In this article I will describe how to use Azure DevOps to:

- build a docker image of your application,
- push this image to Azure Container Register,
- release your kubernetes deployment to Azure Kubernetes Service.

If you don't have any basic knowledge about docker, kubernetes, and Azure CLI please check out my previous article Deploying .NET Core Application to Azure Kubernetes Cluster.

The Benefits of Automating Continuous Integration and Delivery Processes

Automating CI/CD processes allows you to:

- save a lot of time
- eliminate bugs that happen when you do the same job over and over again
- deliver your product much faster

Azure DevOps can help you with that.

Azure DevOps

Azure DevOps can automate your Continuous Integration and Delivery processes. It can get access to your git repository (Azure Repos Git, GitHub, and other git repositories). It can automatically react to your activity in your repository:

- run tests when you create a pull request
- build a docker image when you merge your pull request to a selected branch and push it to Azure Container Register (ACR)
- when everything is ok, it can apply changes to your Azure Kubernetes Service (AKS)

You can use docker images from your ACR to create as many release configurations as you need. For example, one for dev, test, stage and production environment and decide when you want to release them.

If you don't have your Azure Container Register and Azure Kubernetes Service, you can use necessary Azure CLI commands from my previous article:

Azure setup

```
#Create temp variables:
$projectName="shkube"
$argName=$projectName+"RG"
$acrName=$projectName+"ACR"
$aksName=$projectName+"AKS"
$region="northeurope"
```

We are going to use them to create ACR and AKS in Azure Resource Group (ARG).

Login to azure:

```
az login
```

And lets create everyting you will need later:

```
# Create resource group
az group create -n $argName -l $region

# Create azure container register
az acr create -n $acrName -g $argName --sku standard

# Create azure kubernetes service
az aks create -n $aksName -g $argName --generate-ssh-keys --node-count 1  --node-vm-size Standa
rd_B2s --enable-addons monitoring

# Get AKS Client Id and AKS Id
$CLIENT_ID=$(az aks show -g $argName -n $aksName --query "servicePrincipalProfile.clientId" --o
utput tsv)
$ACR_ID=az acr show --name $acrName --resource-group $argName --query "id" --output tsv

# Give AKS access to ACR
az role assignment create --assignee $CLIENT_ID --role acrpull --scope $ACR_ID

# Get credential to your AKS
az aks get-credentials -g $argName -n $aksName
```

Now we can begin to work with your CI/CD.

Prerequirements

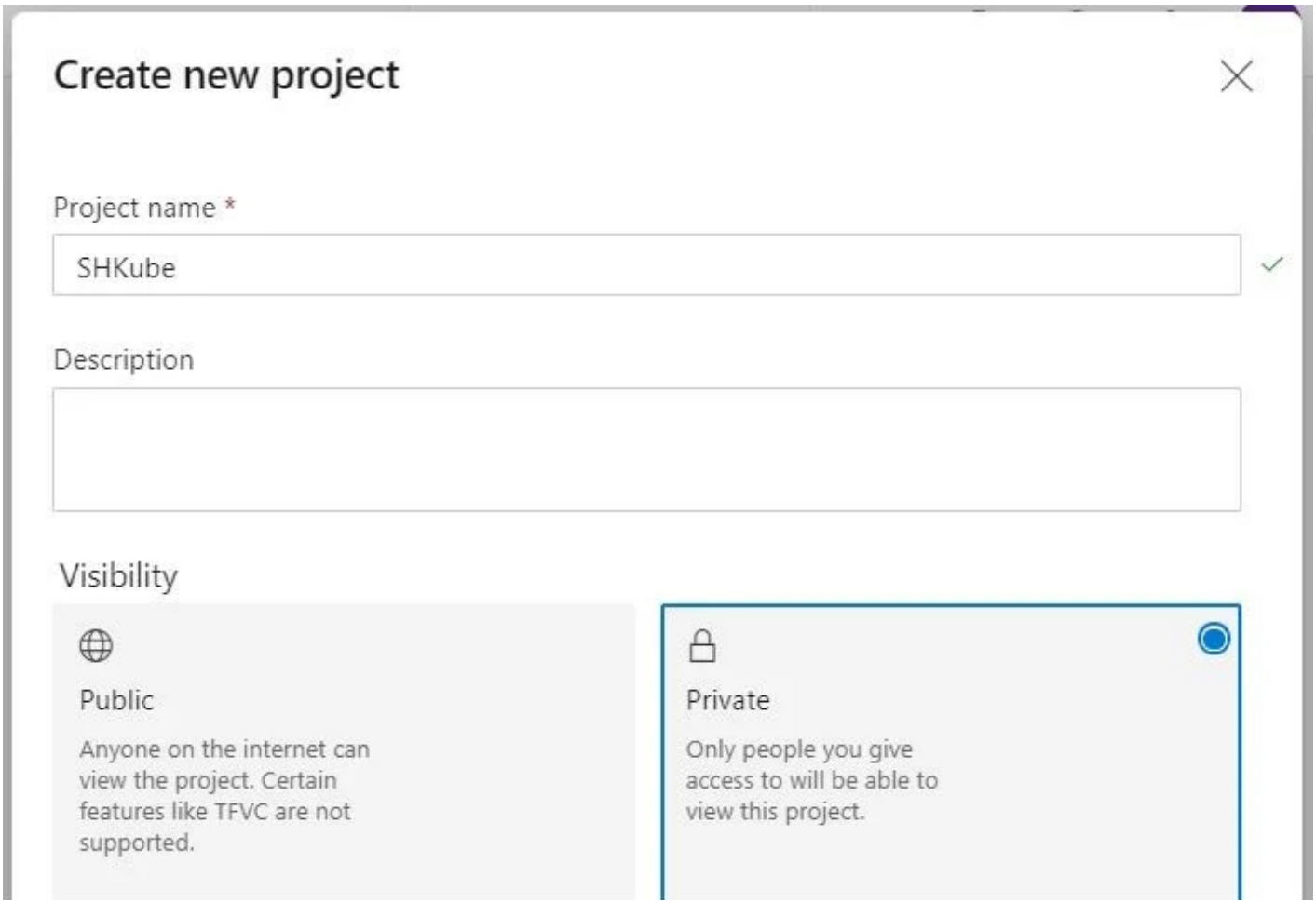
If you are reading this article, then you probably already have your application up and running inside Docker Container or even Kubernetes. If not, then you need three additional files to do that:

