Azure Devops Labs

Within these labs you will explore Azure Devops and learn the basics of the most important features used: yaml file, CI/CD, production/development environments and container steps. You will do this by:

* Developing a simple CI/CD pipeline to build and push an image to Azure Container Registry (ACR) and deploy it to Azure Kubernetes Service (AKS).
* Add additional script steps and tasks.
* Use AKS namespaces to build different environments for productions and development.
* Use pull request.
* Add flake8 as a python linter in the CI/CD pipeline.
* Add isort to sort imports alphabetically.
* Add black as Python code formatter.
* Add additional tests and features to the app provided.
* Add a containerized step.

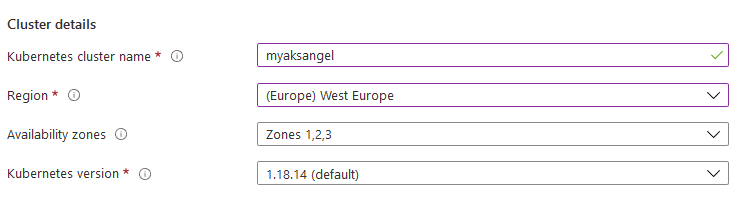
Before you can start building, you need an AKS cluster and an ACR, so that will be our first step.

# 

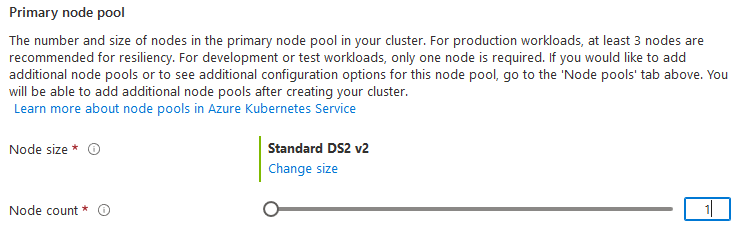
# Lab 1: Create an AKS cluster and an ACR (To be carried out by the teacher).

Within this lab you need a functional AKS cluster and an ACR. After this lab, you will have your own set of resources. For completeness, it has been included here a manual creation of AKS cluster. However, it has been automated, check the ‘deployment’ folder for additional details.

1. To create an AKS cluster with an ACR, go to the Azure Portal > Resource group and press ‘Add Resource’ ( ), look for ‘Kubernetes Service’ and create one.
2. Give it a name (each a different name) and select (Europe) West Europe Region:



1. For testing purposes, only one node is needed in ‘Node count’



1. Proceed with the defaults until ‘Integration’, where a new Container registry should be created. Press ‘Create new’:



1. Enter a unique name and press ‘Ok’:

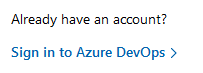


1. Press review and create and create the AKS cluster.

# Lab 1: CI/CD using a yaml file (To be carried out by the teacher).

Within this lab you will explore the Azure Devops Pipeline with Azure repos. A yaml file will be generated with all the information about the CI/CD pipeline. After this lab you will have your own CI/CD pipeline to deploy a simple python Flask app.

1. Go to [Azure Devops login](https://azure.microsoft.com/en-us/services/devops/) and press ‘Sign in to Azure Devops’.



1. Go to Repos and import a new repo called ‘myapp’ from the following:

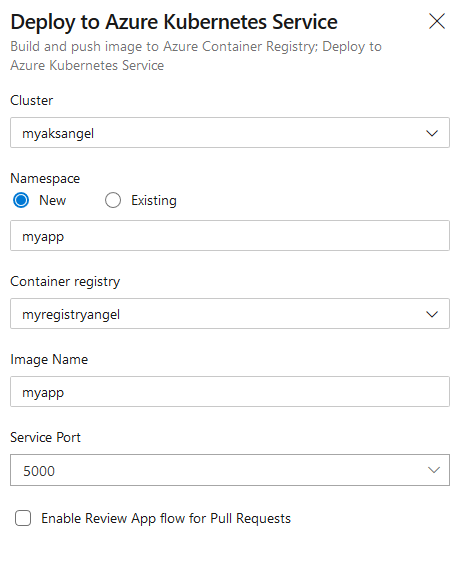
<https://github.com/Anchormen/azuredevops>

1. Go to pipelines and press ‘New pipeline’. Select ‘Azure Repos Git’ and the ‘myapp’ repository that was created in the previous step.

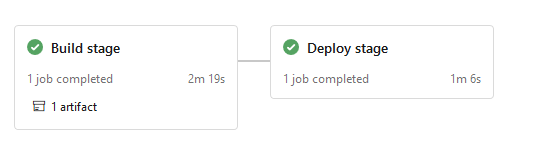
****

1. Select Deploy to Azure Kubernetes Service and add:

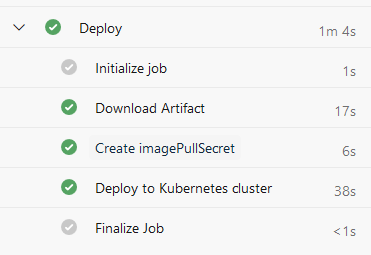
* AKS Cluster.
* As namespace, set new and add a name, for example, ‘myapp’.
* As Container registry, select the one created in Lab 1.
* Give a name to the image, for example, ‘myapp’.
* Confirm that the Service Port is 5000.



