --output jsonc

REM Create an Azure Container Registry. The name of the Container Registry must be unique
az acr create --resource-group ContainerRegistryResourceGroup --name
AZURE_CONTAINER_REGISTRY --sku Basic --admin-enabled true

REM Login to the newly created Azure Container Registry
az acr login --name AZURE_CONTAINER_REGISTRY
```

To tag and register the images in your repository on Docker Hub, execute the push-images-to-azure-container-registry.cmd command file. Make sure to replace the AZURE_CONTAINER_REGISTRY placeholder with the name of your Container Registry.

```
REM Login to the newly created Azure Container Registry
call az acr login --name AZURE_CONTAINER_REGISTRY

REM Each container image needs to be tagged with the loginServer name of the registry.
REM This tag is used for routing when pushing container images to an image registry.
REM Save the loginServer name to the AKS_CONTAINER_REGISTRY environment variable.
for /f "delims=" %%a in ('call az acr list --resource-group ContainerRegistryResourceGroup
--query "[].{acrLoginServer:loginServer}" --output tsv') do @set
AKS_CONTAINER_REGISTRY=%%a

REM tag the local todoapi:v1 image with the loginServer of the container registry
docker tag todoapi:v1 %AKS_CONTAINER_REGISTRY%/todoapi:v1
```

```
REM publish <container registry>/todoapi:v1 to the container registry on Azure
docker push %AKS_CONTAINER_REGISTRY%/todoapi:v1

REM tag the local todoweb:v1 image with the loginServer of the container registry
docker tag todoweb:v1 %AKS_CONTAINER_REGISTRY%/todoweb:v1

REM publish <container registry>/todoweb:v1 to the container registry on Azure
docker push %AKS_CONTAINER_REGISTRY%/todoweb:v1

REM List images in the container registry on Azure
call az acr repository list --name AZURE_CONTAINER_REGISTRY --output table
```

Store secrets in Key Vault

If you plan to deploy the application to a Service Fabric cluster in Azure, you can store sensitive data such as connection strings, passwords, and instrumentation keys in Key Vault. The front-end and back-end services that compose the multi-container application in this sample are ASP.NET Core projects. ASP.NET Core supplies a configuration provider for Key Vault in the Microsoft.Extensions.Configuration.AzureKeyVault NuGet package. This configuration provider allows an application to use the application ID and application key of an Azure AD application to authenticate against Key Vault.

The Azure Key Vault configuration provider in the ASP.NET Core documentation goes into more detail. However, the documentation explains an unsafe approach that defines the **ClientId** and **ClientSecret** settings in the service configuration. A malicious user could use these credentials to access secrets in Key Vault. Another way to authenticate an Azure AD application is to use a client ID and a certificate instead of a client ID and a client secret.

The documentation also explains how to use the Key Vault configuration provider to load application configuration values from Key Vault secrets. First you must get or create a certificate. Then you need to associate the certificate with an Azure AD application. Finally, you must add code to your ASP.NET Core application to read and use the certificate to access Key Vault. The sections below describe these steps in more detail. See also Microsoft.Extensions.Configuration.AzureKeyVault.

Get or create a certificate

For this project, you can make a test certificate. It takes just a few commands in a command prompt.

1. Run the following PowerShell script to create and export a self-signed .pfx certificate.

   CreateSelfSignedPfxCertificate.ps1

```
# Cluster Name
$clusterName = "MyCluster"

# Cluster Location
$clusterRegion = "WestEurope"
```

```
# Certificate Password
$certificatePassword = 'passw@rd123!'

# Secure Password
$securePassword = ConvertTo-SecureString -String $certificatePassword `
                                         -AsPlainText `
                                         -Force

# The certificate's subject name must match the domain used to access the Service Fabric
cluster
$certificateDNSName = $clusterName.ToLower() + "." + $clusterRegion.ToLower() +
".cloudapp.azure.com"

# Path of Service Fabric cluster primary certificate file
$primaryCertificateFileFullPath = $PSScriptRoot + '\' + "KeyVaultCertificate.pfx"

# Service Fabric cluster primary certificate file
$primaryCertificate = New-SelfSignedCertificate -CertStoreLocation Cert:\CurrentUser\My `
                                                -DnsName $certificateDNSName

# Export Service Fabric cluster secondary certificate file
Export-PfxCertificate -FilePath $primaryCertificateFileFullPath `
                      -Password $securePassword `
                      -Cert $primaryCertificate
```

2. Make a note of the password for the .pfx (in this example, *passw@rd123!*). You need this later. Then:

   a. If you are planning to deploy the sample application to a Service Fabric cluster on Windows, the .pfx and .cer files that are created are all you need.

   b. If you plan to deploy the application to a Service Fabric cluster on Linux, and you want to debug the front-end and back-end services locally, you must create a .pem file and a .key file. Start from the .pfx file and use the following commands:

```
openssl pkcs12 -in KeyVaultCertificate.pfx -out KeyVaultCertificatePEM.pem -nodes -nokeys
openssl pkcs12 -in KeyVaultCertificate.pfx -out KeyVaultCertificatePEM.key -nodes -nocerts
```

Create a key vault and add secrets using Azure CLI

To protect sensitive data from unauthorized users, you should store secrets in Key Vault. You can use CreateKeyVault.cmd to create a key vault:

```
REM Create a Resource Group for Key Vault
call az group create --name TodoListKeyVaultResourceGroup --location WestEurope

REM Create Key Vault
call az keyvault create --name TodoListKeyVault --resource-group
TodoListKeyVaultResourceGroup
```

To add sensitive configuration data to your key vault, you can use AddSecretsToKeyVault.cmd. Before running it, make the following changes:

- Replace COSMOS_DB_ENDPOINT_URI with your Azure Cosmos DB endpoint URI.
- Replace COSMOS_DB_PRIMARY_KEY with your Azure Cosmos DB primary key.
- Replace COSMOS_DB_DATABASE_NAME with the Azure Cosmos DB database name.
- Replace COSMOS_DB_COLLECTION_NAME with the Azure Cosmos DB collection name.
- Replace SERVICE_BUS_CONNECTION_STRING with the connection string of your Service Bus Messaging namespace.
- Replace STORAGE_ACCOUNT_CONNECTION_STRING with the connection string of the Storage Account used by ASP.NET Core Data Protection.
- Replace APPLICATION_INSIGHTS_INSTRUMENTATION_KEY with the instrumentation key of the Application Insights resource used to monitor the multi-container application.

AddSecretsToKeyVault.cmd

```
REM add Azure Cosmos DB Endpoint URI secret to Key Vault
call az keyvault secret set --name RepositoryService--CosmosDb--EndpointUri --vault-name
TodoListKeyVault  --value "COSMOS_DB_ENDPOINT_URI" --description "Azure Cosmos DB endpoint
URI"

REM add Azure Cosmos DB Primary Key secret to Key Vault
call az keyvault secret set --name RepositoryService--CosmosDb--PrimaryKey --vault-name
TodoListKeyVault  --value "COSMOS_DB_PRIMARY_KEY --description "Azure Cosmos DB primary
key"

REM add Azure Cosmos DB Database Name secret to Key Vault
call az keyvault secret set --name RepositoryService--CosmosDb--DatabaseName --vault-name
TodoListKeyVault  --value "COSMOS_DB_DATABASE_NAME" --description "Azure Cosmos DB
database name"

REM add Azure Cosmos DB Collection Name secret to Key Vault
call az keyvault secret set --name RepositoryService--CosmosDb--CollectionName --vault-
name TodoListKeyVault  --value "COSMOS_DB_COLLECTION_NAME" --description "Azure Cosmos DB
collection name"

REM add Service Bus Connection String secret to Key Vault
call az keyvault secret set --name NotificationService--ServiceBus--ConnectionString --
vault-name TodoListKeyVault  --value "SERVICE_BUS_CONNECTION_STRING" --description
"Service Bus connection string"

REM add Data Protection Blob Storage Connection String secret to Key Vault
call az keyvault secret set --name DataProtection--BlobStorage--ConnectionString --vault-
name TodoListKeyVault  --value "STORAGE_ACCOUNT_CONNECTION_STRING" --description "Data
Protection blob storage connection string"

REM add Application Insights Instrumentation Key secret to Key Vault
call az keyvault secret set --name ApplicationInsights--InstrumentationKey --vault-name
TodoListKeyVault  --value "APPLICATION_INSIGHTS_INSTRUMENTATION_KEY" --description
"Application Insights instrumentation key"

REM List secrets in Key Vault
call az keyvault secret list --vault-name TodoListKeyVault --output table
```

Associate the certificate with an Azure AD application

The next step is to associate the certificate with a new Azure AD application called ServiceFabricTodoListApp. Using PowerShell, run the commands in CreateAADApplication.ps1:

```
# Login to Azure Resource Manager
Login-AzureRmAccount

# Select a default subscription for your current session in case your account has multiple
Azure subscriptions
Get-AzureRmSubscription –SubscriptionName "SUBSCRIPTION-NAME" | Select-AzureRmSubscription

# Variables
$pfxFile = $PSScriptRoot + '\KeyVaultCertificate.cer'
$displayName = "ServiceFabricTodoListApp"
$appUrl = "http://ServiceFabricTodoListApp"
$keyVaultName = "TodoListKeyVault"
$keyVaultResourceGroup = "TodoListKeyVaultResourceGroup"

# Get certificate from file
$x509 = New-Object System.Security.Cryptography.X509Certificates.X509Certificate2
$x509.Import($pfxFile)

# Get certificate raw data in base64 format
$credValue = [System.Convert]::ToBase64String($x509.GetRawCertData())

# Create a new Azure AD application for TodoListApp
$adapp = New-AzureRmADApplication -DisplayName $displayName `
                                  -HomePage $appUrl `
                                  -IdentifierUris $appUrl `
                                  -CertValue $credValue `
                                  -StartDate $x509.NotBefore `
                                  -EndDate $x509.NotAfter

# Create a new Azure AD service principal for the Azure AD TodoListApp application
$sp = New-AzureRmADServicePrincipal -ApplicationId $adapp.ApplicationId

# Grants permissions for a user, application, or security group to perform operations with
a key vault.
Set-AzureRmKeyVaultAccessPolicy -VaultName $keyVaultName `
                                -ResourceGroupName $keyVaultResourceGroup `
                                -ServicePrincipalName $adapp.ApplicationId `
                                -PermissionsToSecrets all

# get the thumbprint to use in your app settings
$x509.Thumbprint

# get the application id to use in your app settings
$adapp.ApplicationId
```

After running these commands, the application appears in Azure AD. For more information, see Application Objects and Service Principal Objects.

ⓘ **NOTE:** In the output of the script, make a note of the certificate thumbprint and application ID.

How Service Fabric passes a certificate to a container

Service Fabric provides a mechanism that allows services running inside a container to access a certificate installed on the nodes of a cluster on Windows or Linux. You can help secure your container services by specifying a certificate. The certificate information is provided in the application manifest under the **ContainerHostPolicies** tag as shown:

```
<ContainerHostPolicies CodePackageRef="Code">
    <CertificateRef Name="MyCert1" X509StoreName="My" X509FindValue="[Thumbprint1]"/>
    <CertificateRef Name="MyCert2" X509FindValue="[Thumbprint2]"/>
```

The way the certifications are read depends on whether your clusters run on Windows or Linux:

- For Windows clusters, when starting the application, the runtime reads the certificates and generates a .pfx file and password for each certificate. This .pfx file and password file are accessible inside the container using the **Certificates_ServicePackageName_CodePackageName_CertName_PFX** and **Certificates_ServicePackageName_CodePackageName_CertName_Password** environment variables. In a Service Fabric Windows cluster in Azure, the **todoapi** back-end service retrieves the location of the .pfx file password files from the **Certificates_TodoWebPkg_Code_TodoListCert_PFX** and **Certificates_TodoWebPkg_Code_TodoListCert_Password** environment variables.
- For Linux clusters, the certificates(.pem files) are copied from the store specified by X509StoreName onto the container. The **Certificates_ServicePackageName_CodePackageName_CertName_PEM** and **Certificates_ServicePackageName_CodePackageName_CertName_PrivateKey** environment variables are used. In a Service Fabric Linux cluster in Azure, the **todoapi** back-end service retrieves the location of the .pfx file password files from the **Certificates_TodoWebPkg_Code_TodoListCert_PEM** and **Certificates_TodoWebPkg_Code_TodoListCert_PrivateKey** environment variables.

Alternatively, if you already have the certificate in the required form and want to access it inside the container, you can create a data package inside your app package and specify the following inside your application manifest:

```
<ContainerHostPolicies CodePackageRef="NodeContainerService.Code">
        <CertificateRef Name="MyCert1" DataPackageRef="[DataPackageName]" DataPackageVersi
on="[Version]" RelativePath="[Relative Path to certificate inside DataPackage]" Password="
[password]" IsPasswordEncrypted="[true/false]"/>
```

For more information about configuring certificates for a containerized service in Service Fabric, see Container Security. For more information about managing the certificates, see Add or remove certificates for a Service Fabric cluster in Azure.

Read the certificate from code and initialize Key Vault configuration provider

When deploying the application to a Service Fabric cluster in Azure, you need to specify a certificate in the **CertificateRef** element inside the **ContainerHostPolicies** settings of both the front-end and back-end service, using one of the techniques described in the previous section. Service Fabric copies the certificate files inside the container and creates two environment variables that contain the path of:

- .pfx and password files in a Windows cluster.
- .pem and .key files in a Linux cluster.

Below you can see the code used by the **Program** class of the **TodoApi** service:

```
using System;
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;

namespace TodoApi
{
    public class Program
    {
        public static void Main(string[] args)
        {
            BuildWebHost(args).Run();
        }

        public static IWebHost BuildWebHost(string[] args)
        {
            var builder = WebHost.CreateDefaultBuilder(args)
                .UseApplicationInsights()
                .CaptureStartupErrors(true)
                .UseSetting(WebHostDefaults.DetailedErrorsKey, "true")
                .ConfigureAppConfiguration(ConfigConfiguration)
                .UseStartup<Startup>()
                .Build();

            return builder;
        }

        private static void ConfigConfiguration(WebHostBuilderContext webHostBuilderContex
t, IConfigurationBuilder configurationBuilder)
        {
            var configuration = configurationBuilder.Build();

            // Read the name of the environment variable set by Service Fabric that contai
n the location of the PEM file
            var certificateEnvironmentVariable = configuration["AzureKeyVault:Certificate:
CertificateEnvironmentVariable"];
            if (string.IsNullOrWhiteSpace(certificateEnvironmentVariable))
            {
                return;
            }

            // Read the name of the environment variable set by Service Fabric that contai
n the location of the KEY file
            var keyEnvironmentVariable = configuration["AzureKeyVault:Certificate:KeyEnvir
onmentVariable"];
            if (string.IsNullOrWhiteSpace(keyEnvironmentVariable))
            {
```

```
                return;
            }

            // Read the client ID
            var clientId = configuration["AzureKeyVault:ClientId"];
            if (string.IsNullOrWhiteSpace(clientId))
            {
                return;
            }

            // Read the name of the Azure Key Vault
            var name = configuration["AzureKeyVault:Name"];
            if (string.IsNullOrWhiteSpace(name))
            {
                return;
            }

            // Read the location of the certificate file from the environment variable
            var certificateFilePath = Environment.GetEnvironmentVariable(certificateEnviro
nmentVariable);
            if (string.IsNullOrWhiteSpace(certificateFilePath))
            {
                return;
            }

            // Read the location of the key file from the environment variable
            var keyFilePath = Environment.GetEnvironmentVariable(keyEnvironmentVariable);
            if (string.IsNullOrWhiteSpace(keyFilePath))
            {
                return;
            }

            // Read the certificate used to authenticate against Azure Key Vault
            var certificate = Helpers.CertificateHelper.GetCertificateAsync(certificateFil
ePath, keyFilePath).Result;
            if (certificate == null)
            {
                return;
            }

            // Configure the application to read settings from Azure Key Vault
            configurationBuilder.AddAzureKeyVault($"https://{name}.vault.azure.net/",
                                                  clientId,
                                                  certificate);
        }
    }
}
```

The **ConfigureAppConfiguration** method does the following:

- Reads the name of the environment variable initialized by Service Fabric that holds the path of the .pfx (Windows) or .pem (Linux) file from the **AzureKeyVault__Certificate__CertificateEnvironmentVariable** environment variable.
- Reads the name of the environment variable initialized by Service Fabric which holds the path of the password (Windows) or .key (Linux) file from the **AzureKeyVault__Certificate__KeyEnvironmentVariable** environment variable.
- Reads the Azure AD application ID used to authenticate against Key Vault from the **AzureKeyVault__ClientId** environment variable.
- Reads the name of the key vault from the **AzureKeyVault__Name** environment variable.
- Reads the path of the .pfx (Windows) or .pem (Linux) file from the environment variable initialized by Service Fabric.
- Reads the path of the password (Windows) or .pem file (Linux) from the environment variable initialized by Service Fabric.
- Calls the **CertificateHelper.GetCertificateAsync** method (shown below) to retrieve a X509Certificate2 object containing the certificate used to authenticate against Key Vault.
- Add the Key Vault configuration provider to the application.

ⓘ **NOTE:** To read certificates in a Linux cluster, the application uses the classes contained in the System.Security.Cryptography.OpenSsl NuGet package.

CertificateHelper.GetCertificateAsync method

```
public static async Task<X509Certificate2> GetCertificateAsync(string certificateFilePath,
 string keyFilePath)
{
    // Validate parameters
    if (string.IsNullOrEmpty(certificateFilePath))
    {
        throw new ArgumentException($"{nameof(certificateFilePath)} parameter cannot be nu
ll or empty.", nameof(certificateFilePath));
    }

    if (string.IsNullOrEmpty(keyFilePath))
    {
        throw new ArgumentException($"{nameof(keyFilePath)} parameter cannot be null or em
pty.", nameof(keyFilePath));
    }

    if (!File.Exists(certificateFilePath))
    {
        throw new FileNotFoundException($"{certificateFilePath} file not found.", certific
ateFilePath);
    }

    if (!File.Exists(keyFilePath))
```

```
    {
        throw new FileNotFoundException($"{keyFilePath} file not found.", keyFilePath);
    }

    if (Environment.OSVersion.Platform.ToString().ToLower().Contains("win"))
    {
        SetReadPermission(certificateFilePath);
        SetReadPermission(keyFilePath);
        var password = File.ReadAllLines(keyFilePath, Encoding.Default)[0];
        password = password.Replace("\0", string.Empty);
        var certificate = new X509Certificate2(certificateFilePath, password);
        return certificate;
    }
    else
    {
        var pemCertificate = await File.ReadAllTextAsync(certificateFilePath);
        var pemKey = await File.ReadAllTextAsync(keyFilePath);

        var certBuffer = GetBytesFromPem(pemCertificate, CertificateFileType.Certificate);
        var keyBuffer = GetBytesFromPem(pemKey, CertificateFileType.Pkcs8PrivateKey);