- Dockerfile
- docker-compose.yml
- deployment.yml

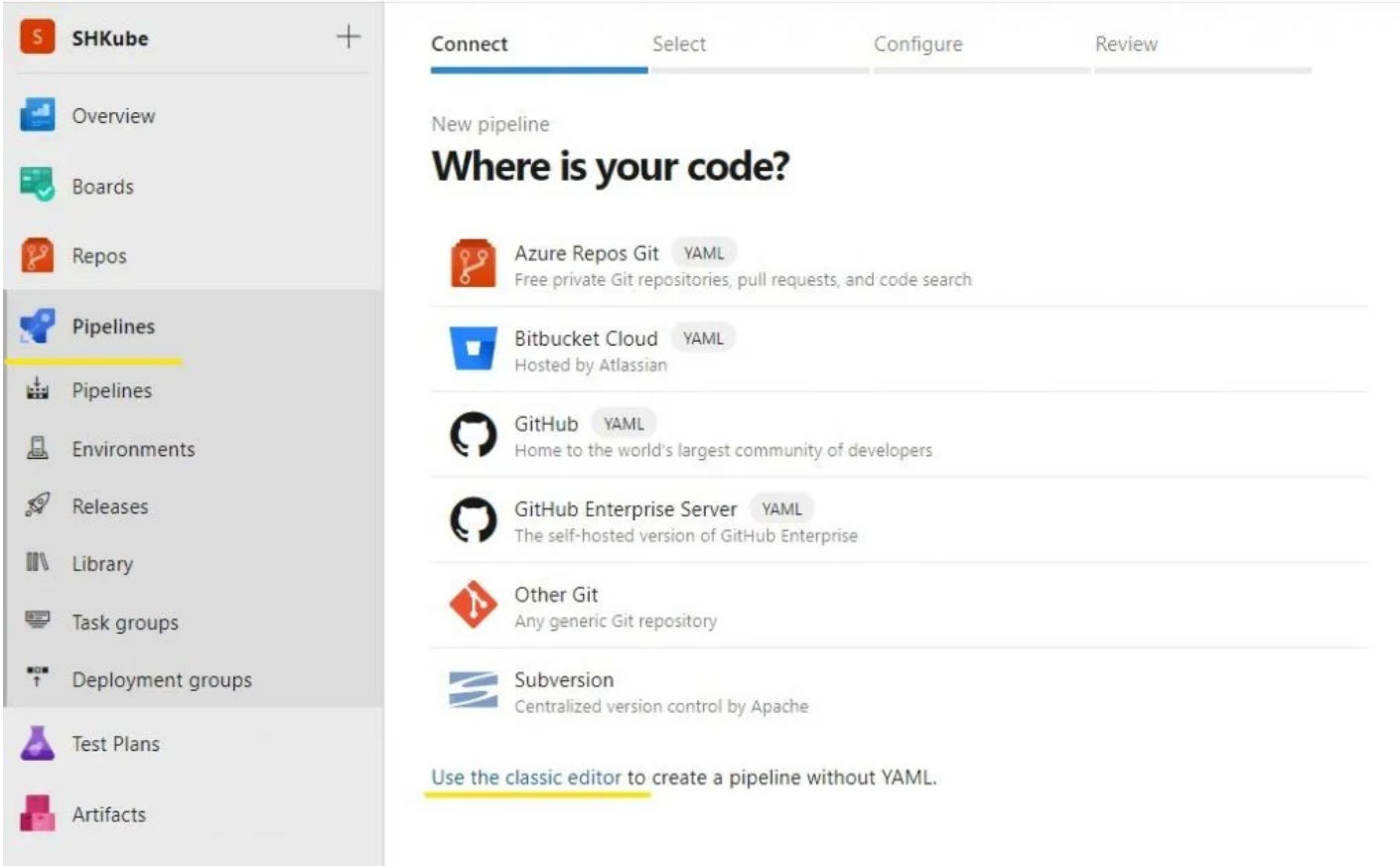
If you don’t have them you can use the ones from this project.

Azure DevOps

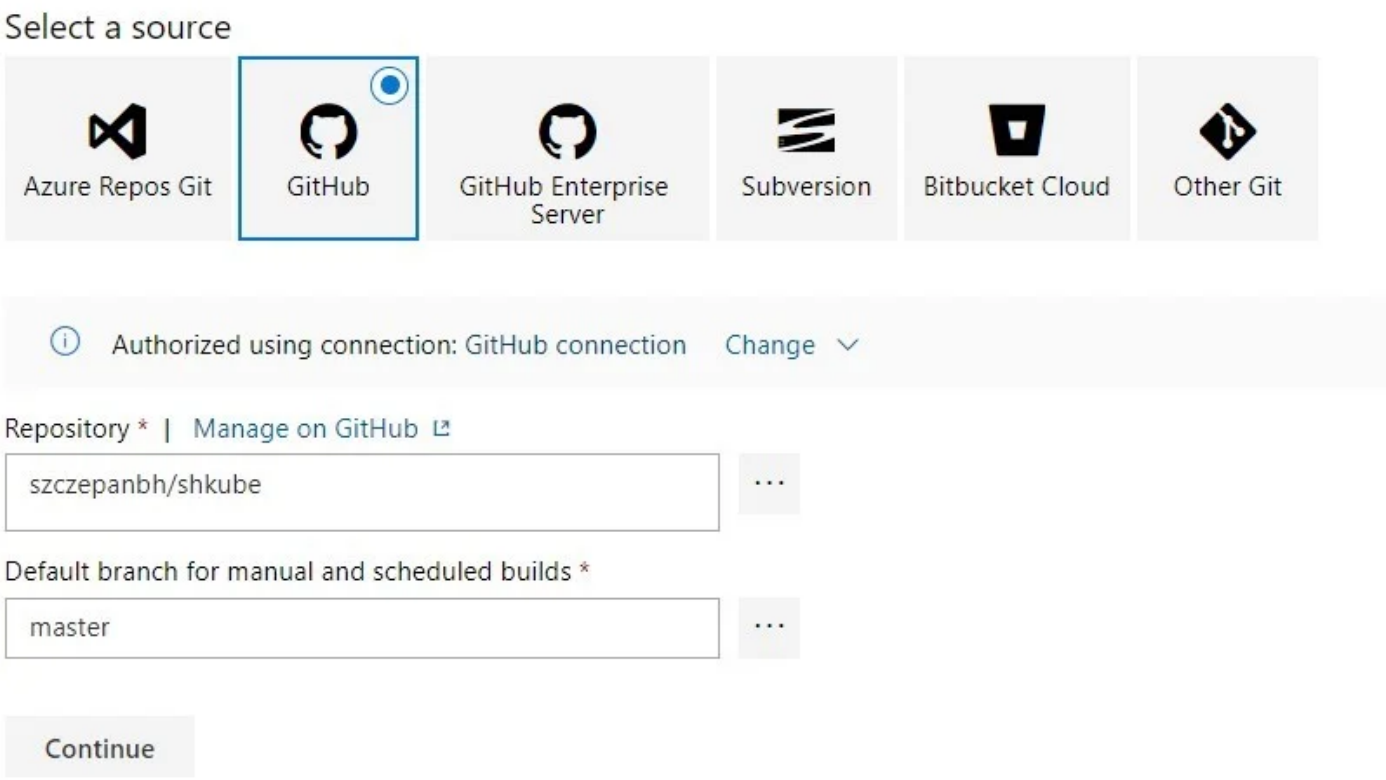
Go to your Azure DevOps website and create a new project.