1. Press , check all the defaults and press  as required.
2. In the meantime, check the files that were created automatically by Azure Devops, namely, ‘azure-pipelines.yml’ and the folder ‘manifests’.
3. Go to Pipelines and check the status of the deployment in the AKS cluster. When this is finished, go to Pipelines > Pipeline yaml > Last Job and click on Stages > Deploy stage:



The following should be shown:



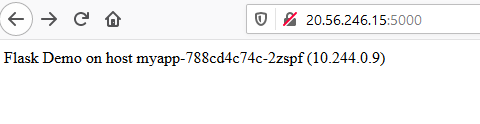
1. Click on ‘Deploy to Kubernetes cluster’ and look for the external IP:



1. Open your favorite browser and go to the following URL:

http://<EXTERNAL\_IP>:5000/

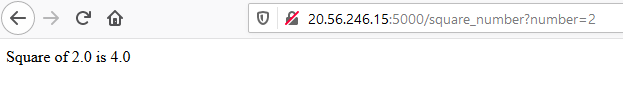
The following should be shown:



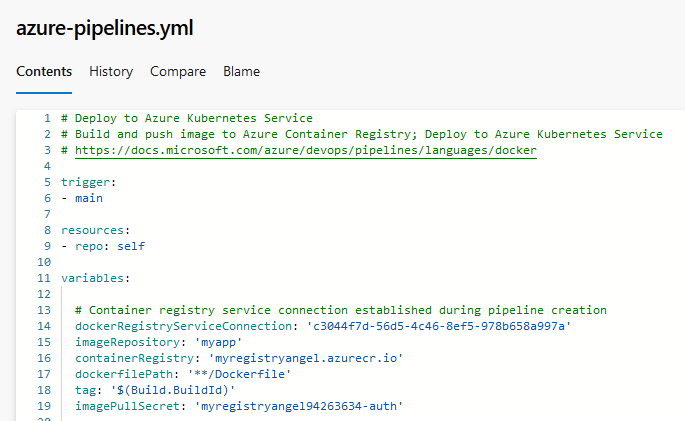
Additionally, check that the flask app can compute squares by going to the following URL:

http://<EXTERNAL\_IP>:5000/square\_number?number=2

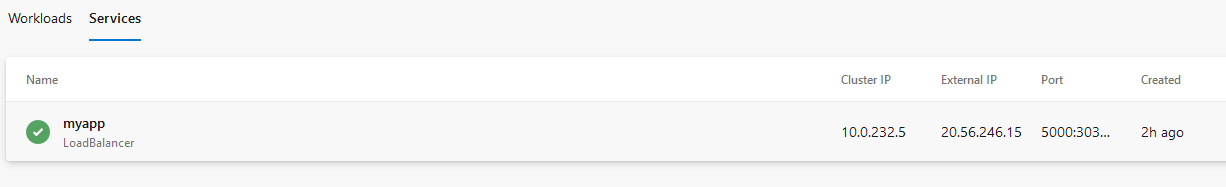
The following should be shown:

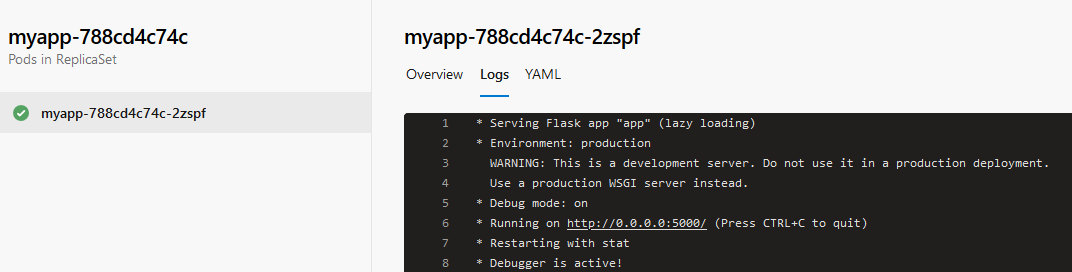


1. Go to the repository and take a look to the new file generated by the Azure pipeline called azure-pipeline.yaml:



1. Go to Pipelines > Environments, a new environment called ‘myapp’ should have been created. Here, the external IP as well as important information like Logs, can be checked.

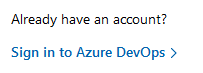




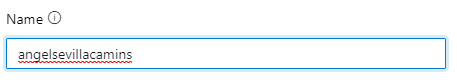
# Lab 2: Adding production/development environments.

Applications should be always available, independently that additional developments are being carried out. For that, a stable environment, called production or prod, will only be changed updated after changed are tested in a different environment, called, for example, development or dev. After this lab you will have two environments, namely, dev and prod, as two namespaces in a common AKS cluster.