        var certificate = new X509Certificate2(certBuffer);
        var privateKey = DecodePrivateKeyInfo(keyBuffer);
        certificate = certificate.CopyWithPrivateKey(privateKey);
        return certificate;
    }
}
```

Use application and service manifests for a Service Fabric deployment

In the GitHub solution, you can find three projects to deploy the multi-container application to a Service Fabric cluster using Service Fabric's native application packaging and description mechanisms.

- The **TodoAppForLinuxFromACR** project deploys the multi-container application to a cluster on Linux that pulls the Docker images from Container Registry. This project does not require the secrets parameters to be stored in Key Vault but for testing purposes only allows them to be specified in cleartext in the Cloud.xml file. Do not use this approach in a production environment. Store sensitive configuration data in Key Vault. Unauthorized users may be able to read sensitive data such as connection strings, keys, and passwords from the manifests stored in GitHub or another source code repository. After the application is deployed, users may be able to retrieve this data from the Service Fabric Explorer dashboard.

- The **TodoAppForLinuxFromDockerHub** project deploys the multi-container application to a cluster on Linux that pulls the Docker images from Docker Hub. The front-end and back-end services read non sensitive configuration data in cleartext from the Cloud.xml file, while they retrieve sensitive configuration data is read from Key Vault.

- The **TodoAppForWindowsFromDockerHub** project deploys the multi-container application to a cluster on Windows that pulls the Docker images for Windows containers from a Docker Hub repository. The front-end and back-end services read non sensitive configuration data in cleartext from the Cloud.xml file, while they retrieve sensitive configuration data is read from Key Vault. Note that the manifests for this project are on GitHub only, not in this document.

ⓘ **NOTE:** For greater security, it is recommended to encrypt the repository password using an encipherment certificate deployed on all nodes of the cluster. When Service Fabric deploys the service package to the cluster, the encipherment certificate is used to decrypt the cipher text. The **Invoke-ServiceFabricEncryptText** cmdlet is used to create the cipher text for the password, which is added to the ApplicationManifest.xml file. If you encrypt parameters in the application manifest or Cloud.xml, and then you want to decrypt them in the code, you have two options:

- If the service uses the Service Fabric SDK and Settings.xml, you can simply set the value of the **IsEncrypted** attribute to *true* for any encrypted parameter.

- If the service, like this sample deployment, does not use the Service Fabric SDK, or it uses environment variables to pass configuration data to services, you should pass the certificate used to encrypt parameters to the service and add code to decrypt their value.

Linux: manifests and parameters for deployment

This section shows the service manifests, application manifest, and application parameters file contained in this project. Notice that in the service manifests below, all configuration data is passed to each service using environment variables.

\TodoAppForLinuxFromACR\ApplicationPackageRoot\TodoApiPkg\ServiceManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<ServiceManifest Name="TodoApiPkg"
                 Version="1.0.0"
                 xmlns="http://schemas.microsoft.com/2011/01/fabric"
                 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <ServiceTypes>
    <!-- This is the name of your ServiceType.
         The UseImplicitHost attribute indicates this is a guest service. -->
    <StatelessServiceType ServiceTypeName="TodoApiType" UseImplicitHost="true" />
  </ServiceTypes>

  <!-- Code package is your service executable. -->
  <CodePackage Name="Code" Version="1.0.0">
    <EntryPoint>
      <!-
- Follow this link for more information about deploying Windows containers to Service Fabr
ic: https://aka.ms/sfguestcontainers -->
      <ContainerHost>
        <ImageName>AZURE_CONTAINER_REGISTRY_NAME.azurecr.io/todoapi:v1</ImageName>
      </ContainerHost>
    </EntryPoint>
    <!-- Pass environment variables to your container: -->
```

```
    <EnvironmentVariables>
      <EnvironmentVariable Name="ASPNETCORE_ENVIRONMENT" Value=""/>
      <EnvironmentVariable Name="RepositoryService__CosmosDb__EndpointUri" Value=""/>
      <EnvironmentVariable Name="RepositoryService__CosmosDb__PrimaryKey" Value=""/>
      <EnvironmentVariable Name="RepositoryService__CosmosDb__DatabaseName" Value=""/>
      <EnvironmentVariable Name="RepositoryService__CosmosDb__CollectionName" Value=""/>
      <EnvironmentVariable Name="NotificationService__ServiceBus__ConnectionString" Value=
""/>
      <EnvironmentVariable Name="NotificationService__ServiceBus__QueueName" Value=""/>
      <EnvironmentVariable Name="DataProtection__BlobStorage__ConnectionString" Value=""/>
      <EnvironmentVariable Name="DataProtection__BlobStorage__ContainerName" Value=""/>
      <EnvironmentVariable Name="ApplicationInsights__InstrumentationKey" Value=""/>
    </EnvironmentVariables>
  </CodePackage>

  <!-
- Config package is the contents of the Config directoy under PackageRoot that contains an

       independently-
updateable and versioned set of custom configuration settings for your service. -->
  <ConfigPackage Name="Config" Version="1.0.0" />

  <Resources>
    <Endpoints>
      <!-
- This endpoint is used by the communication listener to obtain the port on which to
           listen. Please note that if your service is partitioned, this port is shared wi
th
           replicas of different partitions that are placed in your code. -->
      <Endpoint Name="TodoApiEndpoint" Port="80" UriScheme="http" Protocol="http"/>
    </Endpoints>
  </Resources>
</ServiceManifest>
```

\TodoAppForLinuxFromACR\ApplicationPackageRoot\TodoWebPkg\ServiceManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<ServiceManifest Name="TodoWebPkg"
                 Version="1.0.0"
                 xmlns="http://schemas.microsoft.com/2011/01/fabric"
                 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <ServiceTypes>
    <!-- This is the name of your ServiceType.
         The UseImplicitHost attribute indicates this is a guest service. -->
    <StatelessServiceType ServiceTypeName="TodoWebType" UseImplicitHost="true" />
  </ServiceTypes>

  <!-- Code package is your service executable. -->
  <CodePackage Name="Code" Version="1.0.0">
    <EntryPoint>
```

```
      <!-
- Follow this link for more information about deploying Windows containers to Service Fabr
ic: https://aka.ms/sfguestcontainers -->
      <ContainerHost>
        <ImageName>AZURE_CONTAINER_REGISTRY_NAME.azurecr.io/todoweb:v1</ImageName>
      </ContainerHost>
    </EntryPoint>
    <!-- Pass environment variables to your container: -->
    <EnvironmentVariables>
      <EnvironmentVariable Name="ASPNETCORE_ENVIRONMENT" Value=""/>
      <EnvironmentVariable Name="TodoApiService__EndpointUri" Value=""/>
      <EnvironmentVariable Name="DataProtection__BlobStorage__ConnectionString" Value=""/>
      <EnvironmentVariable Name="DataProtection__BlobStorage__ContainerName" Value=""/>
      <EnvironmentVariable Name="ApplicationInsights__InstrumentationKey" Value=""/>
    </EnvironmentVariables>
  </CodePackage>

  <!-
- Config package is the contents of the Config directoy under PackageRoot that contains an

       independently-
updateable and versioned set of custom configuration settings for your service. -->
  <ConfigPackage Name="Config" Version="1.0.0" />

  <Resources>
    <Endpoints>
      <!-
- This endpoint is used by the communication listener to obtain the port on which to
           listen. Please note that if your service is partitioned, this port is shared wi
th
           replicas of different partitions that are placed in your code. -->
      <Endpoint Name="TodoWebEndpoint" Port="8080" UriScheme="http" Protocol="http"/>
    </Endpoints>
  </Resources>
</ServiceManifest>
```

In the application manifest below, notice that the certificate used by each service must be installed in the Service Fabric cluster before deploying the application. In fact, the certificate identified by its thumbprint needs to exist in the cluster nodes before Docker containers are created for each service.

\TodoAppForLinuxFromACR\ApplicationPackageRoot\ApplicationManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<ApplicationManifest ApplicationTypeName="TodoAppType"
                     ApplicationTypeVersion="1.0.0"
                     xmlns="http://schemas.microsoft.com/2011/01/fabric"
                     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Parameters>
    <Parameter Name="ACR_Username" DefaultValue="" />
    <Parameter Name="ACR_Password" DefaultValue="" />
    <Parameter Name="TodoWeb_InstanceCount" DefaultValue="-1" />
```

```
    <Parameter Name="TodoWeb_ASPNETCORE_ENVIRONMENT" DefaultValue="Development"/>
    <Parameter Name="TodoWeb_TodoApiService__EndpointUri" DefaultValue=""/>
    <Parameter Name="TodoWeb_DataProtection__BlobStorage__ConnectionString" DefaultValue="
"/>
    <Parameter Name="TodoWeb_DataProtection__BlobStorage__ContainerName" DefaultValue=""/>
    <Parameter Name="TodoWeb_ApplicationInsights__InstrumentationKey" DefaultValue=""/>
    <Parameter Name="TodoApi_InstanceCount" DefaultValue="-1" />
    <Parameter Name="TodoApi_ASPNETCORE_ENVIRONMENT" DefaultValue="Development"/>
    <Parameter Name="TodoApi_RepositoryService__CosmosDb__EndpointUri" DefaultValue=""/>
    <Parameter Name="TodoApi_RepositoryService__CosmosDb__PrimaryKey" DefaultValue=""/>
    <Parameter Name="TodoApi_RepositoryService__CosmosDb__DatabaseName" DefaultValue=""/>
    <Parameter Name="TodoApi_RepositoryService__CosmosDb__CollectionName" DefaultValue=""/
>
    <Parameter Name="TodoApi_NotificationService__ServiceBus__ConnectionString" DefaultVal
ue=""/>
    <Parameter Name="TodoApi_NotificationService__ServiceBus__QueueName" DefaultValue=""/>
    <Parameter Name="TodoApi_DataProtection__BlobStorage__ConnectionString" DefaultValue="
"/>
    <Parameter Name="TodoApi_DataProtection__BlobStorage__ContainerName" DefaultValue=""/>
    <Parameter Name="TodoApi_ApplicationInsights__InstrumentationKey" DefaultValue=""/>
  </Parameters>
  <!-
- Import the ServiceManifest from the ServicePackage. The ServiceManifestName and ServiceM
anifestVersion
       should match the Name and Version attributes of the ServiceManifest element defined
 in the
       ServiceManifest.xml file. -->
  <ServiceManifestImport>
    <ServiceManifestRef ServiceManifestName="TodoWebPkg" ServiceManifestVersion="1.0.0" />
    <ConfigOverrides />
    <EnvironmentOverrides CodePackageRef="Code">
      <EnvironmentVariable Name="ASPNETCORE_ENVIRONMENT" Value="[TodoWeb_ASPNETCORE_ENVIRO
NMENT]"/>
      <EnvironmentVariable Name="TodoApiService__EndpointUri" Value="[TodoWeb_TodoApiServi
ce__EndpointUri]"/>
      <EnvironmentVariable Name="DataProtection__BlobStorage__ConnectionString" Value="[To
doWeb_DataProtection__BlobStorage__ConnectionString]"/>
      <EnvironmentVariable Name="DataProtection__BlobStorage__ContainerName" Value="[TodoW
eb_DataProtection__BlobStorage__ContainerName]"/>
      <EnvironmentVariable Name="ApplicationInsights__InstrumentationKey" Value="[TodoWeb_
ApplicationInsights__InstrumentationKey]"/>
    </EnvironmentOverrides>
    <Policies>
      <ContainerHostPolicies CodePackageRef="Code">
        <RepositoryCredentials AccountName="[ACR_Username]" Password="[ACR_Password]" Pass
wordEncrypted="false"/>
        <PortBinding ContainerPort="80" EndpointRef="TodoWebEndpoint" />
      </ContainerHostPolicies>
    </Policies>
  </ServiceManifestImport>
  <ServiceManifestImport>
    <ServiceManifestRef ServiceManifestName="TodoApiPkg" ServiceManifestVersion="1.0.0" />
    <ConfigOverrides />
```

```
    <EnvironmentOverrides CodePackageRef="Code">
      <EnvironmentVariable Name="ASPNETCORE_ENVIRONMENT" Value="[TodoApi_ASPNETCORE_ENVIRO
NMENT]"/>
      <EnvironmentVariable Name="RepositoryService__CosmosDb__EndpointUri" Value="[TodoApi
_RepositoryService__CosmosDb__EndpointUri]"/>
      <EnvironmentVariable Name="RepositoryService__CosmosDb__PrimaryKey" Value="[TodoApi_
RepositoryService__CosmosDb__PrimaryKey]"/>
      <EnvironmentVariable Name="RepositoryService__CosmosDb__DatabaseName" Value="[TodoAp
i_RepositoryService__CosmosDb__DatabaseName]"/>
      <EnvironmentVariable Name="RepositoryService__CosmosDb__CollectionName" Value="[Todo
Api_RepositoryService__CosmosDb__CollectionName]"/>
      <EnvironmentVariable Name="NotificationService__ServiceBus__ConnectionString" Value=
"[TodoApi_NotificationService__ServiceBus__ConnectionString]"/>
      <EnvironmentVariable Name="NotificationService__ServiceBus__QueueName" Value="[TodoA
pi_NotificationService__ServiceBus__QueueName]"/>
      <EnvironmentVariable Name="DataProtection__BlobStorage__ConnectionString" Value="[To
dApi_DataProtection__BlobStorage__ConnectionString]"/>
      <EnvironmentVariable Name="DataProtection__BlobStorage__ContainerName" Value="[TodoA
pi_DataProtection__BlobStorage__ContainerName]"/>
      <EnvironmentVariable Name="ApplicationInsights__InstrumentationKey" Value="[TodoApi_
ApplicationInsights__InstrumentationKey]"/>
    </EnvironmentOverrides>
    <Policies>
      <ContainerHostPolicies CodePackageRef="Code">
        <RepositoryCredentials AccountName="[ACR_Username]" Password="[ACR_Password]" Pass
wordEncrypted="false"/>
        <PortBinding ContainerPort="80" EndpointRef="TodoApiEndpoint" />
      </ContainerHostPolicies>
    </Policies>
  </ServiceManifestImport>
  <DefaultServices>
    <!-- The section below creates instances of service types, when an instance of this
         application type is created. You can also create one or more instances of service
 type using the
         ServiceFabric PowerShell module.

         The attribute ServiceTypeName below must match the name defined in the imported S
erviceManifest.xml file. -->
    <Service Name="TodoWeb" ServicePackageActivationMode="ExclusiveProcess" ServiceDnsName
="todoweb.todoapp">
      <StatelessService ServiceTypeName="TodoWebType" InstanceCount="[TodoWeb_InstanceCoun
t]">
        <SingletonPartition />
      </StatelessService>
    </Service>
    <Service Name="TodoApi" ServicePackageActivationMode="ExclusiveProcess" ServiceDnsName
="todoapi.todoapp">
      <StatelessService ServiceTypeName="TodoApiType" InstanceCount="[TodoApi_InstanceCoun
t]">
        <SingletonPartition />
      </StatelessService>
    </Service>
  </DefaultServices>
</ApplicationManifest>
```