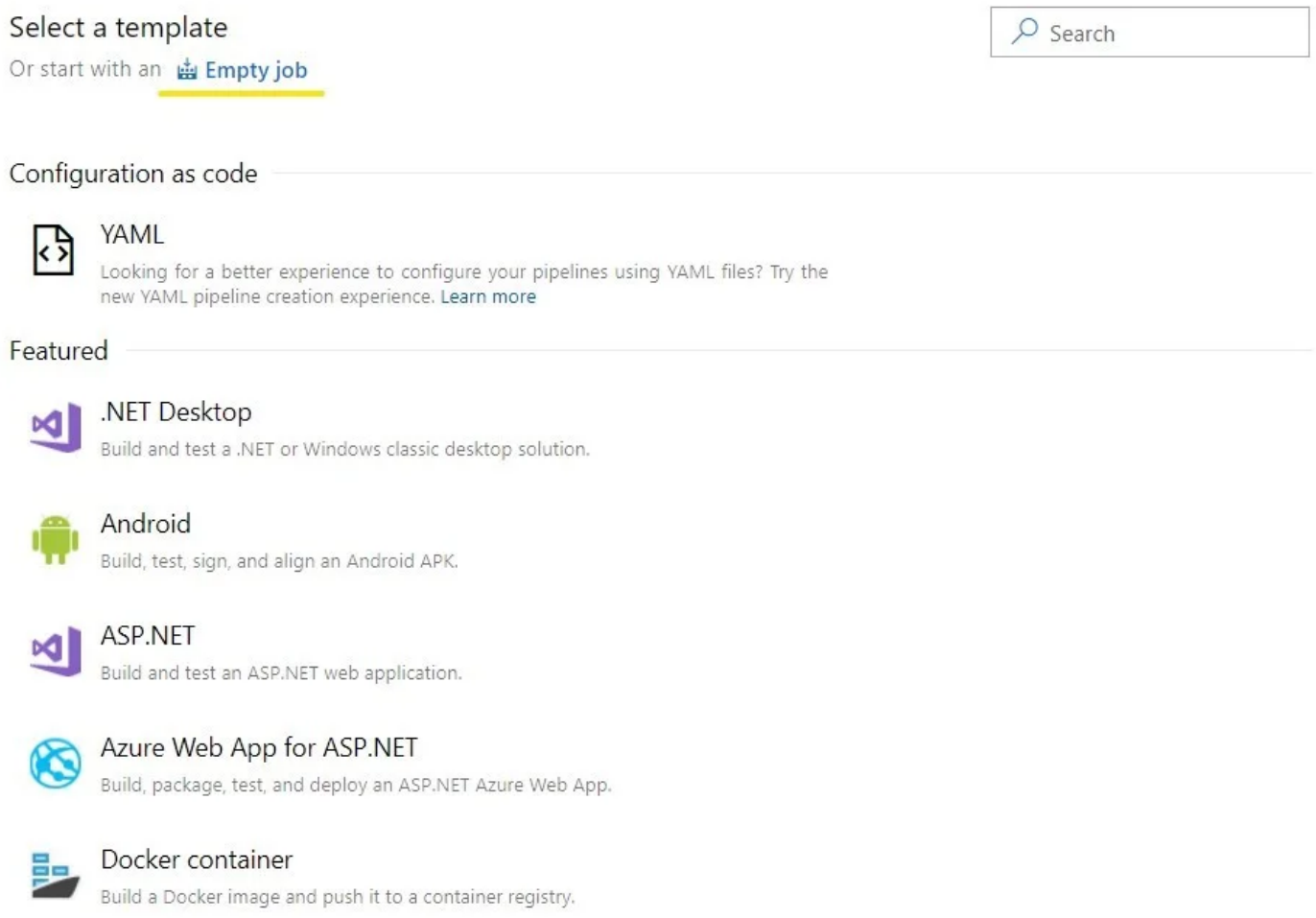
Create a new pipeline:



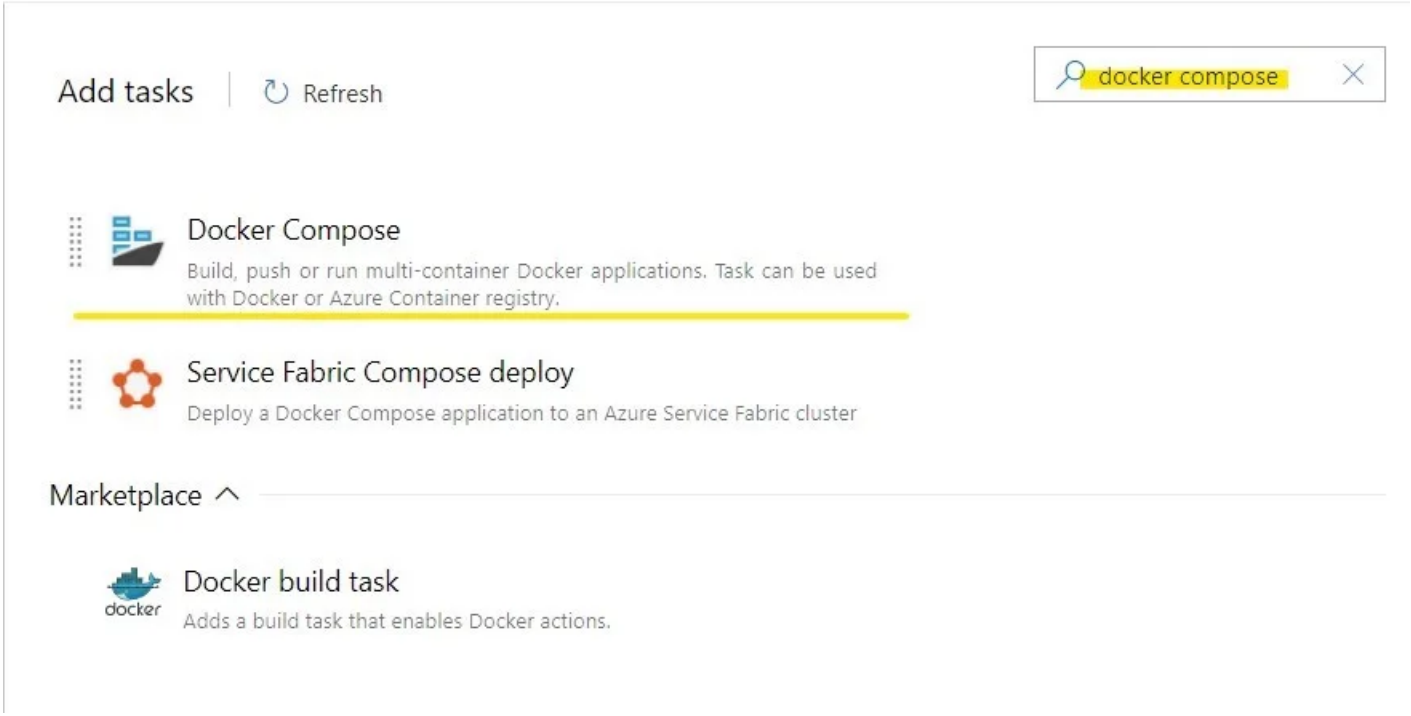
I'm going to use a classic editor and select Github repository. Pick your repository and a branch that you are going to use to build a container from this project.



If you see Docker Compose on a list, you can use it, but I will pick `Empty job`.



Use + button to add the first step of your pipeline! In this step we build a new docker image with your application. We use `Docker Compose` step that will use your `docker-compose.yml` file.



Now there are a few things to do:

- select your Azure Subscription
- select Azure Container Register

- put a path to your `docker-compose.yml` file
- change Action to `Build service image`

and put `$(Build.BuildId)` as Additional Image Tags (without this, we won't be able to determine which image version to deploy later).

Display name *

Build service

Container Registry Type *

Azure Container Registry

Azure subscription | Manage

Visual Studio Enterprise — MPN (15d3cb86-4161-46a4-adf6-f123b1ff28ec)

Scoped to subscription 'Visual Studio Enterprise — MPN'

Azure Container Registry

shkubeACR

Docker Compose File *

docker-compose.yml

Additional Docker Compose Files

Environment Variables

Project Name

$(Build.Repository.Name)

Qualify Image Names

Action *

Build service images

Additional Image Tags

$(Build.BuildId)

Great! Now we have to push this image to our ACR. Let's add second `Docker Compose` step. The only difference is the Action field: now pick `Push service image`.

Display name *

Push services

Container Registry Type *

Azure Container Registry

Azure subscription | Manage

Azure Container Registry

Docker Compose File *

docker-compose.yml

Additional Docker Compose Files

Environment Variables

Project Name

$(Build.Repository.Name)

Qualify Image Names

Action *

Push service images

The third step is optional but recommended. Lock an image version or a repository so that it can't be deleted or updated.

As before, add `Docker Compose` step. The only difference is the Action field. Now pick `Lock image service`. `Output Docker Compose File` will fill automatically.

Display name *

Lock services

Container Registry Type *

Azure Container Registry

Azure subscription | Manage

Visual Studio Enterprise — MPN (15d3cb86-4161-46a4-adf6-f123b1ff28ec)

Scoped to subscription 'Visual Studio Enterprise — MPN'

Azure Container Registry

shkubeACR

Docker Compose File *

docker-compose.yml

Additional Docker Compose Files

Environment Variables

Project Name

$(Build.Repository.Name)

Qualify Image Names

Action *

Lock service images

Remove Build Options

Base Resolve Directory

Output Docker Compose File *

$(Build.StagingDirectory)/docker-compose.yml

There are two more steps to follow.