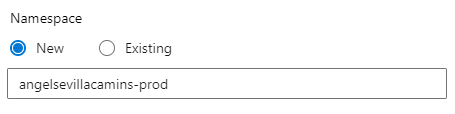
1. Go to [Azure Devops login](https://azure.microsoft.com/en-us/services/devops/) and press ‘Sign in to Azure Devops’.



1. Go to pipelines > Environments and press .
2. Set your name as name of the environment



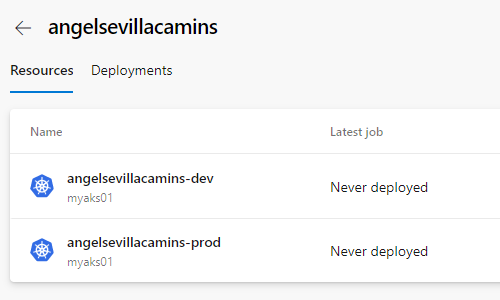
1. Select Kubernetes as resource .
2. Press next.
3. The AKS cluster created in the Lab 1 should be filled automatically.
4. As namespace, select ‘New’ and name it ‘YOUR\_NAME-prod’:



1. Press 
2. Press  and select 
3. Repeat the process to create another namespace called ‘YOUR\_NAME-prod’.



1. Check that both namespaces were created:



**Since the next steps are tedious and error-prone, they have been automated for you. Follow the instructions of the teacher.**

1. Go to pipelines > my app pipeline and press .
2. From now on, branch called dev will be deployed in the ‘dev’ namespace. Whereas commits on the branch ‘master’ will be deployed in the ‘prod’ namespace. In this way, we can test additional features in the ‘dev’ namespace before being deployed in the ‘prod’ namespace.
3. Deployment stage will not be run in any other branch but build stage will run also in branches called feature/\*.
4. Go to Pipelines and select the pipeline ‘myapp’.
5. Click on  to incorporate the following changes:
   1. To trigger the pipeline when branch dev is modified, add - dev as follows:

trigger:

- master

- dev

- feature/\*

* 1. To change the environment in which commits are going to be deployed, add the following at the end of the variables key:

variables:

  …

  # Environment Deployment depends on branch

  ${{ if eq(variables['Build.SourceBranchName'], 'dev') }}:

    environmentDeployment: 'dev.dev'

  ${{ if eq(variables['Build.SourceBranchName'], 'master') }}:

environmentDeployment: 'prod.prod'

Here, depending on the branch name, a different value to the ‘environmentDeployment’ variable is assigned.

The way to describe namespaces within environments in Azure devops is:

<NAME OF THE ENVIRONMENT>.<NAMESPACE>

* 1. As set before, development will only occur when previous steps are successful and for branches ‘dev’ and ‘master’. Regarding the latter, we will check if the variable created in the previous step is not empty, which is not allowed in any case.

- stage: Deploy

  displayName: Deploy stage

  dependsOn: Build

**condition:  and(succeeded(), ne(variables['environmentDeployment'], ''))**

* 1. To change the environment in which commits are going to be deployed, change the ‘environment’ value within the ‘Deploy’ job to:

 jobs:

  - deployment: Deploy

    displayName: Deploy

    pool:

      vmImage: $(vmImageName)

**environment: $(environmentDeployment)**

1. After all the modifications, it should look like the following:

# Deploy to Azure Kubernetes Service

# Build and push image to Azure Container Registry; Deploy to Azure Kubernetes Service

# https://docs.microsoft.com/azure/devops/pipelines/languages/docker

trigger:

- master

- dev

- feature/\*

resources:

- repo: self

variables:

  # Container registry service connection established during pipeline creation

  dockerRegistryServiceConnection: ' 09b503ea-722b-418f-b9e4-d7e478987f07'

  imageRepository: 'camins'

  containerRegistry: ' itvitaedockerregistry.azurecr.io'

  dockerfilePath: '\*\*/Dockerfile'

  tag: '$(Build.BuildId)'

  imagePullSecret: itvitaedockerregistry16527679-auth'

  # Agent VM image name

  vmImageName: 'ubuntu-latest'

  # Environment Deployment depends on branch

  ${{ if eq(variables['Build.SourceBranchName'], 'dev') }}:

    environmentDeployment: 'camins-dev.camins-dev'

  ${{ if eq(variables['Build.SourceBranchName'], 'master') }}:

    environmentDeployment: 'camins.camins-prod'

stages:

- stage: Build

  displayName: Build stage

  jobs:

  - job: Build

    displayName: Build

    pool:

      vmImage: $(vmImageName)

    steps:

    - task: Docker@2

      displayName: Build and push an image to container registry

      inputs:

        command: buildAndPush

        repository: $(imageRepository)

        dockerfile: $(dockerfilePath)

        containerRegistry: $(dockerRegistryServiceConnection)

        tags: |

          $(tag)

    - upload: manifests

      artifact: manifests

- stage: Deploy

  displayName: Deploy stage

  dependsOn: Build

  condition:  and(succeeded(), ne(variables['environmentDeployment'], ''))

  jobs:

  - deployment: Deploy

    displayName: Deploy

    pool:

      vmImage: $(vmImageName)

    environment: $(environmentDeployment)

    strategy:

      runOnce:

        deploy:

          steps:

          - task: KubernetesManifest@0

            displayName: Create imagePullSecret

            inputs:

              action: createSecret

              secretName: $(imagePullSecret)

              dockerRegistryEndpoint: $(dockerRegistryServiceConnection)

          - task: KubernetesManifest@0

            displayName: Deploy to Kubernetes cluster

            inputs:

              action: deploy

              manifests: |

                $(Pipeline.Workspace)/manifests/deployment.yml

                $(Pipeline.Workspace)/manifests/service.yml

              imagePullSecrets: |

                $(imagePullSecret)

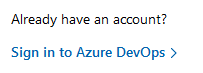
              containers: |

                $(containerRegistry)/$(imageRepository):$(tag)

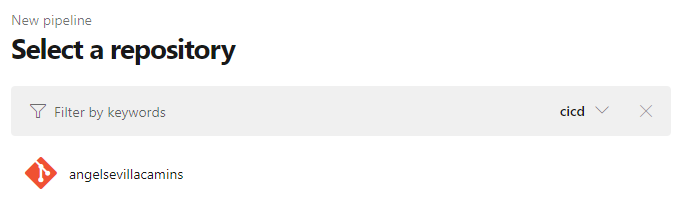
1. Press  and commit the changes

# Lab 3: Create pipelines for dev and prod using dev and master branches.

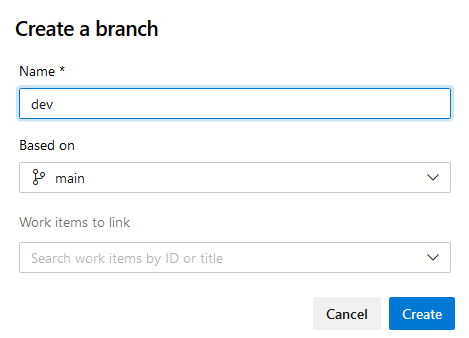
1. Go to [Azure Devops login](https://azure.microsoft.com/en-us/services/devops/) and press ‘Sign in to Azure Devops’.



1. Go to 
2. Press 
3. Select 
4. Pick the repo with your name:



1. Press  and wait until the pipeline is finished.
2. Check that the app has been deployed in the ‘prod’ environment in Pipelines > Environments > Resources > Services. It might take a while, be patient.
3. Go to Repos > Branches and click .
4. Create a branch called ‘dev’ based on the master branch as follows:



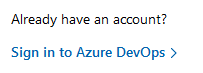
1. Press  and wait until the pipeline is finished.
2. Check that the app has been deployed in the ‘dev’ environment in Pipelines > Environments > Resources > Services. It might take a while, be patient.

# 

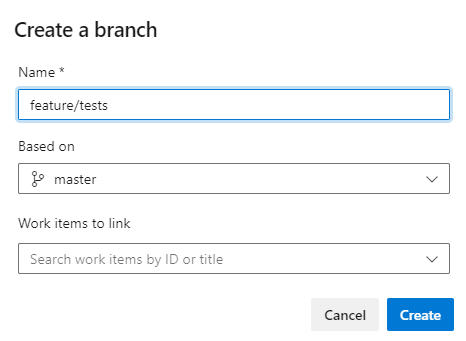
# Lab 4: Adding script steps and tasks to a pipeline using a yaml file.

Applications should be tested before initializing the deployment process. In this way, errors or bugs can be found before even the application is deployed in the ‘dev’ environment. After this lab you will test your app before building the docker image.

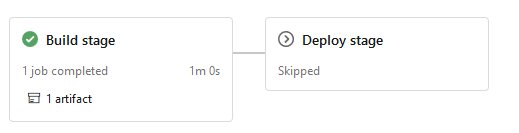
1. Go to [Azure Devops login](https://azure.microsoft.com/en-us/services/devops/) and press ‘Sign in to Azure Devops’.



1. Go to Repos > Branches and click .
2. Create a branch called ‘feature/tests’ based on the ‘dev’ branch as follows.



1. Go to pipelines > my app and wait until the new pipeline is finished. Check that ONLY the Build stage was carried out:



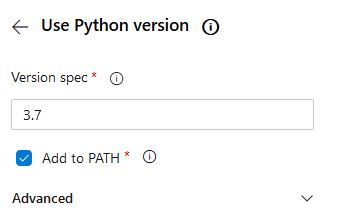
1. Go to pipelines, select the one that you created the previous lab and press .
2. **IMPORTANT!!! Select the proper branch on the slide menu:**



1. An additional step will be added to the job ‘Build’. Put the cursor after the ‘steps:’ line in the Stage Build (see below at the bottom of the figure).



1. At the right side of the page, in the tasks panel, search for ‘python’ and select ‘Python version’. Then, use 3.7 as ‘Version spec’ and check that ‘Add to PATH’ is selected. Finally, and press ‘Add’.