\TodoAppForLinuxFromACR\ApplicationParameters\Cloud.xml

```
<?xml version="1.0" encoding="utf-8"?>
<Application Name="fabric:/TodoApp" xmlns="http://schemas.microsoft.com/2011/01/fabric" xm
lns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
  <Parameters>
    <Parameter Name="ACR_Username" Value="ACR_USERNAME" />
    <Parameter Name="ACR_Password" Value="ACR_PASSWORD" />
    <Parameter Name="TodoWeb_InstanceCount" Value="-1" />
    <Parameter Name="TodoWeb_ASPNETCORE_ENVIRONMENT" Value="Development"/>
    <Parameter Name="TodoWeb_TodoApiService__EndpointUri" Value="todoapi.todoapp"/>
    <Parameter Name="TodoWeb_DataProtection__BlobStorage__ConnectionString" Value="STORAGE
_ACCOUNT_CONNECTION_STRING"/>
    <Parameter Name="TodoWeb_DataProtection__BlobStorage__ContainerName" Value="todoweb"/>
    <Parameter Name="TodoWeb_ApplicationInsights__InstrumentationKey" Value="APPLICATION_I
NSIGHTS_INSTRUMENTATION_KEY"/>
    <Parameter Name="TodoApi_InstanceCount" Value="-1" />
    <Parameter Name="TodoApi_ASPNETCORE_ENVIRONMENT" Value="Development"/>
    <Parameter Name="TodoApi_RepositoryService__CosmosDb__EndpointUri" Value="COSMOS_DB_EN
DPOINT_URI"/>
    <Parameter Name="TodoApi_RepositoryService__CosmosDb__PrimaryKey" Value="COSMOS_DB_PRI
MARY_KEY"/>
    <Parameter Name="TodoApi_RepositoryService__CosmosDb__DatabaseName" Value="COSMOS_DB_D
ATABASE_NAME"/>
    <Parameter Name="TodoApi_RepositoryService__CosmosDb__CollectionName" Value="COSMOS_DB
_COLLECTION_NAME"/>
    <Parameter Name="TodoApi_NotificationService__ServiceBus__ConnectionString" Value="SER
VICE_BUS_CONNECTION_STRING"/>
    <Parameter Name="TodoApi_NotificationService__ServiceBus__QueueName" Value="todoapi"/>
    <Parameter Name="TodoApi_DataProtection__BlobStorage__ConnectionString" Value="STORAGE
_ACCOUNT_CONNECTION_STRING"/>
    <Parameter Name="TodoApi_DataProtection__BlobStorage__ContainerName" Value="todoapi"/>
    <Parameter Name="TodoApi_ApplicationInsights__InstrumentationKey" Value="APPLICATION_I
NSIGHTS_INSTRUMENTATION_KEY"/>
  </Parameters>
</Application>
```

Configure the manifests for deployment to a cluster on Linux

Before deploying the application to your Service Fabric cluster on Linux, make sure to configure the value of the parameters used by the front-end and back-end services in the Cloud.xml file under the **ApplicationParameters** folder and in the manifests.

Do the following:

1. Open Cloud.xml and replace the placeholders as follows:
    a. Replace ACR_USERNAME with your Container Registry username.
    b. Replace ACR_PASSWORD with your Container Registry password.
    c. Replace COSMOS_DB_ENDPOINT_URI with your Azure Cosmos DB endpoint URI.
    d. Replace COSMOS_DB_PRIMARY_KEY with your Azure Cosmos DB primary key.

e. Replace COSMOS_DB_DATABASE_NAME with the name of the Azure Cosmos DB database.

f. Replace COSMOS_DB_COLLECTION_NAME with the name of the Azure Cosmos DB collection.

g. Replace SERVICE_BUS_CONNECTION_STRING with the connection string of your Service Bus Messaging namespace.

h. Replace STORAGE_ACCOUNT_CONNECTION_STRING with the connection string of the Storage Account used by ASP.NET Core Data Protection.

i. Replace APPLICATION_INSIGHTS_INSTRUMENTATION_KEY with the instrumentation key of the Application Insights resource used to monitor the multi-container application.

2. Open \TodoAppForLinuxFromACR\TodoApiServiceManifest.xml. Replace AZURE_CONTAINER_REGISTRY_NAME with the name of your Container Registry.

3. Open \TodoAppForLinuxFromACR\TodoWebServiceManifest.xml. Replace AZURE_CONTAINER_REGISTRY_NAME with the name of your Container Registry.

Using stateless services

Both the **TodoApi** and **TodoWeb** containerized services are defined as stateless services. The instance count for both services is equal to **–1**. This means that a container for each service is created on each Service Fabric cluster node.

For the **TodoApi** service, the ApplicationManifest.xml file defines **ServiceDnsName** as *todoapi.todoapp*. The **TodoAPI**'s DNS name and assigned port are provided in the **TodoAPIService_EndpointUri** environment variable (for example, **todoapi.todoapp:8081**). This environment variable is passed to the **TodoWeb** front-end service, which uses it to create the HTTP address of the **TodoApi** back-end service, as shown here:

```
namespace TodoWeb.Services
{
    /// <summary>
    /// TodoApiService class
    /// </summary>
    public class TodoApiService : ITodoApiService
    {
        private const string DefaultBaseAddress = "todoapi";
        ...

        private readonly IOptions<TodoApiServiceOptions> _options;
        private readonly ILogger<TodoApiService> _logger;
        private readonly HttpClient _httpClient;

        /// <summary>
        /// Initializes a new instance of the TodoApiService class.
        /// </summary>
        /// <param name="options"Service options.</param>
        /// <param name="logger">Logger.</param>
        public TodoApiService(IOptions<TodoApiServiceOptions> options,
                              ILogger<TodoApiService> logger)
        {
            _options = options;
```

```
            _logger = logger;

            var endpoint = string.IsNullOrWhiteSpace(_options?.Value?.EndpointUri) ?
                           DefaultBaseAddress :
                           _options.Value.EndpointUri;
            _httpClient = new HttpClient
            {
                BaseAddress = new Uri($"http://{endpoint}")
            };
            _httpClient.DefaultRequestHeaders.Accept.Clear();
            _httpClient.DefaultRequestHeaders.Add("ContentType", "application/json");
            _httpClient.DefaultRequestHeaders.Accept.Add(new MediaTypeWithQualityHeaderVal
ue("application/json"));

            _logger.LogInformation(LoggingEvents.Configuration, $"HttpClient.BaseAddress =
 {_httpClient.BaseAddress}");
        }
    ...
    }
}
```

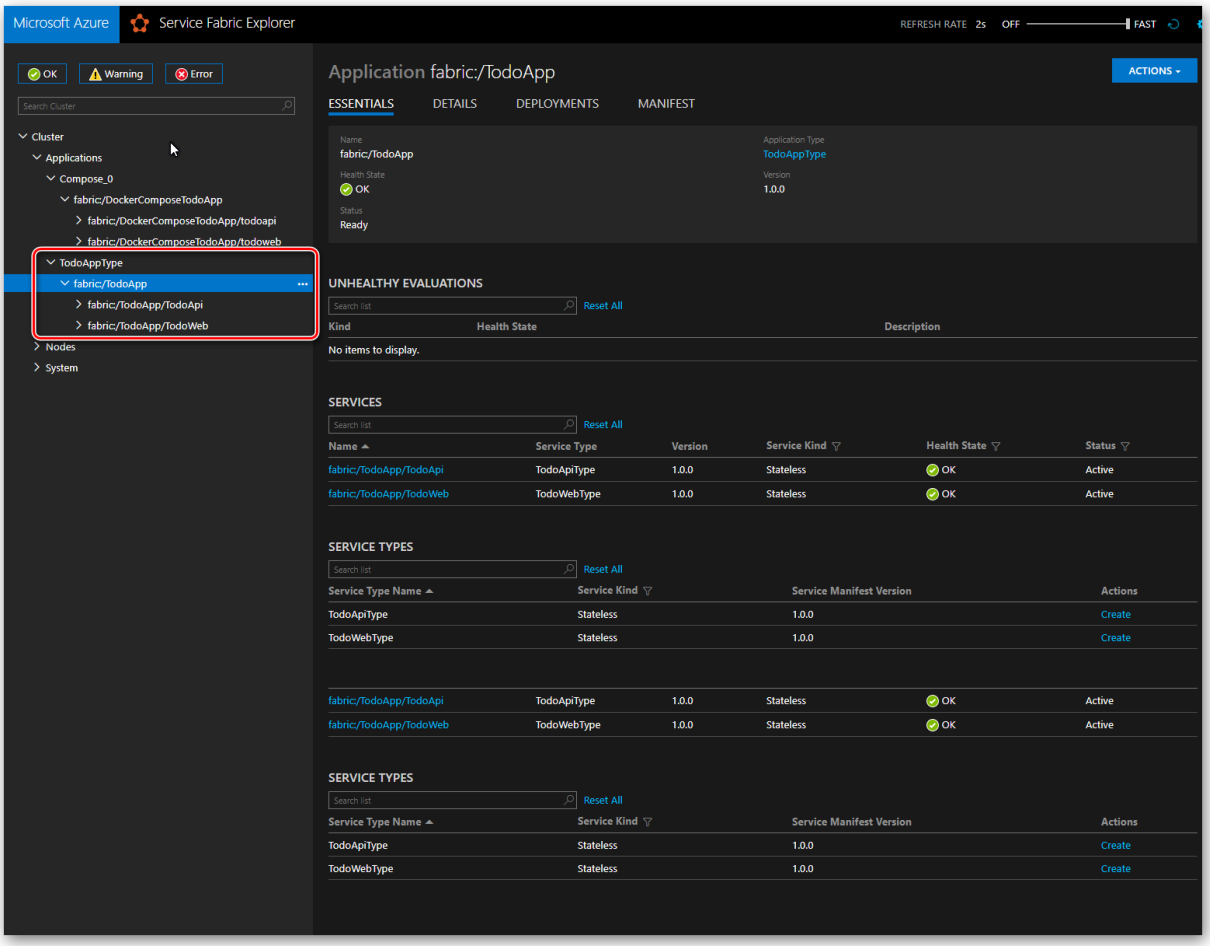

Figure 2. Service Fabric Explorer displays the cluster in the sample multi-container application.