Add `Copy Files` step.

- In `Contents` put a name or a path to your `deployment.yml` file (we will use this file during Release)
- In `Target folder` put `$(Build.ArtifactStagingDirectory)`

Copy files

Task version 2.*

Display name *

Copy Files

Source Folder

Contents *

deployment.yml

Target Folder *

$(Build.ArtifactStagingDirectory)

Advanced

Control Options

Output Variables

And `Publish build artifacts` step. Leave it as it is.

Publish build artifacts

Task version 1.*

Display name *

Publish Artifact: drop

Path to publish *

$(Build.ArtifactStagingDirectory)

Artifact name *

drop

Artifact publish location *

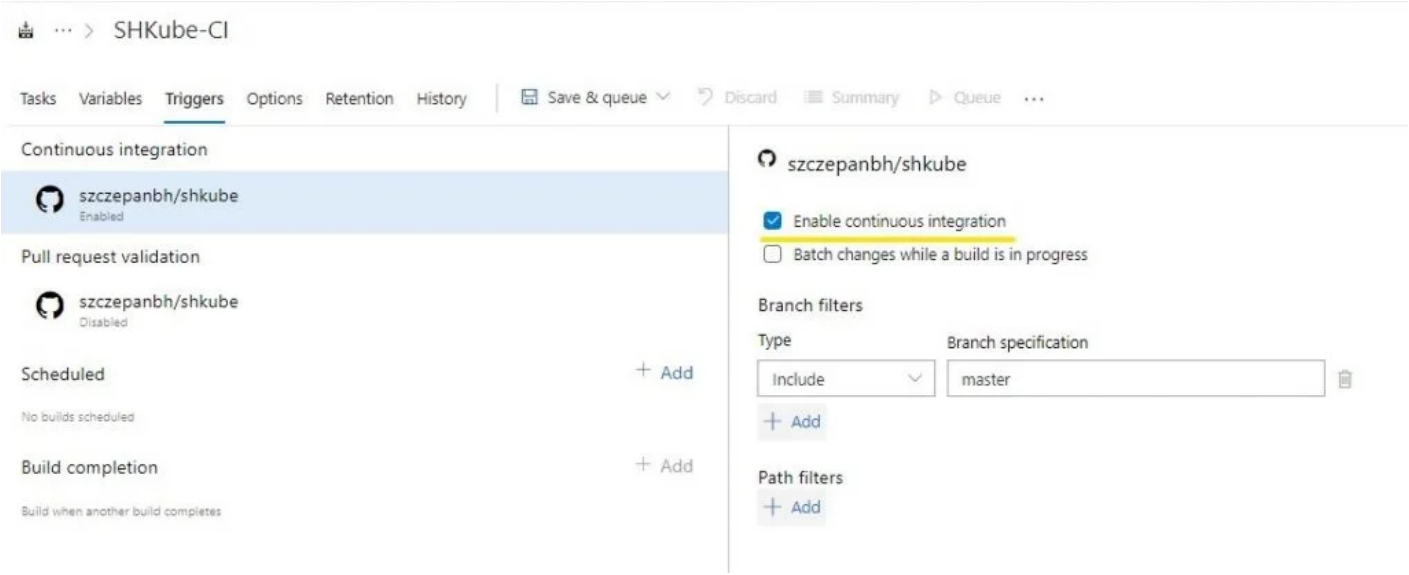
Azure Pipelines

Advanced

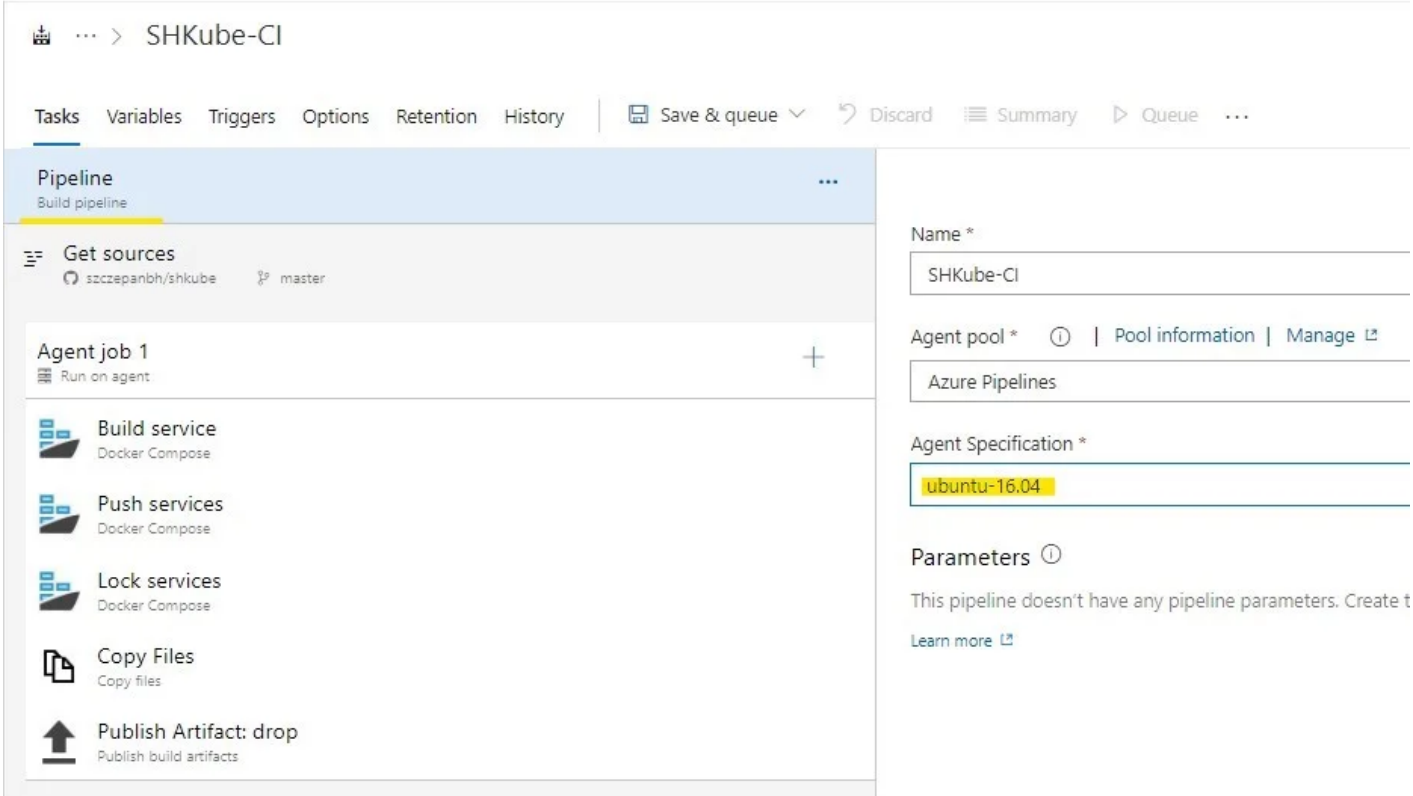
Control Options

Output Variables

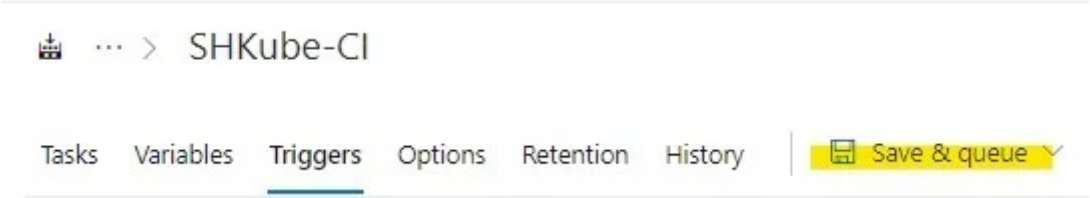
We will use `Artifact name` during Release. If you want your pipeline to trigger automatically after each merge, go to `Triggers` tab and select `Enable continuous integration`.