1. After the text automatically generated in the previous step, add a blank space and the following snippet from :

    - script: |

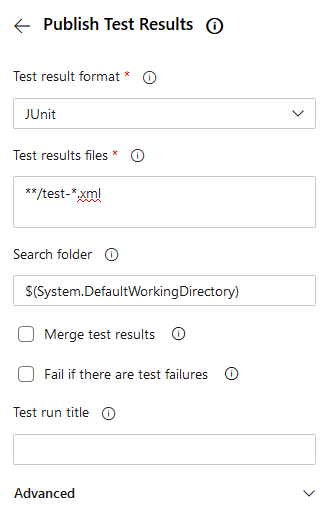
        pip install -r requirements.txt

        pytest --junitxml=junit/test-results.xml --cov=. --cov-report=xml

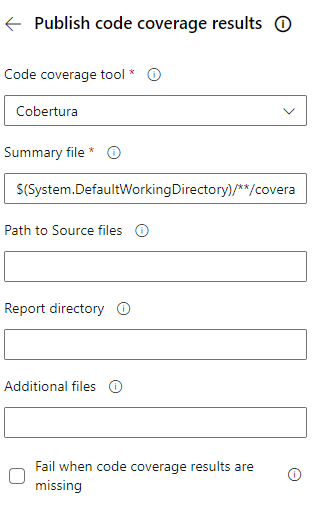
      displayName: 'Install dependencies and test with pytest'

This will create script to install the required dependencies including pytest and run the tests. As output, it will generate a file called ‘test-results.xml’ using the JUnit format.

1. Add a blank space and search for ‘publish’ in the tasks panel. Select ‘Publish Test Results’. Then, modify the ‘Test result files’ as ‘\*\*/test-\*.xml’ (the default contains TEST in UPPERCASE and it should be changed) and press ‘Add’.



1. Add a blank space and search for ‘publish’ in the tasks panel. Select ‘Publish code coverage results’. Then, select ‘Cobertura’ as ‘Code Coverage tool’ and as summary file '$(System.DefaultWorkingDirectory)/\*\*/coverage.xml'. Finally, press ‘Add’.



1. After all the modifications, the Build stage should look like the following:

stages:

- stage: Build

  displayName: Build stage

  jobs:

  - job: Build

    displayName: Build

    pool:

      vmImage: $(vmImageName)

    steps:

    - task: UsePythonVersion@0

      inputs:

        versionSpec: '3.7'

        addToPath: true

        architecture: 'x64'

    - script: |

        pip install -r requirements.txt

        pytest --junitxml=junit/test-results.xml --cov=. --cov-report=xml

      displayName: 'Install dependencies and test with pytest'

    - task: PublishTestResults@2

      inputs:

        testResultsFormat: 'JUnit'

        testResultsFiles: '\*\*/test-\*.xml'

    - task: PublishCodeCoverageResults@1

      inputs:

        codeCoverageTool: Cobertura

        summaryFileLocation: '$(System.DefaultWorkingDirectory)/\*\*/coverage.xml'

    - task: Docker@2

      displayName: Build and push an image to container registry

      inputs:

        command: buildAndPush

        repository: $(imageRepository)

        dockerfile: $(dockerfilePath)

        containerRegistry: $(dockerRegistryServiceConnection)

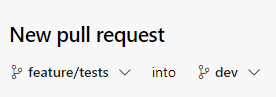
        tags: |

          $(tag)

    - upload: manifests

      artifact: manifests

1. Press  and commit the changes. Wait until the pipeline is finished.
2. Check that the tests results are also available in the pipeline outcome.
3. Go to 
4. Create a new pull request to the ‘dev’ branch and check that a new pipeline was triggered. In this case, it should be deployed into the ‘dev’ namespace.



1. Add a meaningful name and press . In a real world scenario, some reviewers should be added.
2. Press 
3. Check that ‘Delete feature/tests after merging’ and press 
4. Check that a new pipeline was triggered

# Lab 5: Install VS code and connect with the repository.

For some of the integration tools, like linting or code formatting, it is important to use a graphical user interface to produce code and check it in advance. In this lab, VScode will be installed in a Virtual Machine with Ubuntu. Additionally, a connection to the Git repos created in the previous lab will be set up with SSH.

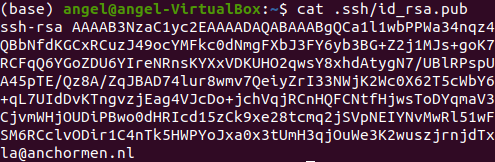
1. Create a ssh key by running the following command. Modify YOUR\_EMAIL\_HERE accordingly.

ssh-keygen -C "YOUR\_EMAIL\_HERE"