Windows: manifests and parameters for deployment

The **TodoAppForWindowsFromDockerHub** project shows how to safely deploy a multi-container application to a Service Fabric cluster on Windows in a production environment. This project uses Key Vault for storing connection strings and other secret parameters, while non-sensitive configuration data, such as the name of the Service Bus queue, is stored in cleartext in the Cloud.xml file. A best practice is to always protect sensitive configuration data in the application manifest, service manifest, and application parameters file of a Service Fabric application and prevent unauthorized users from gaining access.

This project makes use of a single Key Vault repository for storing secrets. For larger projects, use multiple vaults for different environments such as development and test, quality assurance, performance testing, and production. Grant access to these resources only to a restricted set of authorized developers and operators.

This project requires you to store the following sensitive data in Key Vault:

- The endpoint URI of the Azure Cosmos DB database used by the back-end service to store data.
- The Azure Cosmos DB primary key.
- The name of the Azure Cosmos DB database.
- The name of the Azure Cosmos DB collection.
- The connection string of the Service Bus Messaging namespace used by the backend service for notifications.
- The connection string of the Storage Account used by ASP.NET Core Data Protection.
- The instrumentation key of the Application Insights resource used to monitor the multi-container application.

In addition, the following parameters are defined in cleartext in the Cloud.xml file:

- The name of the environment variable that contains the path of the .pem certificate file passed by Service Fabric when it starts the container.
- The name of the environment variable that contains the path of the .key certificate file passed by Service Fabric when it starts the container.
- The name of the Service Bus queue used by the back-end service to send notifications when an operation is performed on an Azure Cosmos DB document.
- The DNS name of the front-end and back-end services.
- The name of the container used by the two services to store data protection keys in the storage account.

Service manifests

In the service manifests below, you can see that both the front-end and back-end services read configuration data from environment variables, while the certificate used to authenticate against Key Vault is contained in data package called **Data**.

\TodoAppForWindowsFromDockerHub\ApplicationPackageRoot\TodoApiPkg\ServiceManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<ServiceManifest Name="TodoApiPkg"
                 Version="1.0.0"
                 xmlns="http://schemas.microsoft.com/2011/01/fabric"
                 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <ServiceTypes>
    <!-- This is the name of your ServiceType.
         The UseImplicitHost attribute indicates this is a guest service. -->
    <StatelessServiceType ServiceTypeName="TodoApiType" UseImplicitHost="true" />
  </ServiceTypes>

  <!-- Code package is your service executable. -->
  <CodePackage Name="Code" Version="1.0.0">
    <EntryPoint>
      <!-
- Follow this link for more information about deploying Windows containers to Service Fabr
ic: https://aka.ms/sfguestcontainers -->
      <ContainerHost>
        <ImageName>DOCKER_HUB_REPOSITORY/todoapi:v1</ImageName>
      </ContainerHost>
    </EntryPoint>
    <!-- Pass environment variables to your container: -->
    <EnvironmentVariables>
      <EnvironmentVariable Name="ASPNETCORE_ENVIRONMENT" Value=""/>
      <EnvironmentVariable Name="AzureKeyVault__Certificate__CertificateEnvironmentVariabl
e" Value=""/>
      <EnvironmentVariable Name="AzureKeyVault__Certificate__KeyEnvironmentVariable" Value
=""/>
      <EnvironmentVariable Name="AzureKeyVault__ClientId" Value=""/>
      <EnvironmentVariable Name="AzureKeyVault__Name" Value=""/>
      <EnvironmentVariable Name="NotificationService__ServiceBus__QueueName" Value=""/>
      <EnvironmentVariable Name="DataProtection__BlobStorage__ContainerName" Value=""/>
    </EnvironmentVariables>
  </CodePackage>

  <!-
- Config package is the contents of the Config directoy under PackageRoot that contains an

       independently-
updateable and versioned set of custom configuration settings for your service. -->
  <ConfigPackage Name="Config" Version="1.0.0" />

  <Resources>
    <Endpoints>
      <!-
- This endpoint is used by the communication listener to obtain the port on which to
           listen. Please note that if your service is partitioned, this port is shared wi
th
           replicas of different partitions that are placed in your code. -->
      <Endpoint Name="TodoApiEndpoint" Port="80" UriScheme="http" Protocol="http"/>
```

```
    </Endpoints>
  </Resources>
</ServiceManifest>
```

\TodoAppForWindowsFromDockerHub\ApplicationPackageRoot\TodoWebPkg\ServiceManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<ServiceManifest Name="TodoWebPkg"
                 Version="1.0.0"
                 xmlns="http://schemas.microsoft.com/2011/01/fabric"
                 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <ServiceTypes>
    <!-- This is the name of your ServiceType.
         The UseImplicitHost attribute indicates this is a guest service. -->
    <StatelessServiceType ServiceTypeName="TodoWebType" UseImplicitHost="true" />
  </ServiceTypes>

  <!-- Code package is your service executable. -->
  <CodePackage Name="Code" Version="1.0.0">
    <EntryPoint>
      <!-
- Follow this link for more information about deploying Windows containers to Service Fabr
ic: https://aka.ms/sfguestcontainers -->
      <ContainerHost>
        <ImageName>DOCKER_HUB_REPOSITORY/todoweb:v1</ImageName>
      </ContainerHost>
    </EntryPoint>
    <!-- Pass environment variables to your container: -->
    <EnvironmentVariables>
      <EnvironmentVariable Name="ASPNETCORE_ENVIRONMENT" Value=""/>
      <EnvironmentVariable Name="AzureKeyVault__Certificate__CertificateEnvironmentVariabl
e" Value=""/>
      <EnvironmentVariable Name="AzureKeyVault__Certificate__KeyEnvironmentVariable" Value
=""/>
      <EnvironmentVariable Name="AzureKeyVault__ClientId" Value=""/>
      <EnvironmentVariable Name="AzureKeyVault__Name" Value=""/>
      <EnvironmentVariable Name="TodoApiService__EndpointUri" Value=""/>
      <EnvironmentVariable Name="DataProtection__BlobStorage__ContainerName" Value=""/>
    </EnvironmentVariables>
  </CodePackage>

  <!-
- Config package is the contents of the Config directoy under PackageRoot that contains an

       independently-
updateable and versioned set of custom configuration settings for your service. -->
  <ConfigPackage Name="Config" Version="1.0.0" />

  <Resources>
    <Endpoints>
      <!-
- This endpoint is used by the communication listener to obtain the port on which to
```

```
          listen. Please note that if your service is partitioned, this port is shared wi
th
          replicas of different partitions that are placed in your code. -->
      <Endpoint Name="TodoWebEndpoint" Port="8080" UriScheme="http" Protocol="http"/>
    </Endpoints>
  </Resources>
</ServiceManifest>
```