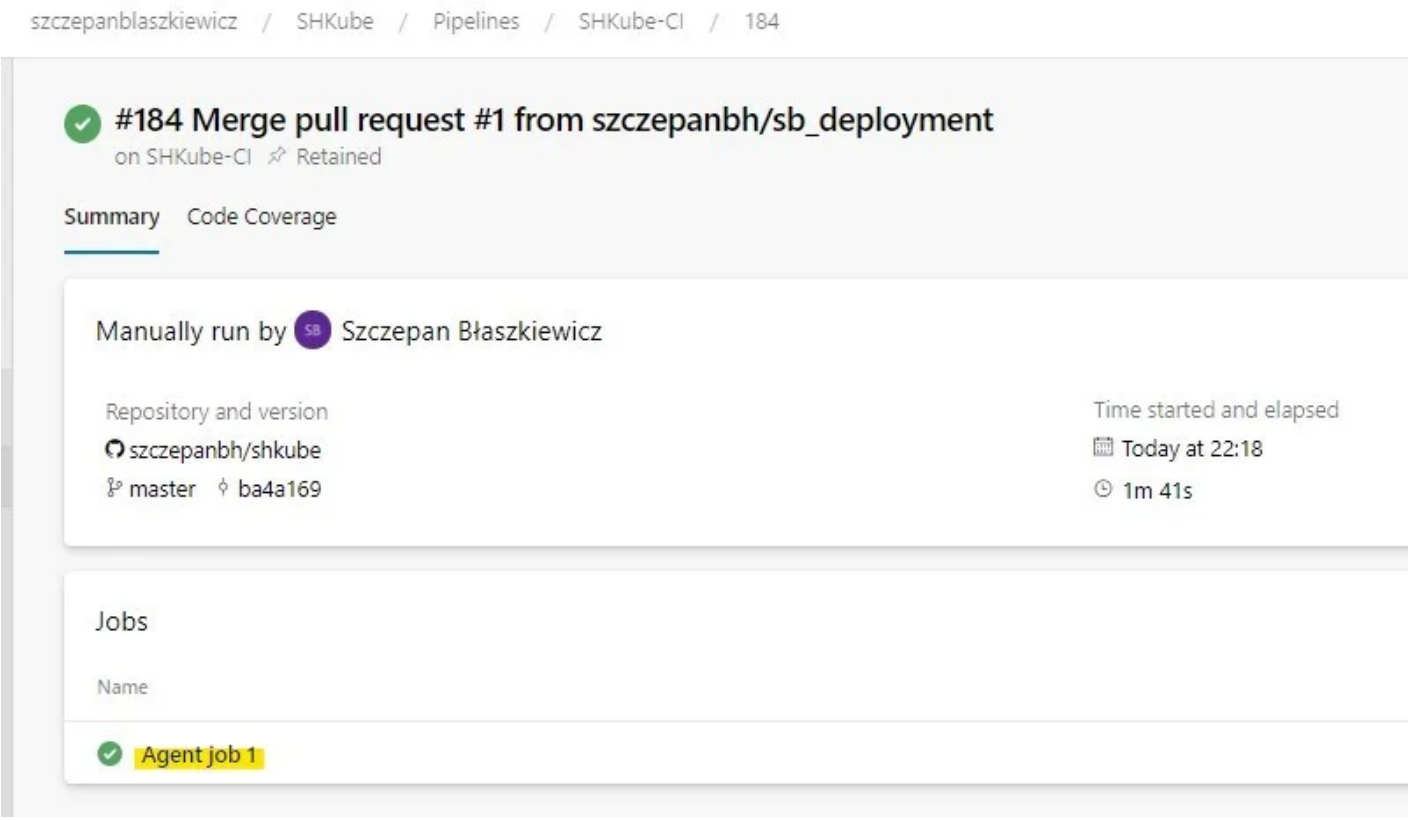
Click on `Build pipeline` and change `Agent Specification` to use Ubuntu (if you prefer Windows then you will have to change all paths to match Windows).



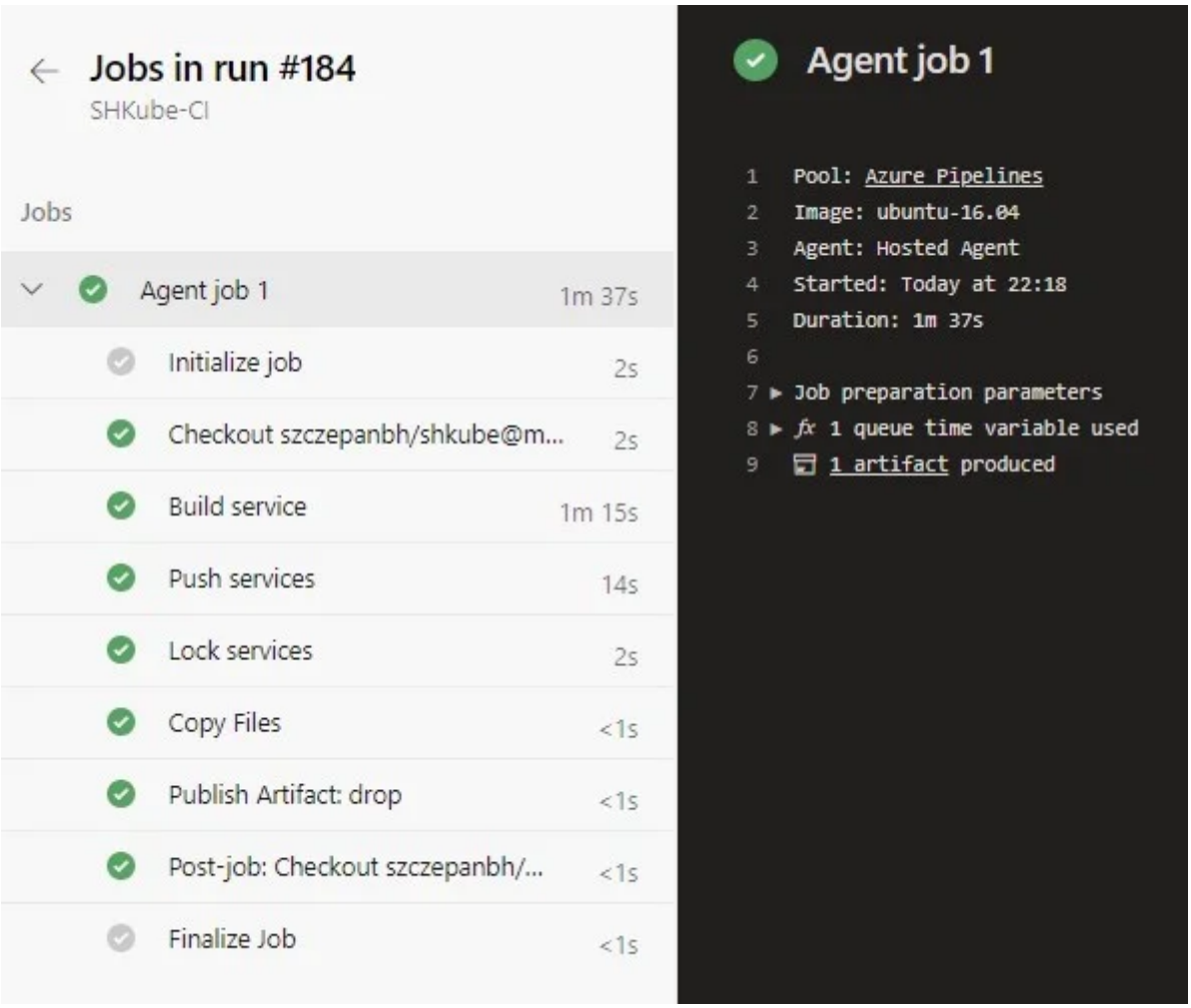
Now we can test our pipeline! Hit `Save & queue`.



