

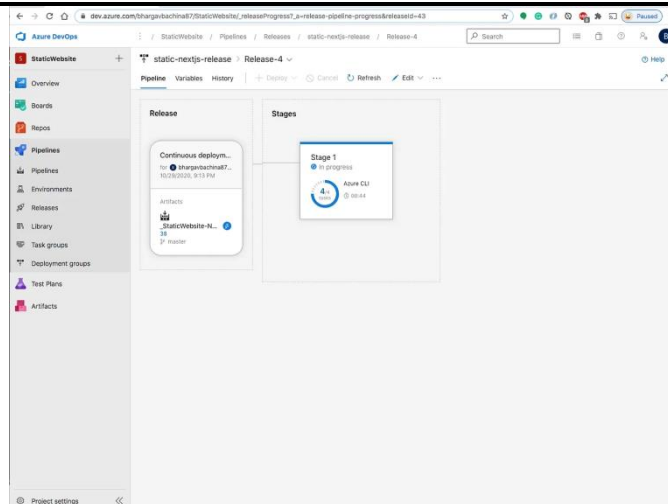


## Implementing CI/CD Pipelines for Python Azure Functions with Azure DevOps

Bhargav Bachina

**Abstract** Azure Functions provide a seamless solution for deploying small code segments in the cloud, supporting various programming languages. This paper explores deployment strategies, focusing on Azure DevOps for function app deployment. Additionally, it demonstrates creating a Python REST API using Azure Functions. Key sections include an introduction to Azure Functions and an example project showcasing Azure DevOps deployment. Through this exploration, developers gain insights into leveraging serverless computing for efficient application deployment and development.

**Keywords** Azure, Cloud Computing, Programming, Python, DevOps



AN AZURE Function is a simple way of running small pieces of code in the cloud. You don't have to worry about the infrastructure required to host that code. You can write the Function in C#, Java, JavaScript, PowerShell, Python, or any of the languages that are listed in the Supported languages in the Azure Functions (<https://docs.microsoft.com/en-us/azure/azure-functions/supported-languages>) article.

There are a lot of deployment strategies when you deploy your Azure functions to production and your deployment strategy entirely depends on your application architecture and the DevOps tools you are using. For example, If you are using Jenkins you can build the build pipeline and deploy the pipeline in Jenkins 2.

Using Azure DevOps, you have a ready-made template to deploy your function app. In this post, we will see how we can deploy an Azure function app using Azure DevOps. In this post, we will go through how to write Python REST API using Azure Functions.

- Introduction
- Example Project

- Prerequisites
- Build Pipeline
- Release Pipeline
- Demo
- Summary
- Conclusion

## 1. Introduction

The function app contains individual functions in which your function code resides. Each function app contains either one or more than one function serving each endpoint. When the function app's Authorization level is anonymous you can access the function app endpoints directly from the internet. We can directly deploy our Azure functions to function app from VSCode with the Azure Functions extension (<https://marketplace.visualstudio.com/items?itemName=ms-azuretools.vscode-azurefunctions>) for Visual Studio Code. But, when you are working in a team, deploying directly from the local machine is not a feasible solution. We need to build a pipeline in which the entire team can collaborate.

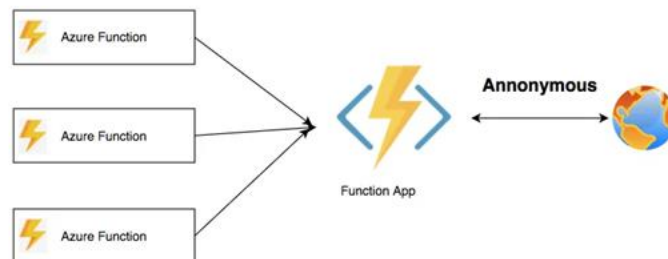


Figure 1: Function App Serving Directly

We must build two pipelines to deploy this application using Azure DevOps.

### A. Build Pipeline

This pipeline takes the code from the Azure Repos or any git source and goes through a series of actions such as install, test, build and finally, generate the artifacts ready for the deployment.



Figure 2: Build Pipeline

### B. Release Pipeline

This pipeline takes the artifact and deploys it into the function app. You can have pre-deployment conditions such as approvals, manual only, etc. for the release.