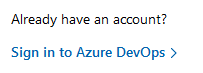
1. This command produces the two keys needed for SSH authentication: your private key ( id\_rsa ) and the public key ( id\_rsa.pub ). It is important to never share the contents of your private key. To display the public key run:

cat .ssh/id\_rsa.pub

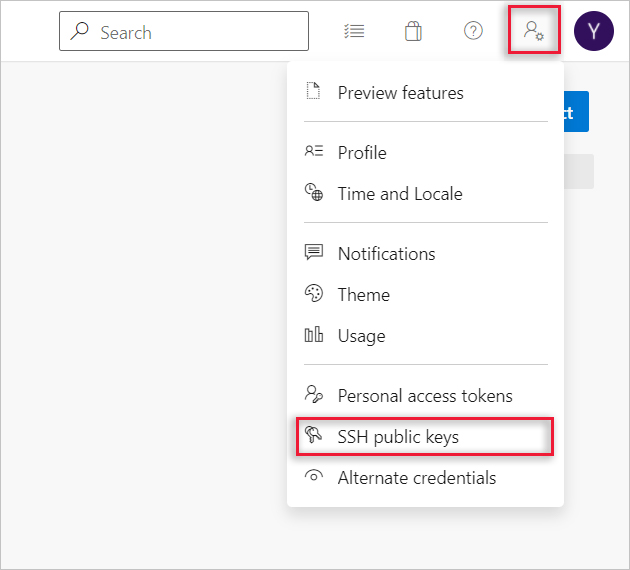
Copy the public key, which should look as below:



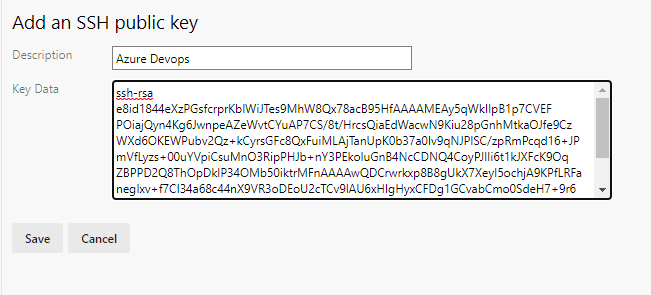
1. Add the public key to Azure DevOps. For that go to Go to [Azure Devops login](https://azure.microsoft.com/en-us/services/devops/) and press ‘Sign in to Azure Devops’.



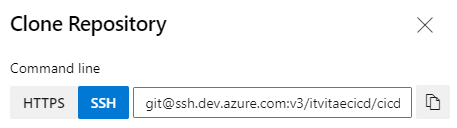
1. Associate the public key generated in the previous step with your user ID.



1. Press  and give it a name like ‘Azure Devops’ and paste your PUBLIC KEY:



1. Go to your repository and press in 
2. Select SSH and copy the content:



1. Go to your terminal in the VM and run the following commands, replacing the last part with the command copied in the previous step:

git clone [git@ssh.dev.azure.com:v3/itvitaecicd/cicd/YOUR\_NAME](mailto:git@ssh.dev.azure.com:v3/itvitaecicd/cicd/YOUR_NAME)

cd <YOUR\_NAME>

1. Anaconda should be installed, follow this [link](https://www.anaconda.com/products/individual#linux) if it was not installed previously.
2. Run the following commands to create a new environment with the dependencies of the exercise:

conda create --name flask python=3.7 --file requirements.txt -y

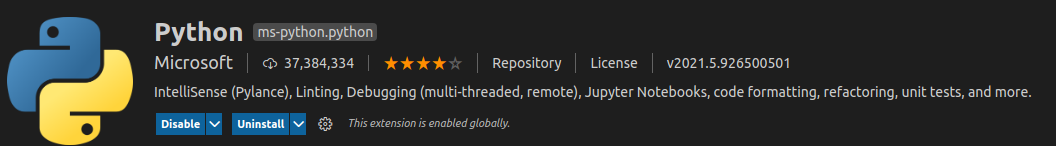
conda activate flask

1. Install VSCode and execute by running:

sudo snap install --classic code

code .

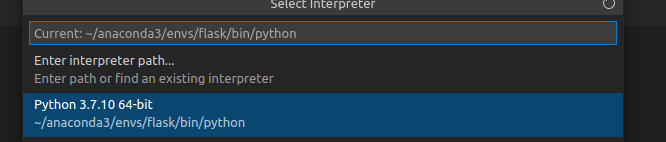
1. Install python extension by pressing . Search for ‘Python’ and install the extension:



1. Once it is installed, look at the bottom of the screen, the following icon should appear. If not, restart VSCode:



1. Press this icon and select the proper environment:



1. Press CTRL + SHIFT + P and look for ‘lint’. Press in the following and enable linting:



1. Press CTRL + SHIFT + P and look for ‘lint’. Press in the following:

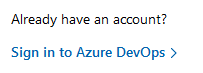


Select **flake8** as linter.

# Lab 6: Adding additional script steps.

Linting highlights syntactical and stylistic problems in your Python source code. One of the most used linters for python is [flake8](https://flake8.pycqa.org/en/latest/). After this lab, flake8 will be installed and run before the app is deployed.

1. Go to [Azure Devops login](https://azure.microsoft.com/en-us/services/devops/) and press ‘Sign in to Azure Devops’.



1. Create a branch called ‘feature/additionalsteps’ based on the ‘dev’ branch but this time use git from the command line. “git checkout -b <name>”
2. Review the flake8 documentation [here](https://flake8.pycqa.org/en/latest/) and the Python Azure devops documentation [here](https://docs.microsoft.com/en-us/azure/devops/pipelines/ecosystems/python?view=azure-devops#run-lint-tests-with-flake8).
3. Make the appropriate changes in your repository to install and run flake8.