\TodoAppForWindowsFromDockerHub\ApplicationPackageRoot\ApplicationManifest.xml

In the application manifest below, note that the certificate used by the front-end and back-end services to authenticate against Key Vault is defined in a **data** package.

```
<?xml version="1.0" encoding="utf-8"?>
<ApplicationManifest ApplicationTypeName="TodoAppType"
                     ApplicationTypeVersion="1.0.0"
                     xmlns="http://schemas.microsoft.com/2011/01/fabric"
                     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Parameters>
    <!-- Shared Parameters -->
    <Parameter Name="DockerHub_Username" DefaultValue="" />
    <Parameter Name="DockerHub_Password" DefaultValue="" />
    <Parameter Name="ASPNETCORE_ENVIRONMENT" DefaultValue=""/>
    <Parameter Name="Certificate_Thumbprint" DefaultValue="" />
    <Parameter Name="AzureKeyVault__ClientId" DefaultValue=""/>
    <Parameter Name="AzureKeyVault__Name" DefaultValue=""/>
    <!-- TodoWeb Parameters -->
    <Parameter Name="TodoWeb_InstanceCount" DefaultValue="-1" />
    <Parameter Name="TodoWeb_AzureKeyVault__Certificate__CertificateEnvironmentVariable" D
efaultValue=""/>
    <Parameter Name="TodoWeb_AzureKeyVault__Certificate__KeyEnvironmentVariable" DefaultVa
lue=""/>
    <Parameter Name="TodoWeb_TodoApiService__EndpointUri" DefaultValue=""/>
    <Parameter Name="TodoWeb_DataProtection__BlobStorage__ContainerName" DefaultValue=""/>
    <!-- TodoApi Parameters -->
    <Parameter Name="TodoApi_InstanceCount" DefaultValue="-1" />
    <Parameter Name="TodoApi_AzureKeyVault__Certificate__CertificateEnvironmentVariable" D
efaultValue=""/>
    <Parameter Name="TodoApi_AzureKeyVault__Certificate__KeyEnvironmentVariable" DefaultVa
lue=""/>
    <Parameter Name="TodoApi_NotificationService__ServiceBus__QueueName" DefaultValue=""/>
    <Parameter Name="TodoApi_DataProtection__BlobStorage__ContainerName" DefaultValue=""/>
  </Parameters>
  <!-
- Import the ServiceManifest from the ServicePackage. The ServiceManifestName and ServiceM
anifestVersion
       should match the Name and Version attributes of the ServiceManifest element defined
 in the
       ServiceManifest.xml file. -->
  <ServiceManifestImport>
    <ServiceManifestRef ServiceManifestName="TodoWebPkg" ServiceManifestVersion="1.0.0" />
    <ConfigOverrides />
    <EnvironmentOverrides CodePackageRef="Code">
```

```
      <EnvironmentVariable Name="ASPNETCORE_ENVIRONMENT" Value="[ASPNETCORE_ENVIRONMENT]"/
>
      <EnvironmentVariable Name="AzureKeyVault__Certificate__CertificateEnvironmentVariabl
e" Value="[TodoWeb_AzureKeyVault__Certificate__CertificateEnvironmentVariable]"/>
      <EnvironmentVariable Name="AzureKeyVault__Certificate__KeyEnvironmentVariable" Value
="[TodoWeb_AzureKeyVault__Certificate__KeyEnvironmentVariable]"/>
      <EnvironmentVariable Name="AzureKeyVault__ClientId" Value="[AzureKeyVault__ClientId]
"/>
      <EnvironmentVariable Name="AzureKeyVault__Name" Value="[AzureKeyVault__Name]"/>
      <EnvironmentVariable Name="TodoApiService__EndpointUri" Value="[TodoWeb_TodoApiServi
ce__EndpointUri]"/>
      <EnvironmentVariable Name="DataProtection__BlobStorage__ContainerName" Value="[TodoW
eb_DataProtection__BlobStorage__ContainerName]"/>
    </EnvironmentOverrides>
    <Policies>
      <ContainerHostPolicies CodePackageRef="Code">
        <RepositoryCredentials AccountName="[DockerHub_Username]" Password="[DockerHub_Pas
sword]" PasswordEncrypted="false"/>
        <PortBinding ContainerPort="80" EndpointRef="TodoWebEndpoint" />
        <CertificateRef Name="TodoListCert" X509FindValue="[Certificate_Thumbprint]"/>
      </ContainerHostPolicies>
    </Policies>
  </ServiceManifestImport>
  <ServiceManifestImport>
    <ServiceManifestRef ServiceManifestName="TodoApiPkg" ServiceManifestVersion="1.0.0" />
    <ConfigOverrides />
    <EnvironmentOverrides CodePackageRef="Code">
      <EnvironmentVariable Name="ASPNETCORE_ENVIRONMENT" Value="[ASPNETCORE_ENVIRONMENT]"/
>
      <EnvironmentVariable Name="AzureKeyVault__Certificate__CertificateEnvironmentVariabl
e" Value="[TodoApi_AzureKeyVault__Certificate__CertificateEnvironmentVariable]"/>
      <EnvironmentVariable Name="AzureKeyVault__Certificate__KeyEnvironmentVariable" Value
="[TodoApi_AzureKeyVault__Certificate__KeyEnvironmentVariable]"/>
      <EnvironmentVariable Name="AzureKeyVault__ClientId" Value="[AzureKeyVault__ClientId]
"/>
      <EnvironmentVariable Name="AzureKeyVault__Name" Value="[AzureKeyVault__Name]"/>
      <EnvironmentVariable Name="NotificationService__ServiceBus__QueueName" Value="[TodoA
pi_NotificationService__ServiceBus__QueueName]"/>
      <EnvironmentVariable Name="DataProtection__BlobStorage__ContainerName" Value="[TodoA
pi_DataProtection__BlobStorage__ContainerName]"/>
    </EnvironmentOverrides>
    <Policies>
      <ContainerHostPolicies CodePackageRef="Code">
        <RepositoryCredentials AccountName="[DockerHub_Username]" Password="[DockerHub_Pas
sword]" PasswordEncrypted="false"/>
        <PortBinding ContainerPort="80" EndpointRef="TodoApiEndpoint" />
        <CertificateRef Name="TodoListCert" X509FindValue="[Certificate_Thumbprint]"/>
      </ContainerHostPolicies>
    </Policies>
  </ServiceManifestImport>
  <DefaultServices>
    <!-- The section below creates instances of service types, when an instance of this
         application type is created. You can also create one or more instances of service
 type using the
```

```
         ServiceFabric PowerShell module.

         The attribute ServiceTypeName below must match the name defined in the imported S
erviceManifest.xml file. -->
    <Service Name="TodoWeb" ServicePackageActivationMode="ExclusiveProcess" ServiceDnsName
="todoweb.todoapp">
      <StatelessService ServiceTypeName="TodoWebType" InstanceCount="[TodoWeb_InstanceCoun
t]">
        <SingletonPartition />
      </StatelessService>
    </Service>
    <Service Name="TodoApi" ServicePackageActivationMode="ExclusiveProcess" ServiceDnsName
="todoapi.todoapp">
      <StatelessService ServiceTypeName="TodoApiType" InstanceCount="[TodoApi_InstanceCoun
t]">
        <SingletonPartition />
      </StatelessService>
    </Service>
  </DefaultServices>
</ApplicationManifest>
```

\TodoAppForWindowsFromDockerHub\ApplicationParameters\Cloud.xml

```
<?xml version="1.0" encoding="utf-8"?>
<Application Name="fabric:/TodoApp" xmlns="http://schemas.microsoft.com/2011/01/fabric" xm
lns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
  <Parameters>
    <!-- Shared Parameters -->
    <Parameter Name="DockerHub_Username" Value="DOCKER_HUB_USERNAME" />
    <Parameter Name="DockerHub_Password" Value="DOCKER_HUB_PASSWORD" />
    <Parameter Name="ASPNETCORE_ENVIRONMENT" Value="Development"/>
    <Parameter Name="Certificate_Thumbprint" Value="CERTIFICATE_THUMBPRINT" />
    <Parameter Name="AzureKeyVault__ClientId" Value="AZURE_AD_APPLICATION_ID"/>
    <Parameter Name="AzureKeyVault__Name" Value="AZURE_KEY_VAULT_NAME"/>
    <!-- TodoWeb Parameters -->
    <Parameter Name="TodoWeb_InstanceCount" Value="-1" />
    <Parameter Name="TodoWeb_AzureKeyVault__Certificate__CertificateEnvironmentVariable" V
alue="Certificates_TodoWebPkg_Code_TodoListCert_PEM"/>
    <Parameter Name="TodoWeb_AzureKeyVault__Certificate__KeyEnvironmentVariable" Value="Ce
rtificates_TodoWebPkg_Code_TodoListCert_PrivateKey"/>
    <Parameter Name="TodoWeb_TodoApiService__EndpointUri" Value="todoapi.todoapp"/>
    <Parameter Name="TodoWeb_DataProtection__BlobStorage__ContainerName" Value="todoweb"/>
    <!-- TodoApi Parameters -->
    <Parameter Name="TodoApi_InstanceCount" Value="-1" />
    <Parameter Name="TodoWeb_AzureKeyVault__Certificate__CertificateEnvironmentVariable" V
alue="Certificates_TodoWebPkg_Code_TodoListCert_PEM"/>
    <Parameter Name="TodoWeb_AzureKeyVault__Certificate__KeyEnvironmentVariable" Value="Ce
rtificates_TodoWebPkg_Code_TodoListCert_PrivateKey"/>
    <Parameter Name="TodoApi_NotificationService__ServiceBus__QueueName" Value="todoapi"/>
    <Parameter Name="TodoApi_DataProtection__BlobStorage__ContainerName" Value="todoapi"/>
  </Parameters>
</Application>
```

Configure the manifests for Windows cluster deployment

To deploy the multi-container application to a Service Fabric cluster, you must build OS-specific images as mentioned earlier. You cannot use a Docker image for Linux to deploy a Window container to a Service Fabric cluster on Windows in Azure. Likewise, you cannot use a Docker image for Windows to deploy a Linux container to a Service Fabric cluster on Linux in Azure.

Before deploying the application to your cluster:

1. Open Cloud.xml and make the following changes:
    a. Replace DOCKER_HUB_USERNAME with the username of your Docker Hub repository.
    b. Replace DOCKER_HUB_PASSWORD with the password of your Docker Hub repository.
    c. Replace AZURE_AD_APPLICATION_ID with the application ID of the Azure AD application used by the application to authenticate against Key Vault.
    d. Replace AZURE_KEY_VAULT_NAME with the name of the key vault used to store sensitive configuration data.
2. Open \TodoAppForLinuxFromACR\TodoApiServiceManifest.xml. Replace DOCKER_HUB_REPOSITORY with the name of your Docker Hub repository.
3. Open \TodoAppForLinuxFromACR\TodoWebServiceManifest.xml. Replace DOCKER_HUB_REPOSITORY with the name of your Docker Hub repository.

Use Docker Compose for a Service Fabric deployment

In Service Fabric, you can deploy a container-based application using application and service manifests as described earlier or using Docker Compose. To make it easy for developers familiar with Docker to orchestrate existing container applications on Service Fabric, Docker Compose support was added. Docker uses the docker-compose.yml file to define multi-container applications.

At the time of writing, Docker Compose deployment in Service Fabric does not support all the features provided by the manifest-based deployment process, such as the ability to pass a certificate to a Docker container. However, you can use an Azure Resource Manager template to configure the following settings in a Service Fabric cluster running Linux or Windows on Azure:

- Time interval before container is force terminated.
- Runtime to remove unused container images.
- Container image download time.

By comparison, you can set the container retention policy in the application manifest.

This project includes both batch and PowerShell scripts that you can use to deploy the multi-container application to a Service Fabric cluster on Azure with DNS Service. The cluster can pull the Docker images from a Container Registry or a Docker Hub repository.

For more information, see Docker Compose deployment support in Azure Service Fabric and Azure Service Fabric CLI.

ⓘ **NOTE:** Service Fabric supports version 3 and later of docker-compose.yml files.

Pull images from Container Registry

When the cluster pulls the images from Container Registry, use one of the following to deploy the multi-container application:

- **servicefabric-create-deployment-from-azure-container-registry.cmd** is a command file that uses the Service Fabric CLI to deploy the **DockerComposeTodoApp** multi-container application.
- **servicefabric-create-deployment-from-azure-container-registry.ps1** is a PowerShell script to deploy the **DockerComposeTodoApp** multi-container application.