Figure 3: Release Pipeline

## 2. Example Project

This is a simple Python REST API where you can add tasks, delete tasks, update tasks, and get tasks. Here is an example project you can clone and run on your local machine.

```
// clone the project git clone https://github.com/bbachi/serverless-python-restapi-azure.git
// With Azure Functions extension installed func start
```



### 3. Testing

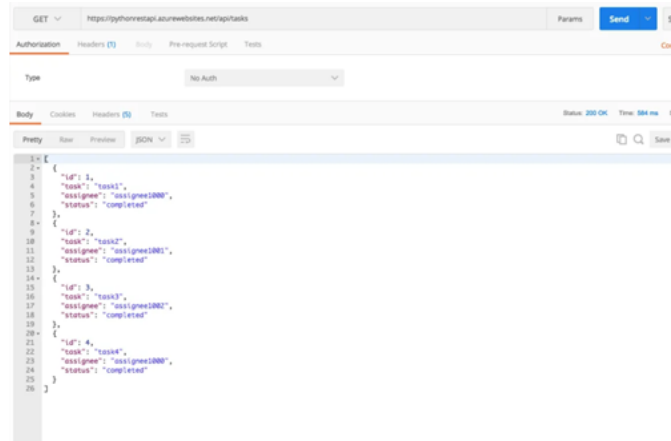


Figure 4: Testing the API

### 4. Prerequisites

You need to know a lot of things as prerequisites if you want to write a serverless Python REST API. First, you need to create two accounts: a Github account to store the source code and Microsoft Account to deploy that code using Function App Service. Let's create these accounts by following the below links. You can start both for free.

- Github Account (<https://github.com/>)
- Microsoft Azure Account (<https://azure.microsoft.com/en-us/>)
- Login into Azure DevOps (<https://azure.microsoft.com/en-us/services/devops/>)

All the API code is written in NodeJS Azure functions. You need to be familiar with the following.

- Azure Functions (<https://azure.microsoft.com/en-us/services/functions/>)
- Azure Functions extension for Visual Studio Code (<https://marketplace.visualstudio.com/items?itemName=ms-azuretools.vscode-azurefunctions>)
- Create a Subscription
- Create a function app

#### A. Create a Subscription

You should have a Microsoft Azure Account. You can get a free account for one year. You should see the below screen after you login.

- **Azure Account** (<https://azure.microsoft.com/en-us/free/>)

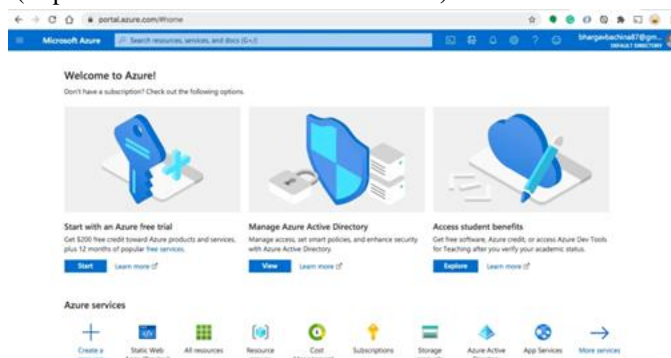


Figure 5: Azure Home Screen

You need to create a subscription for your account. The most common is Pay As You Go subscription.



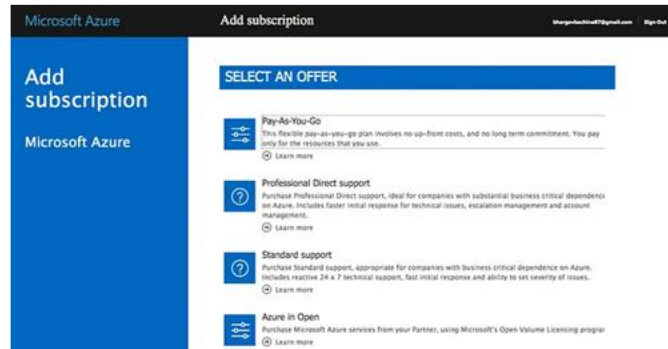


Figure 6: Subscription Offers

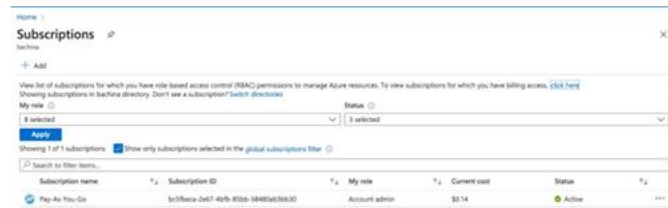


Figure 7: Pay-As-You-Go Subscription

You need a subscription to be associated with your tenant so that all the cost is billed to this subscription.