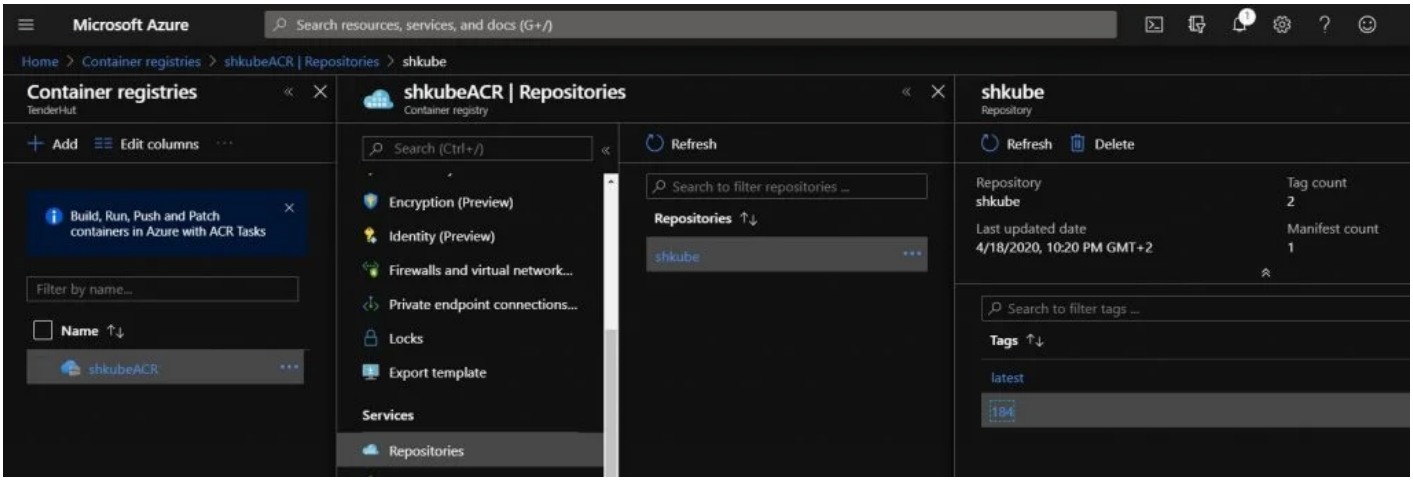
Go to Pipelines, select the pipeline you have created and pick the newest Run. If it's green, everything is ok.



If not, then you have to check which step went wrong and fix it.



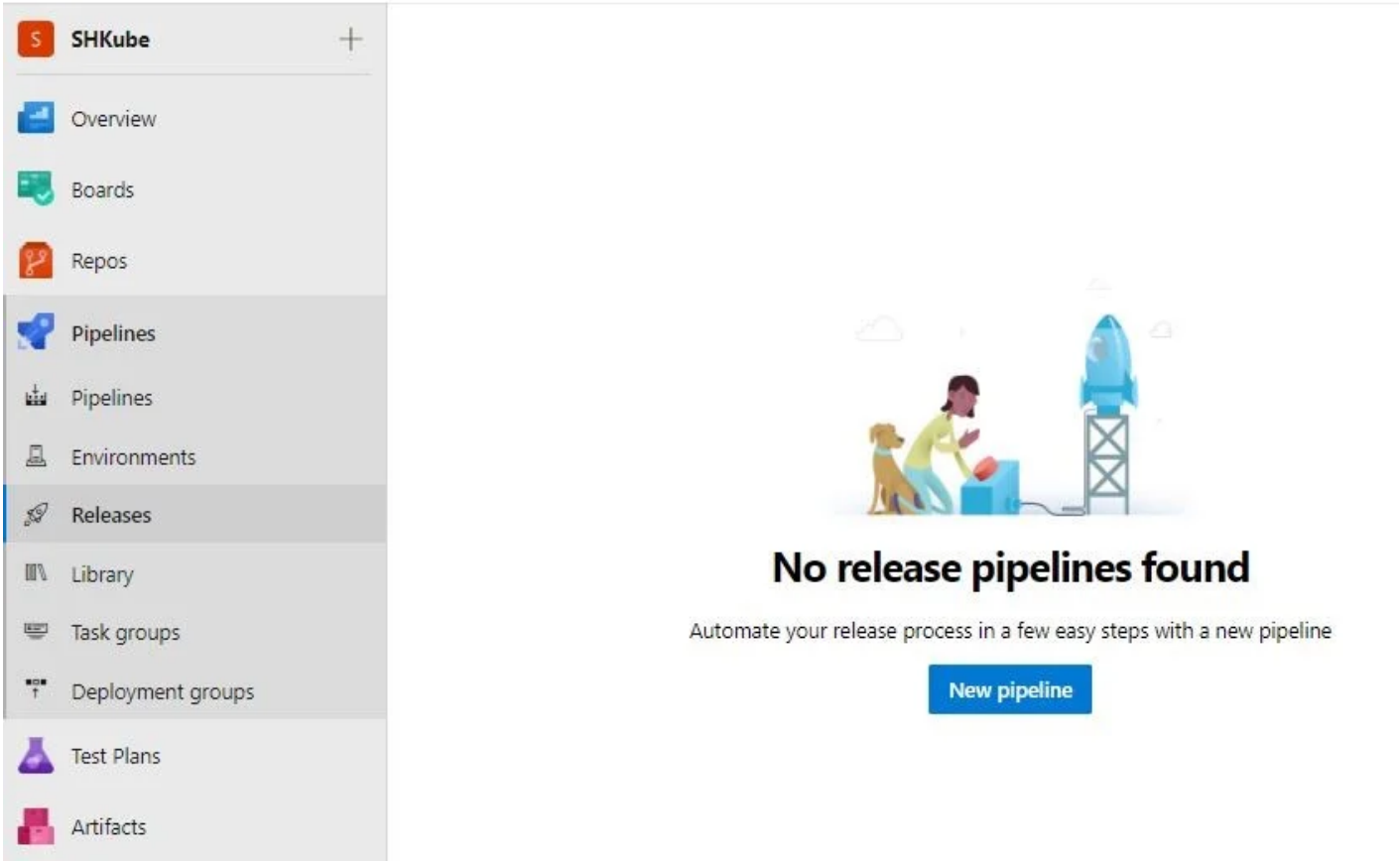
Congratulations! We're halfway through. We have an image and it's available on your ACR. Let's check it out. Please login to your Azure Portal and check if it's there (please notice that it's also tagged by its build number).



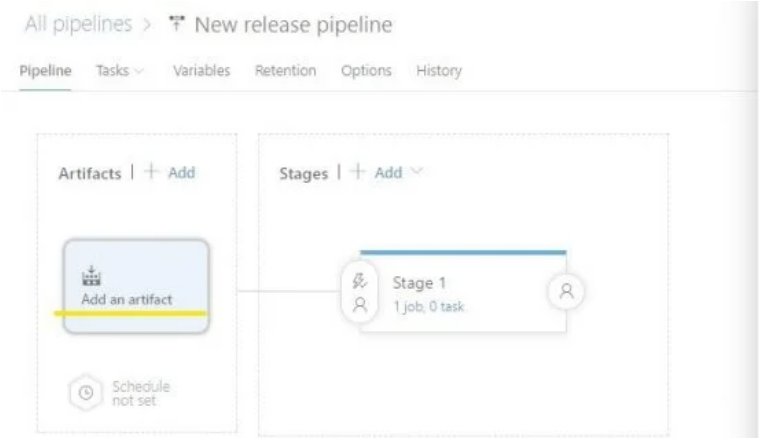
Now we can configure new Release.