Both the command file and the script deploy the application to the cluster using Docker Compose. The cluster pulls the Docker images from Container Registry using the definition for the **TodoWeb** and **TodoApi** services contained in the **servicefabric-docker-compose-from-azure-container-registry.yml** file shown below.

servicefabric-docker-compose-from-azure-container-registry.yml

```
version: '3'

services:
  todoapi:
    image:AZURE_CONTAINER_REGISTRY_NAME.azurecr.io/todoapi:latest
    deploy:
      mode: replicated
      replicas: 5
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - RepositoryService__CosmosDb__EndpointUri=COSMOS_DB_ENDPOINT_URI
      - RepositoryService__CosmosDb__PrimaryKey=COSMOS_DB_PRIMARY_KEY
      - RepositoryService__CosmosDb__DatabaseName=TodoApiDb
      - RepositoryService__CosmosDb__CollectionName=TodoApiCollection
      - NotificationService__ServiceBus__ConnectionString=SERVICE_BUS_CONNECTION_STRING
      - NotificationService__ServiceBus__QueueName=todoapi
      - DataProtection__BlobStorage__ConnectionString=STORAGE_ACCOUNT_CONNECTION_STRING
      - DataProtection__BlobStorage__ContainerName=todoapi
      - ApplicationInsights__InstrumentationKey=APPLICATION_INSIGHTS_INSTRUMENTATION_KEY
    ports:
      - "8081:80"

  todoweb:
    image:AZURE_CONTAINER_REGISTRY_NAME.azurecr.io/todoweb:latest
```

```
    deploy:
      mode: replicated
      replicas: 5
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - TodoApiService__EndpointUri=todoapi:8081
      - DataProtection__BlobStorage__ConnectionString=STORAGE_ACCOUNT_CONNECTION_STRING
      - DataProtection__BlobStorage__ContainerName=todoweb
      - ApplicationInsights__InstrumentationKey=APPLICATION_INSIGHTS_INSTRUMENTATION_KEY
    ports:
      - "8082:80"
```

servicefabric-create-deployment-from-azure-container-registry.cmd

```
REM Create a Service Fabric Compose deployment from a docker-compose.yml file
sfctl compose create --name DockerComposeTodoApp --file-path servicefabric-docker-compose-from-azure-container-registry.yml --user AZURE_CONTAINER_REGISTRY_USERNAME --encrypted-pass AZURE_CONTAINER_REGISTRY_PASSWORD
```

servicefabric-create-deployment-from-azure-container-registry.ps1

```
# Copy the package and register application type of version 1.0
$connectionEndpoint =
"SERVICE_FABRIC_NAME.SERVICE_FABRIC_LOCATION.cloudapp.azure.com:19000"
$dockerComposeFile = $PSScriptRoot + '\servicefabric-docker-compose-from-azure-container-registry.yml'

Connect-ServiceFabricCluster -ConnectionEndpoint $connectionEndpoint

New-ServiceFabricComposeDeployment -DeploymentName DockerComposeTodoApp -Compose $dockerComposeFile -RegistryUserName AZURE_CONTAINER_REGISTRY_USERNAME -RegistryPassword AZURE_CONTAINER_REGISTRY_PASSWORD
```

Configure the Container Registry deployment scripts

Before deploying the application to your cluster, do the following:

1. Open the command file or PowerShell script used to deploy the application and replace the following placeholders:

    a. Replace AZURE_CONTAINER_REGISTRY_USERNAME with the username of your Container Registry. (The username is case-sensitive.)

    b. Replace AZURE_CONTAINER_REGISTRY_PASSWORD with the password of your Container Registry.

    c. Replace SERVICE_FABRIC_NAME in the PowerShell script with the name of your Service Fabric Linux cluster.

    d. Replace SERVICE_FABRIC_LOCATION in the PowerShell script with the location of your Service Fabric Linux cluster.

2. Open the YAML file and replace the following placeholders:

    a. Replace AZURE_CONTAINER_REGISTRY_NAME with the name of your Container Registry.

b. Replace COSMOS_DB_ENDPOINT_URI with the endpoint URI of your Azure Cosmos DB.

c. Replace COSMOS_DB_PRIMARY_KEY with the primary key of your Azure Cosmos DB.

d. Replace SERVICE_BUS_CONNECTION_STRING with the connection string of your Service Bus Messaging namespace.

e. Replace STORAGE_ACCOUNT_CONNECTION_STRING with the connection string of the storage account used by ASP.NET Core Data Protection.

f. Replace APPLICATION_INSIGHTS_INSTRUMENTATION_KEY with the instrumentation key of the Application Insights resource used to monitor the multi-container application.

3. To retrieve the username and password of your Container Registry, run the following command:

```
az acr credential show -n [AZURE_CONTAINER_REGISTRY]
```

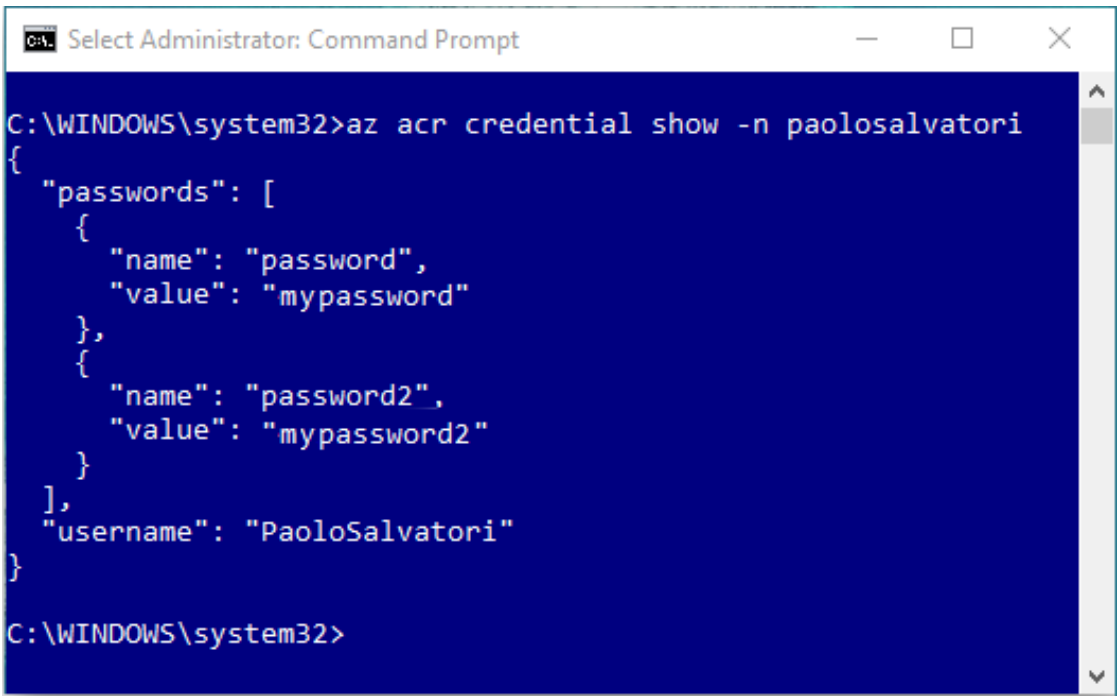

Figure 3. Obtaining the username and password of your Container Registry.

Pull images from Docker Hub

When the cluster pulls the images from a Docker Hub repository, use one of the following to deploy the multi-container application:

- **servicefabric-create-deployment-from-docker-hub.cmd** is a command file that uses the Azure Service Fabric CLI.
- **servicefabric-create-deployment-from-docker-hub.ps1** is a PowerShell script.

Both the command file and the script deploy the **DockerComposeTodoApp** multi-container application to a Service Fabric cluster using Docker Compose. The cluster pulls the Docker images from a Docker Hub

repository using the definition for the **TodoWeb** and **TodoApi** services contained in the servicefabric-docker-compose-from-docker-hub.yml file shown below.

servicefabric-docker-compose-from-docker-hub.yml

```
version: '3'

services:
  todoapi:
    image: DOCKER_HUB_REPOSITORY/todoapi:latest
    deploy:
      mode: replicated
      replicas: 5
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - RepositoryService__CosmosDb__EndpointUri=COSMOS_DB_ENDPOINT_URI
      - RepositoryService__CosmosDb__PrimaryKey=COSMOS_DB_PRIMARY_KEY
      - RepositoryService__CosmosDb__DatabaseName=TodoApiDb
      - RepositoryService__CosmosDb__CollectionName=TodoApiCollection
      - NotificationService__ServiceBus__ConnectionString=SERVICE_BUS_CONNECTION_STRING
      - NotificationService__ServiceBus__QueueName=todoapi
      - DataProtection__BlobStorage__ConnectionString=STORAGE_ACCOUNT_CONNECTION_STRING
      - DataProtection__BlobStorage__ContainerName=todoapi
      - ApplicationInsights__InstrumentationKey=APPLICATION_INSIGHTS_INSTRUMENTATION_KEY
    ports:
      - "8081:80"

  todoweb:
    image: DOCKER_HUB_REPOSITORY/todoweb:latest
    deploy:
      mode: replicated
      replicas: 5
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - TodoApiService__EndpointUri=todoapi:8081
      - DataProtection__BlobStorage__ConnectionString=STORAGE_ACCOUNT_CONNECTION_STRING
      - DataProtection__BlobStorage__ContainerName=todoweb
      - ApplicationInsights__InstrumentationKey=APPLICATION_INSIGHTS_INSTRUMENTATION_KEY
    ports:
      - "8082:80"
```

servicefabric-create-deployment-from-docker-hub.cmd

```
REM Create a Service Fabric Compose deployment from a docker-compose.yml file
sfctl compose create --name DockerComposeTodoApp --file-path servicefabric-docker-compose-
from-docker-hub.yml --user DOCKER_HUB_USERNAME --encrypted-pass DOCKER_HUB_PASSWORD
```

servicefabric-create-deployment-from-docker-hub.ps1

```
# Copy the package and register application type of version 1.0
$connectionEndpoint =
"SERVICE_FABRIC_NAME.SERVICE_FABRIC_LOCATION.cloudapp.azure.com:19000"
$dockerComposeFile = $PSScriptRoot + '\servicefabric-docker-compose-from-docker-hub.yml'
```

```
Connect-ServiceFabricCluster -ConnectionEndpoint $connectionEndpoint

New-ServiceFabricComposeDeployment -DeploymentName DockerComposeTodoApp -Compose
$dockerComposeFile -RegistryUserName DOCKER_HUB_USERNAME -RegistryPassword
DOCKER_HUB_PASSWORD
```

Configure the Docker Hub deployment scripts

Before deploying the application to your cluster, do the following:

1. Open the command file or PowerShell script used to deploy the application and make the following changes:
    a. Replace DOCKER-HUB-USERNAME with your Docker Hub username.
    b. Replace DOCKER_HUB_PASSWORD with your Docker Hub password.
    c. Replace SERVICE_FABRIC_NAME in the PowerShell script with the name of your Service Fabric Linux cluster.
    d. Replace SERVICE_FABRIC_LOCATION in the PowerShell script with the location of your Service Fabric Linux cluster.
2. Open the YAML file and make the following changes:
    a. Replace DOCKER_HUB_REPOSITORY with the name of your Docker Hub repository.
    b. Replace COSMOS_DB_ENDPOINT_URI with the endpoint URI of your Azure Cosmos DB.
    c. Replace COSMOS_DB_PRIMARY_KEY with the primary key of your Azure Cosmos DB.
    d. Replace SERVICE_BUS_CONNECTION_STRING with the connection string of your Service Bus Messaging namespace.
    e. Replace STORAGE_ACCOUNT_CONNECTION_STRING with the connection string of the storage account used by ASP.NET Core Data Protection.
    f. Replace APPLICATION_INSIGHTS_INSTRUMENTATION_KEY with the instrumentation key of the Application Insights resource used to monitor the multi-container application.