**B. Create a function app**

Login into the portal and select a function app resource to create and give required fields such as app name, storage account, and resource group, runtime stack, etc.

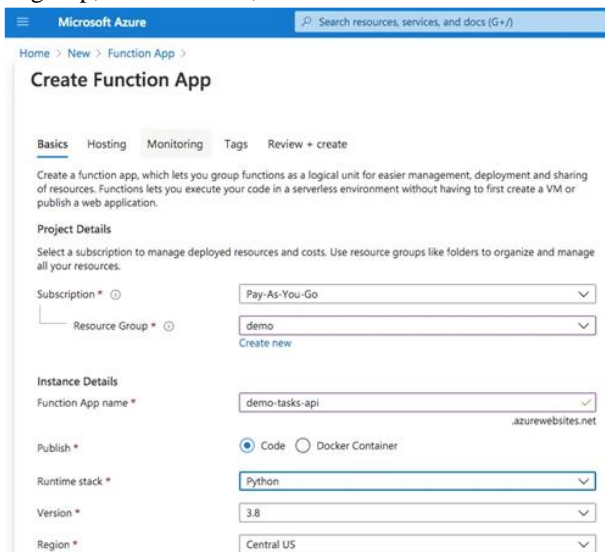


Figure 8 Creating a function app

Your deployment should be complete after some time.

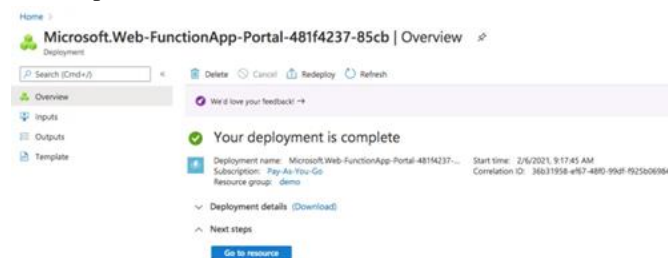


Figure 9: Deployment Completed



Here is the function app created and you can access the APIs with that URL if its authorization level is anonymous.

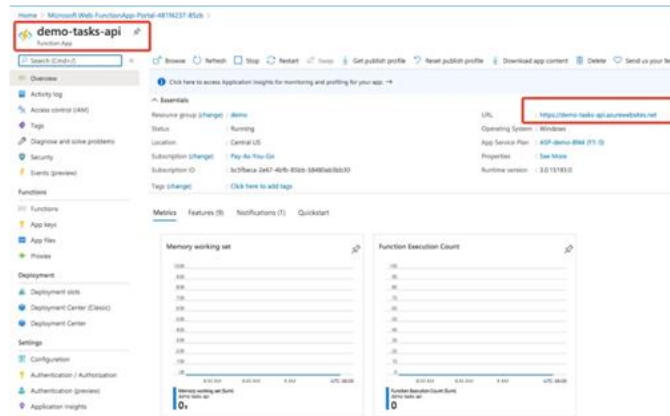


Figure 10: Function app created

## 5. Build Pipeline

We need to build pipelines as part of continuous integration first. The way this pipeline works is that the moment you check-in code into GitHub or Azure Repos, this pipeline builds the project, test it, make the built artifact ready for deployment. Let's follow all the steps to build this pipeline.

1. Create a Project in Azure DevOps
2. Create a Repo and Put your code in Azure Repos
3. Create a pipeline that takes it from the source repository.
4. Install all dependencies.
5. Run the tests.
6. Build the code.
7. Copy files from source for staging.
8. Archive all the copied files.
9. Finally, publish the artifact.
10. Enable Continuous Integration with triggers.

Let's create a project in your Azure DevOps account. I named it **Tasks API** with visibility of your choice. You can make it public or private.