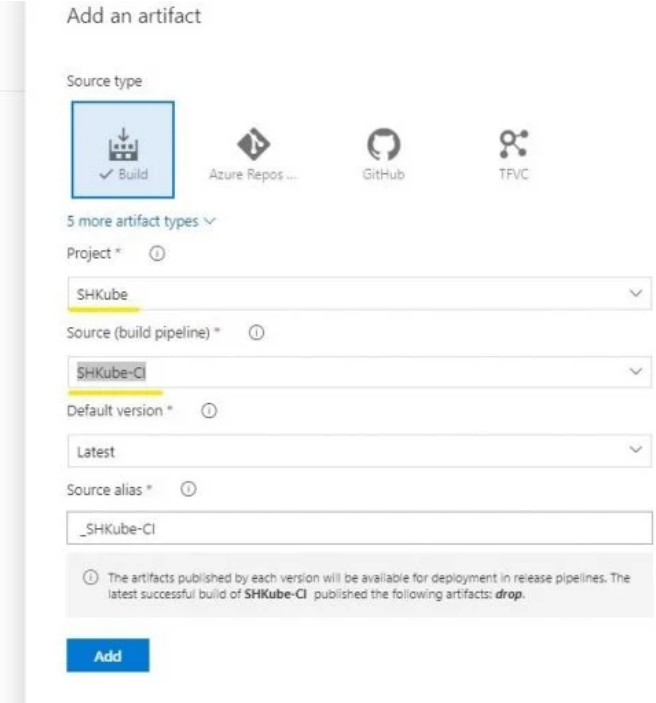
Azure Releases

Go to `Release` tab and create a new one.



Similarly to pipeline configuration, I have select `Empty job`. Now click on `Add an artifact`, select your project and source.





Go to `Tasks` tab add `Kubectl` task.

- change Service connection type,
- select Azure subscription,
- select resource group,
- select Kubernetes cluster,
- pick Apply Command,
- check Use configuration.

Display name *

Create Deployments & Services in AKS

Kubernetes Cluster

Service connection type *

Azure Resource Manager

Azure subscription * | Manage

Visual Studio Enterprise — MPN (15d3cb86-4161-46a4-adf6-f123b1ff28ec)

Scoped to subscription 'Visual Studio Enterprise — MPN'

Resource group *

shkubeRG

Kubernetes cluster *

shkubeAKS

Use cluster admin credentials

Namespace

Commands

Command

apply

Use configuration

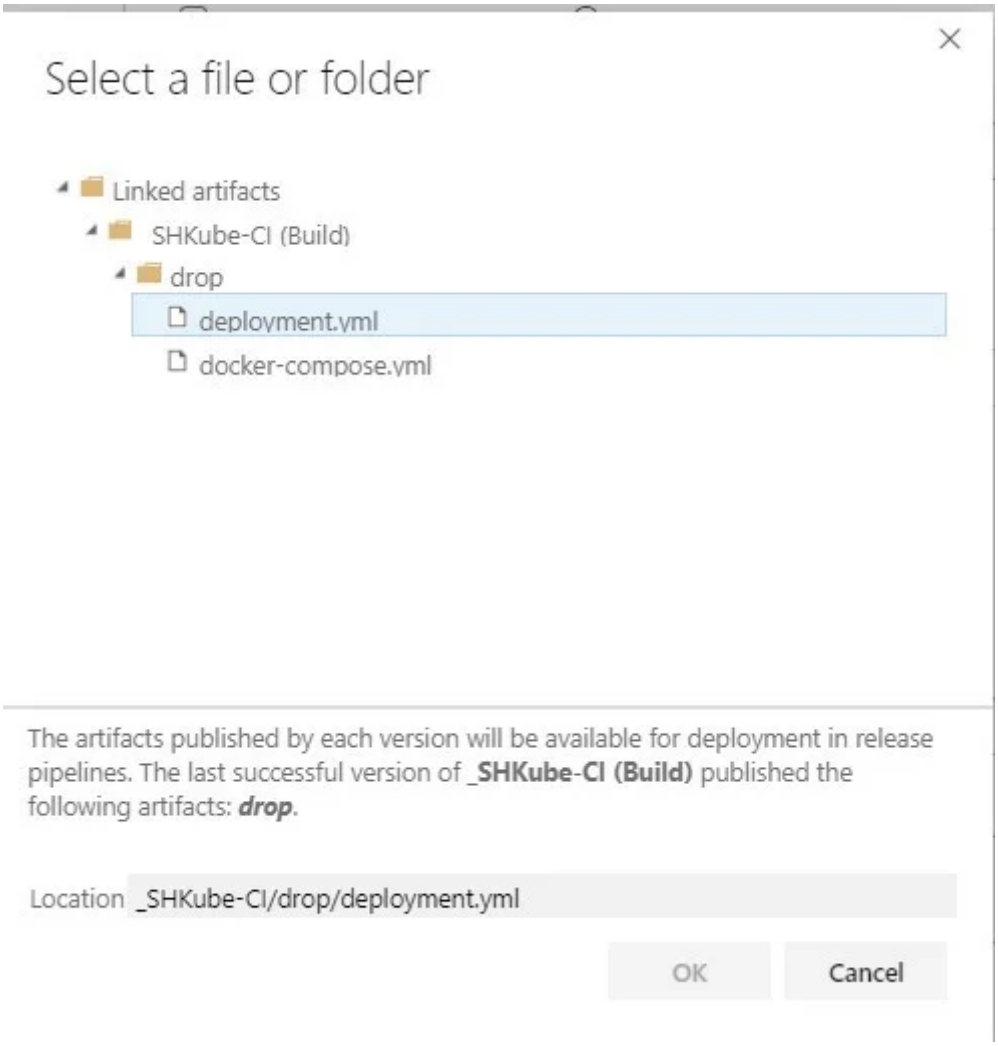
Configuration type

File path Inline configuration

File path *

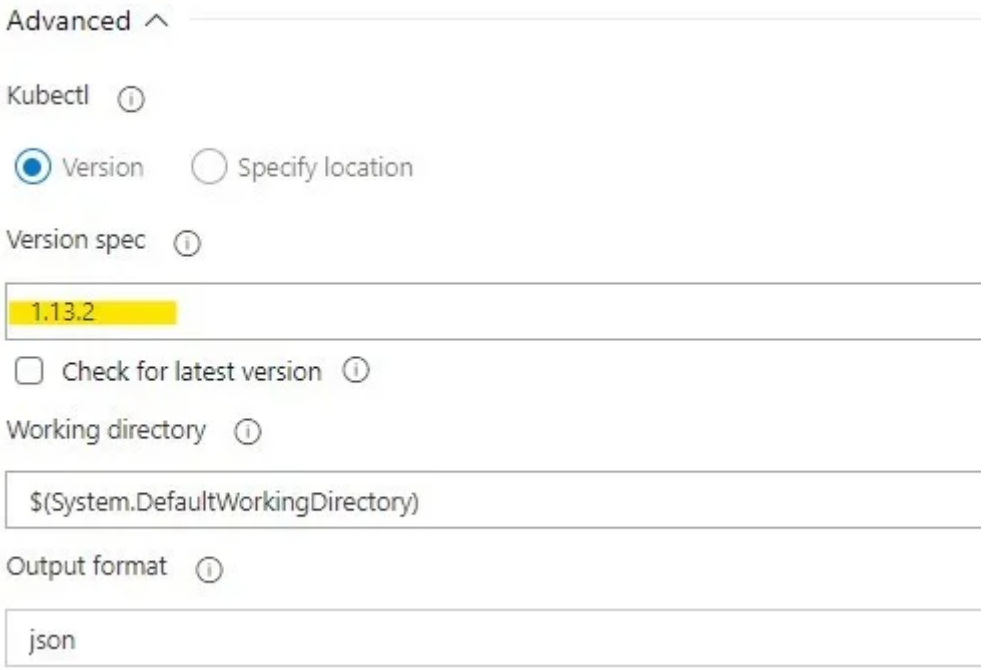
$(System.DefaultWorkingDirectory)/_SHKube-CI/drop/deployment.yml

And pick your `deployment.yml` file in File path.