As hint, you should install and run flake8 similarly to pytest.

**IMPORTANT!!!** Create a new script step, DO NOT combine with the pytest step. Otherwise, it will fail only when the last command of the step fails, independently of the rest.

1. Before flake8 is run in the pipeline, open the file app/app.py in VScode and fix some syntactical errors that have been included.
2. Run the following to check that all errors were corrected:

flake8 app tests

1. Commit the changes into the branch ‘feature/flake8’ and push it into Azure Devops. Wait until the pipeline is finished and check that flake8 run without errors.
2. In the same way, add two additional steps to include:
   1. **isort** which is to sort imports alphabetically, and automatically separated into sections and by type. See [here](https://github.com/pycqa/isort/) more details.

To save you some time, the command to run isort is:

isort -v app tests

* 1. **black** which is a Python code formatter. See [here](https://github.com/psf/black) more details.

To save you some time, the command to run black is:

black app tests

1. If the modifications are satisfactory, create a new pull request to the ‘dev’ branch and check that a new pipeline was triggered. In this case, it should be deployed into the ‘dev’ namespace.

# Lab 7: Adding additional tests and features.

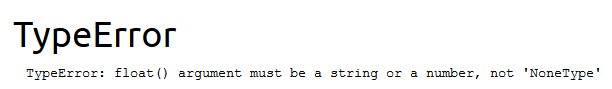
Did you realize what happens to the deployed app when square\_number is called and a string or an empty value is given instead of a number?

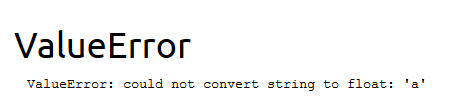
For that, open your favorite browser and go to:

http://<EXTERNAL\_IP>:5000/square\_number?number=a

http://<EXTERNAL\_IP>:5000/square\_number

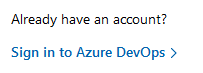
where <EXTERNAL\_IP> should be changed with the external IP given at the bottom of the deploy stage or going to Pipelines > Environments > YOUR\_NAME > YOUR\_NAME-dev > Services.





Your first goal in this lab is to create two new tests in the ‘test\_app.py’ to avoid the deployment of the lab until these errors have been fixed. For that:

1. Go to [Azure Devops login](https://azure.microsoft.com/en-us/services/devops/) and press ‘Sign in to Azure Devops’.



1. Create a branch called ‘feature/tests’ based on the ‘dev’ branch as in the previous Lab.
2. Based on the test already created in the ‘test\_app.py’ file, add two additional tests able to check what happens if no number is provided or a string is given. If no number is provided, the square function will return as output 'Empty input', whereas if a string is provided, the square function will return as output 'Not a numerical value'.
3. Commit the changes into the branch ‘feature/tests’. Is the pipeline stopped after test failures?

The second goal is that the function ‘square’ in the ‘app.py’ file can handle empty inputs or string inputs. For empty inputs, it should return 'Empty input', whereas for string inputs, it should return 'Not a numerical value'.

1. Modify the ‘square’ function in the ‘app.py’ accordingly. As hint, see above that two error types were generated, namely, TypeError and ValueError.
2. Commit the changes into the branch ‘feature/tests’. Wait until the pipeline is finished and check that all the tests are successfully passed.
3. If the modifications are satisfactory, create a new pull request to the ‘dev’ branch and check that a new pipeline was triggered. In this case, it should be deployed into the ‘dev’ namespace.
4. Check that the errors of the app were solved in the development environment by going to:

http://<EXTERNAL\_IP>:5000/square\_number?number=a

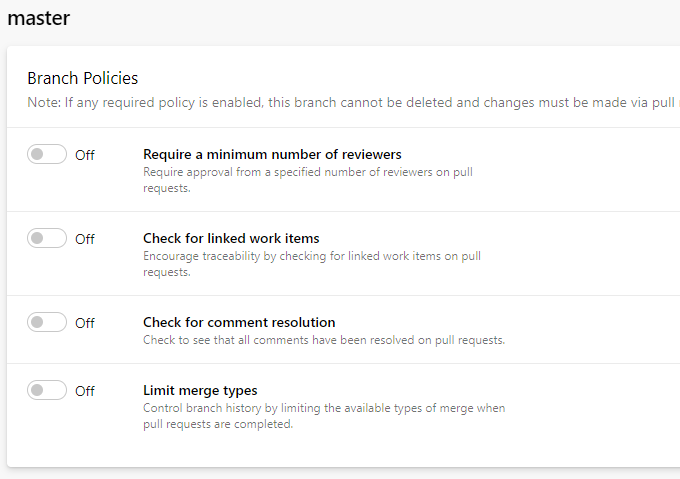
http://<EXTERNAL\_IP>:5000/square\_number

where <EXTERNAL\_IP> should be changed with the external IP given at the bottom of the deploy stage or going to Pipelines > Environments > YOUR\_NAME > YOUR\_NAME-dev > Services

1. If the modifications are satisfactory, create a new pull request to the ‘master’ branch:



1. Check that a new pipeline was triggered. In this case, it should be deployed into the ‘prod’ namespace.
2. In the meantime, it is always important to protect the master branch with Branch Policies, like require a minimum number of reviewers in the Pull request to be approved. To add these policies, go to Project Settings > Repositories and select your repository. Then go to policies > Branch Policies and select the master branch. There, you can see several options:



For this lab, you should not set any of them but be aware that this is a regular practice in real projects.