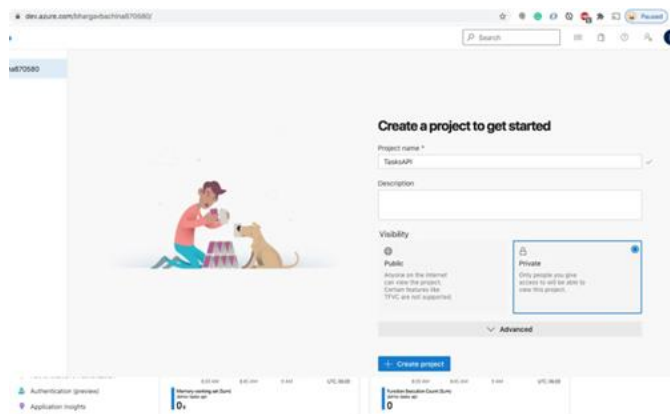


Figure 11: Creating Project

Once you have created this project you can see the dashboard which is empty right now.



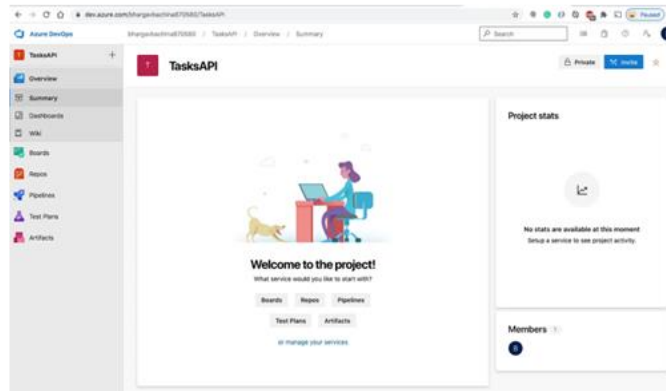


Figure 12: Project Created

**A. Put your code in Azure Repos**

It's time to create a repo and place all the code from the example project from above and push your code into this repo. I created a repo called **tasks-api** and pushed all the code into this repo.

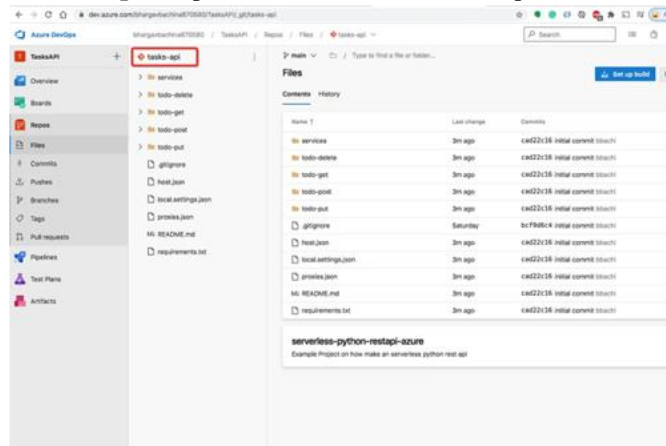


Figure 13: Azure Repos

You can generate the Git credentials when you click on the clone button on the top right corner. You need these credentials if you want to push the code into this repo later.

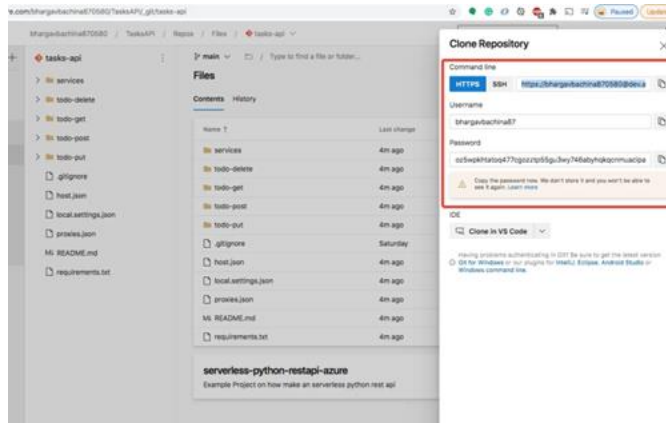


Figure 14: Git Credentials

**B. Create a Build Pipeline**

Let's start a pipeline by selecting a source and repository branch and by selecting a classic editor.