The front-end website looks like this:

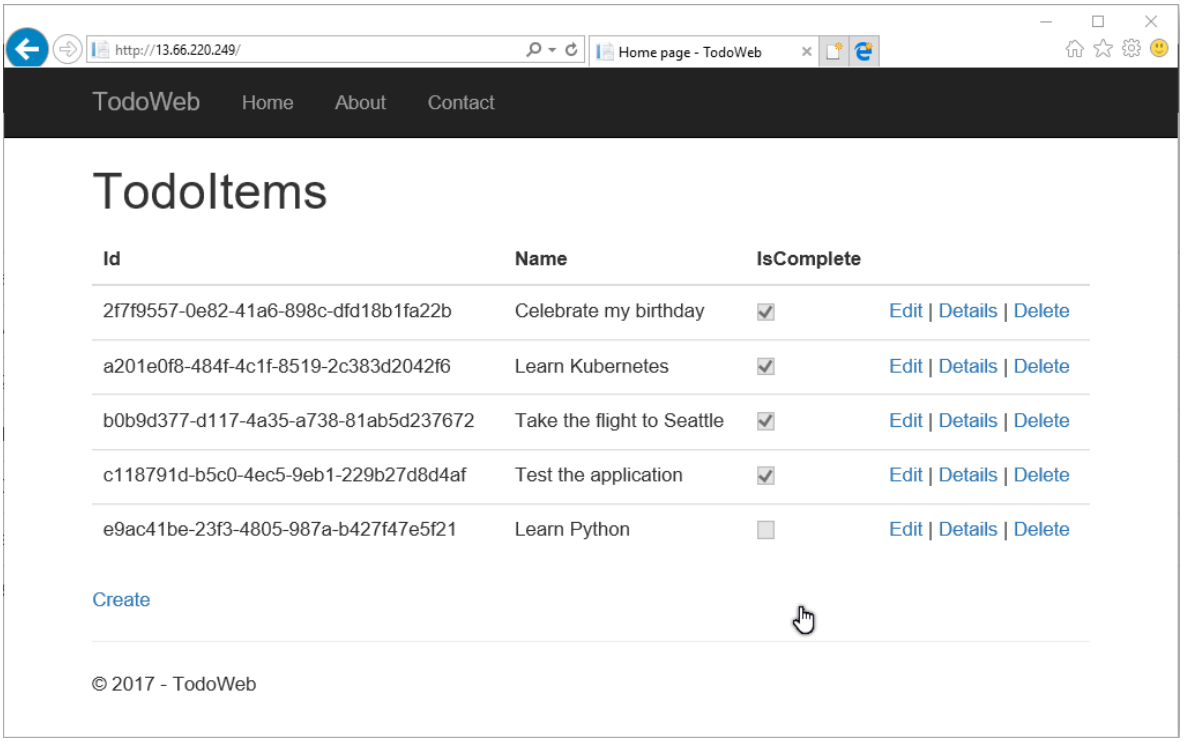

Figure 4. Web interface for TodoWeb service.

The following figure shows the multi-container application in Service Fabric Explorer.

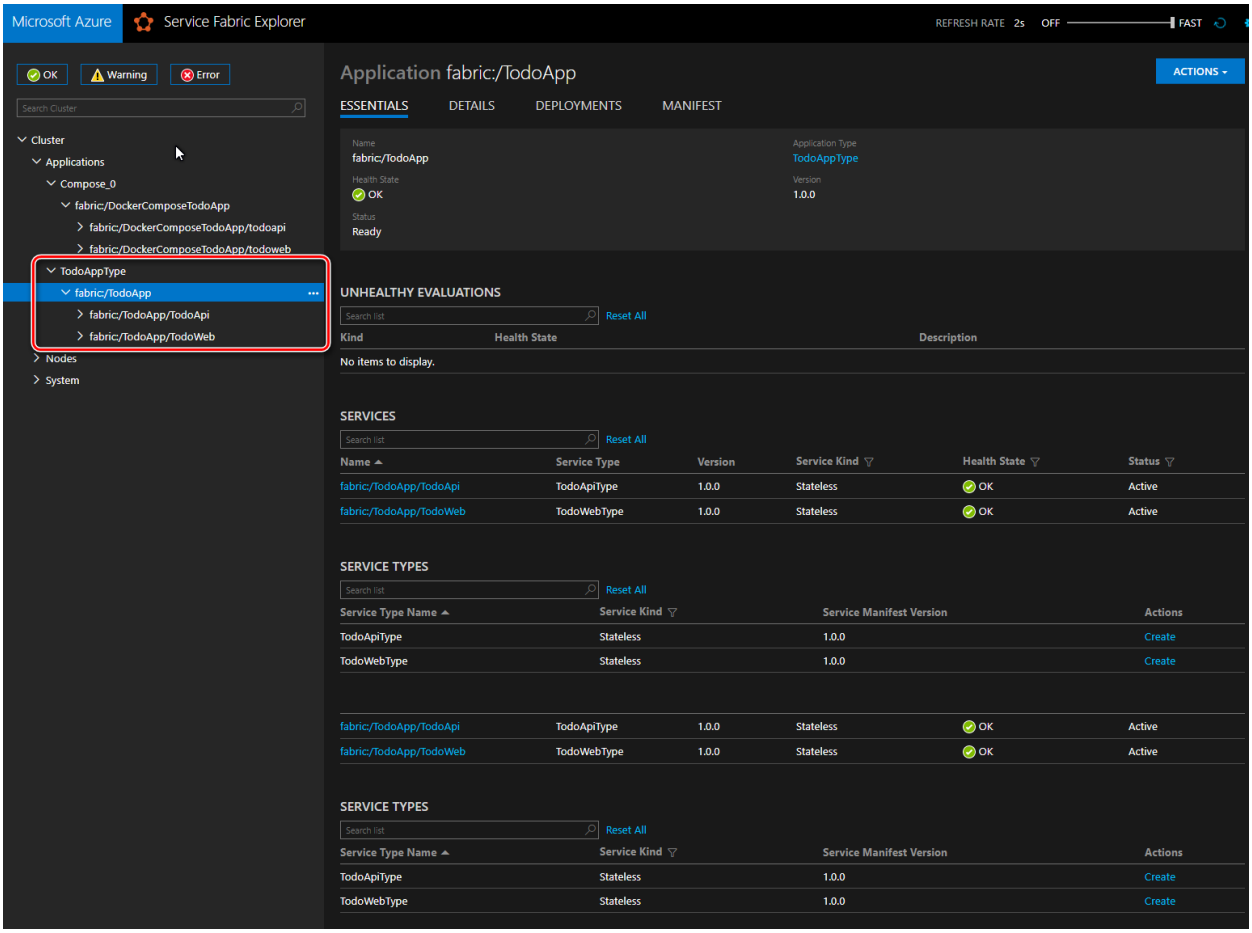

Figure 5. Service Fabric Explorer shows the state of the cluster and services for the deployed application.

Azure services used by this project

Service Fabric is a distributed systems platform that makes it easy to package, deploy, and manage scalable and reliable microservices and containers. Service Fabric also addresses the significant challenges in developing and managing cloud-native applications. Developers and administrators can avoid complex infrastructure problems and focus on implementing mission-critical, demanding workloads that are scalable, reliable, and manageable. Service Fabric represents the next-generation platform for building and managing these enterprise-class, tier-1, cloud-scale applications running in containers.

Application Insights is an extensible Application Performance Management (APM) service for monitoring live web applications. It detects performance anomalies and includes powerful analytics tools to help you diagnose issues and to understand what users do with your apps. It works for apps on a wide variety of platforms including .NET, Node.js, and J2EE, hosted on premises or in the cloud.

Service Bus Messaging is a reliable message delivery service for brokered or third-party communications that connect applications through the cloud. Service Bus messaging with queues, topics, and subscriptions can be thought of as asynchronous, or temporally decoupled. Producers (senders) and consumers (receivers) do not have to be online at the same time. The messaging infrastructure reliably stores messages in a "broker" (for example, a queue) until the consuming party is ready to receive them.

Azure Cosmos DB is a globally distributed, multi-model database that can elastically and independently scale throughput and storage across any number of Azure's geographic regions. Azure Cosmos DB supports multiple data models and popular APIs for accessing and querying data. The sample in this solution uses the Azure Cosmos DB SQL API, which provides a schema-less JSON database engine with SQL querying capabilities.

Azure Key Vault helps safeguard cryptographic keys and secrets used by cloud applications and services. By using Key Vault, you can encrypt keys and secrets (such as authentication keys, storage account keys, data encryption keys, .PFX files, and passwords) by using keys that are protected by hardware security modules (HSMs). For added assurance, you can import or generate keys in HSMs. If you choose to do this, Microsoft processes your keys in FIPS 140-2 Level 2 validated HSMs (hardware and firmware). Key Vault streamlines the key management process and enables you to maintain control of keys that access and encrypt your data. Developers can create keys for development and testing in minutes, and then seamlessly migrate them to production keys. Security administrators can grant (and revoke) permission to keys, as needed.

Azure Container Registry is a managed Docker registry service based on the open-source Docker Registry 2.0. You can create and maintain Azure container registries to store and manage your private Docker container images. Container registries in Azure work with your existing container development and deployment pipelines and draw on the body of Docker community expertise.

Azure Domain Name System (DNS) is responsible for translating (or resolving) a website or service name to its IP address. Azure DNS is a hosting service for DNS domains, providing name resolution using the Azure infrastructure. By hosting your domains in Azure, you can manage your DNS records using the same credentials, APIs, tools, and billing as your other Azure services.

Learn more

The Service Fabric container documentation provides details on the container features and scenarios.

The following useful links contain more in-depth information:

- Create a Linux Container Application
- Deploy an Azure Service Fabric Linux container application on Azure

Docker

- Introduction to Containers and Docker
- What is Docker?
- Building Docker Images for .NET Core Applications
- Docker File Reference
- Docker Compose
- Securing .NET Microservices and Web Applications

ASP.NET

- Configuration in ASP.NET Core
- Introduction to Logging in ASP.NET Core
- Introduction to Error Handling in ASP.NET Core
- Securing .NET Microservices and Web Applications

Service Fabric

- Create a .NET Service Fabric application in Azure
- Deploy a Service Fabric Windows container application on Azure
- Deploy an Azure Service Fabric Linux container application on Azure

Questions, comments and issues on Service Fabric:

- Tagged questions on Stack Overflow
- Service Fabric forum
- Azure Service Fabric Issues on GitHub
- Service Fabric Slack

Service Fabric announcements, courses, samples, and labs

- Azure Service Fabric Team Blog
- Building Microservices Applications on Azure Service Fabric
- Azure Service Fabric Code Samples
- Azure Service Fabric Party Clusters: try Service Fabric on Azure for free
- Introduction to Service Fabric (Lab)
- Focus on... Azure Service Fabric!