1. Check that the errors of the app were solved in the production environment by going to:

http://<EXTERNAL\_IP>:5000/square\_number?number=a

http://<EXTERNAL\_IP>:5000/square\_number

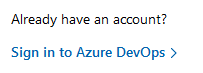
where <EXTERNAL\_IP> should be changed with the external IP given at the bottom of the deploy stage or going to Pipelines > Environments > YOUR\_NAME > YOUR\_NAME-prod > Services.

1. Solutions are available in the folder solutions.

# Optional lab: Adding container step to a pipeline using a yaml file.

In the previous labs, the code was tested in an environment built within an Azure VM which is used as an Agent pool. However, the application will be deployed as a container. In consequence, testing the application using the same image as it will be deployed seems more logical and closer to the final environment. After this lab you will test your app within a docker container.

1. Go to [Azure Devops login](https://azure.microsoft.com/en-us/services/devops/) and press ‘Sign in to Azure Devops’.



1. A connection with the Container registry should have been created. For that, go to  at the left bottom corner.
2. Press to  within the Pipelines section and check that a connection with the ACR created in step 1 is present:



1. Create a branch called ‘feature/container’ based on the ‘dev’ branch as in the previous Lab.
2. Delete all steps in the Build stage except ‘Build and push an image’.
3. Copy a new stage AFTER the ‘Build’ stage, it should look as follows:

stages:

- stage: Build

  displayName: Build stage

  jobs:

  - job: Build

    displayName: Build

    pool:

      vmImage: $(vmImageName)

    steps:

    - task: Docker@2

      displayName: Build and push an image to container registry

      inputs:

        command: buildAndPush

        repository: $(imageRepository)

        dockerfile: $(dockerfilePath)

        containerRegistry: $(dockerRegistryServiceConnection)

        tags: |

          $(tag)

    - upload: manifests

      artifact: manifests

- stage: Test

  displayName: Docker test

  dependsOn: Build

  jobs:

  - job: Test

    displayName: Test

    pool:

      vmImage: $(vmImageName)

    container:

      image: $(containerRegistry)/$(imageRepository):$(tag)

      endpoint: 'itvitaecontainerregistry'

    steps:

    - script: |

        pytest --junitxml=junit/test-results.xml --cov=. --cov-report=xml

    - script: |

        flake8 app tests

    - script: |

        isort -v app tests

    - script: |

        black -v app tests

    - task: PublishTestResults@2

      inputs:

        testResultsFormat: 'JUnit'

        testResultsFiles: 'junit/test-\*.xml'

        searchFolder: '$(Build.SourcesDirectory)'

    - task: PublishCodeCoverageResults@1

      inputs:

        codeCoverageTool: Cobertura

        summaryFileLocation: '$(Build.SourcesDirectory)/\*\*/coverage.xml'

1. Deploy stage should be carried out after the Test stage, for that, change the ‘Depeds on’ value to Test like this:

- stage: Deploy

  displayName: Deploy stage

**dependsOn: Test**

1. Press  and commit the changes.
2. Wait until the pipeline is finished. In the meantime, a few notes about yaml snippet of step 10:
   1. Since this stage should be run after the building process of the docker image, we added dependsOn: Build.
   2. A detailed description about container jobs can be found [here](https://docs.microsoft.com/en-us/azure/devops/pipelines/process/container-phases?view=azure-devops). Take a look to the requirements that are needed.
   3. The syntax is straightforward, just adding:

container:

      image: $(containerRegistry)/$(imageRepository):$(tag)

      endpoint: 'NAME\_OF\_THE\_REGISTRY'

implies that all commands under ‘steps’ will be run INSIDE the container.

* 1. The dotnet-sdk is needed to generate Code Coverage report using a docker step, which was included in the Dockerfile.

1. Check that test results are also available, this time they were executed inside a container.
2. If the modifications are satisfactory, they should be merged with the ‘dev’ branch. For that, go to Repos > Pull request and set a new pull request as follows:



1. Check the modifications and proceed adding a title like ‘added container test’. In a real situation, Reviewers might be added. For shake of simplicity, they are not needed in this exercise. Just press  and then into 
2. Check that a pipeline was triggered. In this case, it should be deployed into the ‘dev’ namespace.
3. Finally, create a pull request to merge also with the master branch.
4. Solutions are available in the folder solutions-docker.

IMPORTANT!!! As required clean up step, delete or stop the AKS cluster created in the Lab1 when finished.

For that, go to the Azure Portal > Resource group > AKS cluster and click ‘delete’.