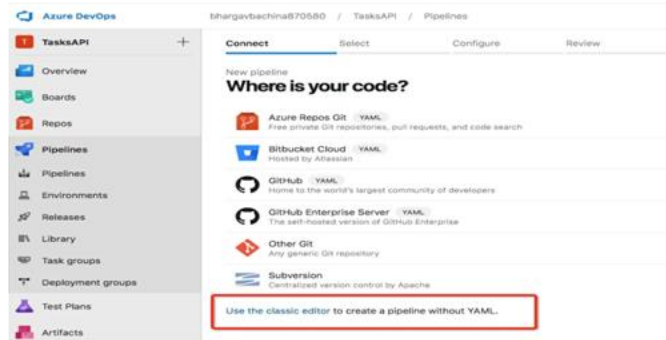


Figure 15: Select Classic Editor

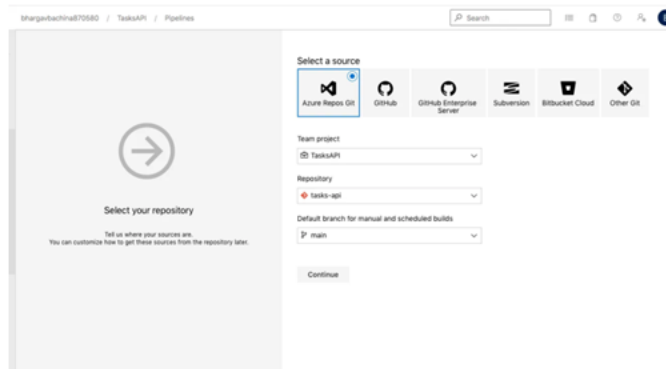


Figure 16: Select a source

On the next page, select the azure function for the Python template as Azure provides readymade templates for us.

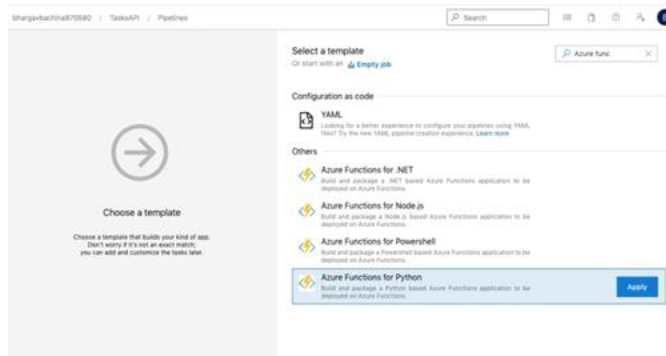


Figure 17: Selecting a template

Once you click on the button Apply you can see the next screen with all these tasks. You can change the pipeline name as you want. If you notice the tasks, it is using the Python version 3.6, install dependencies, turns build script, archive files, and publish the artifact.

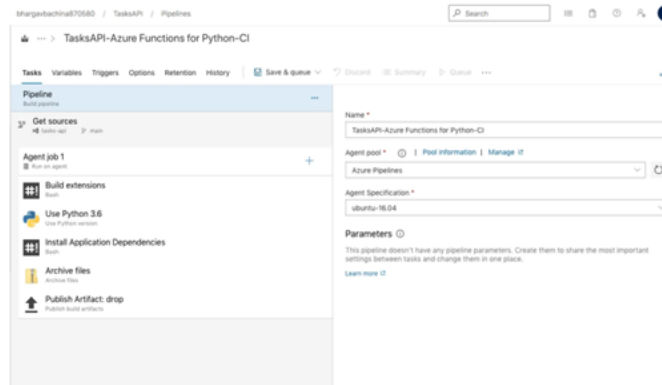


Figure 18: Template Applied

Click on the save and you can save this into any folder. Let’s define the trigger.

**C. Define the trigger**

Let’s define the trigger for this build. Edit the pipeline by clicking on the triggers and enable continuous integration. Any commit to the task-api repo main branch triggers this pipeline.

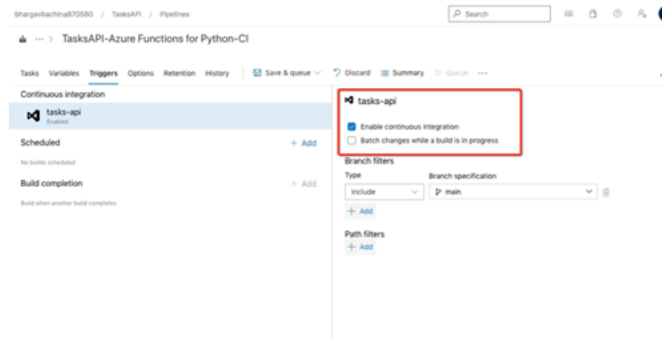


Figure 19: Enable Continuous Integration

Let’s build the pipeline manually this time by clicking on the Queue.