Expand `Advance` and choose `Version spec` that matches the version of Kubernetes on Azure. You can use this command to check the current one:

```
az aks show -g $argName -n $aksName --output table
```



Add new `Kubectl` step (we will have to replace the name of your docker image to match the one on ACR).

Set everything as before but change `Command` to `set`, and `Arguments` input to `image deployments/shkube-deployment shkube=shkubeacr.azurecr.io/shkube:$(Build.BuildId)`.

This command is tricky, we have to navigate by name to deployment, and then to container image.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: shkube-deployment
  labels:
    app: shkube
spec:
  replicas: 1
  template:
    metadata:
      name: shkube
      labels:
        app: shkube
    spec:
      containers:
      - name: shkube
        image: shkube
        imagePullPolicy: IfNotPresent
      restartPolicy: Always
  selector:
    matchLabels:
      app: shkube

---
```

We specify the image version here. It's a better practice than just using latest because the latest image always points to the newest one (created after each merge). It could be ok for your dev environment but not for the production one.

Display name *

Update image

Kubernetes Cluster

Service connection type *

Azure Resource Manager

Azure subscription * | Manage

Visual Studio Enterprise — MPN (15d3cb86-4161-46a4-adf6-f123b1ff28ec)

Scoped to subscription 'Visual Studio Enterprise — MPN'

Resource group *

shkubeRG

Kubernetes cluster *

shkubeAKS

Use cluster admin credentials

Namespace

Commands

Command

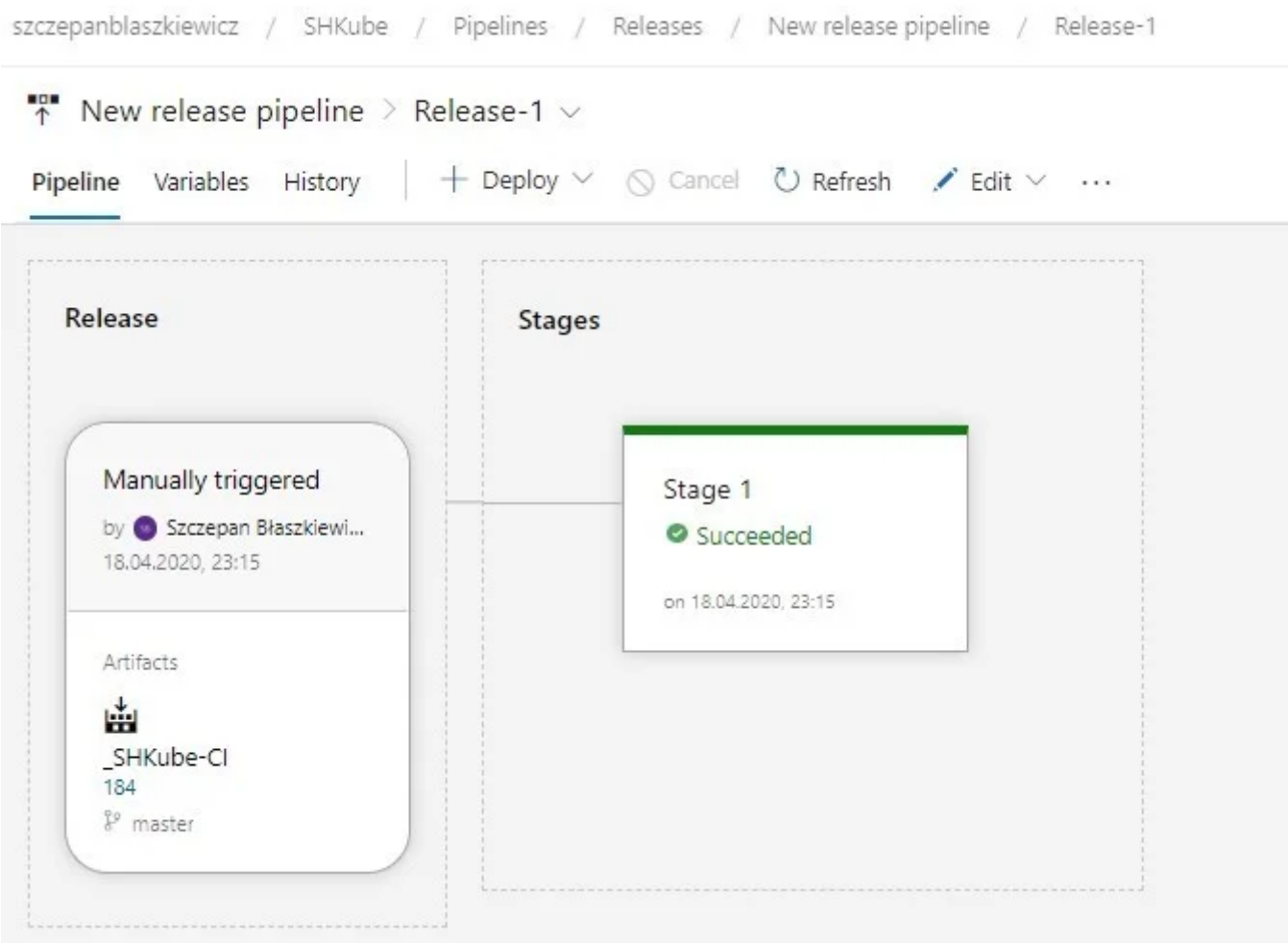
set

Use configuration

Arguments

image deployments/shkube-deployment shkube=shkubeacr.azurecr.io/shkube:$(Build.BuildId)

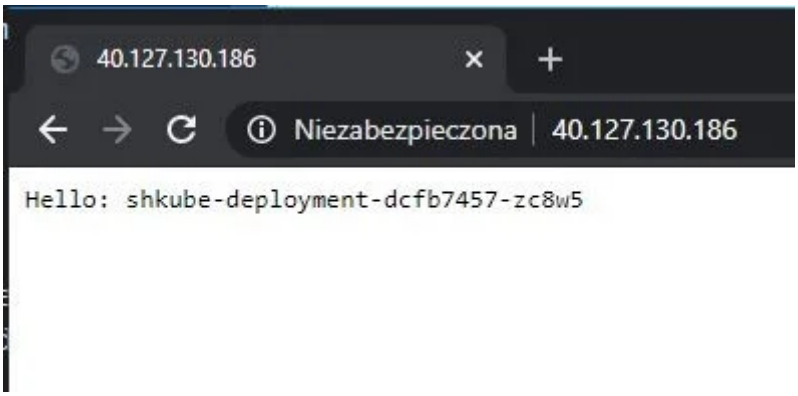
Click `Save` and `Create release` (top right) and go to Release to check if everything's alright.



Ok, let's go to console and check your services and pods.

```
szczepq@szczepan-dell  /c/Users/szcze/source/repos/blog/SHKube  sb_deployment ● ?  kubectl get svc
NAME             TYPE           CLUSTER-IP    EXTERNAL-IP      PORT(S)        AGE
kubernetes       ClusterIP      10.0.0.1      <none>           443/TCP        24h
shkube-service   LoadBalancer   10.0.212.55   40.127.130.186   80:31779/TCP   14m
szczepq@szczepan-dell  /c/Users/szcze/source/repos/blog/SHKube  sb_deployment ● ?  kubectl get pods
NAME                                  READY   STATUS    RESTARTS   AGE
shkube-deployment-dcfb7457-zc8w5      1/1     Running   0          39s
```

And navigate to IP of your service to check if it's up and running.



It's working! Congratulations!

Your Continuous Integration and Delivery setup is ready. You can now use your pipeline to create another Release configuration for your test, stage, or production environment.