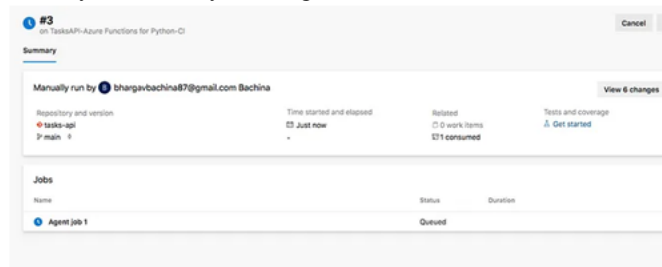


Figure 20: Job Running

Click on Agent Job 1 to see the results.



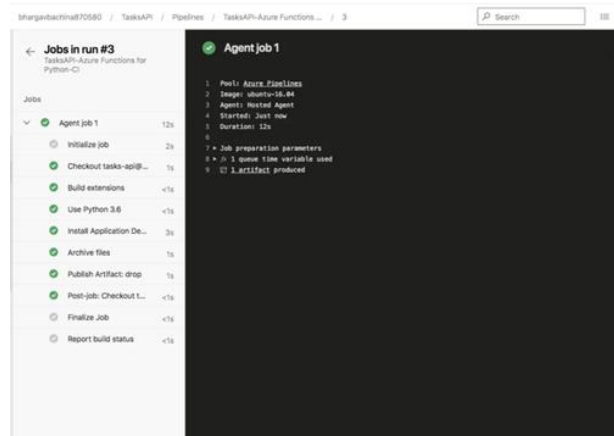


Figure 21: Job Details

Click on the artifact link in the logs to see the artifact details. This creates the Zip file.

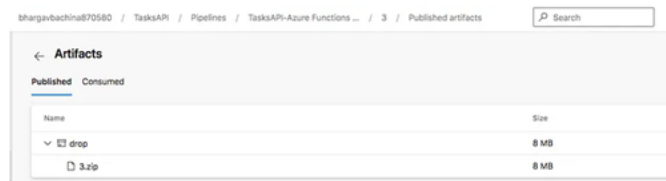


Figure 22: Zip Artifact

If you download the zip file and open you can see the Azure Functions project structure.

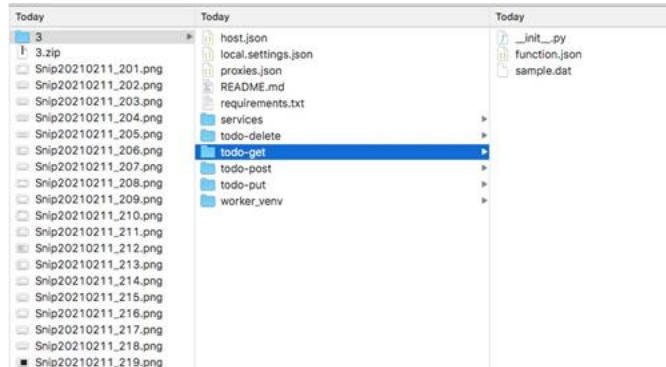


Figure 23: Zip file Extract

Here is the complete YAML for this build pipeline.

<https://gist.github.com/bbachi/82288061db154706c20ccb67cf1c90ab#file-template-yaml>

You can get the above YAML from this edit pipeline screen.

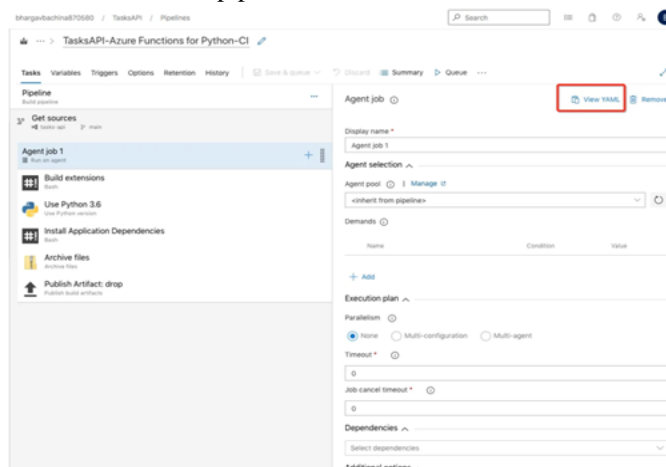


Figure 24: Getting YAML



## 6. Release Pipeline

We are done with the Continuous integration part and let's build a release pipeline for continuous delivery. Click on the releases and new release pipeline and follow these steps.

**A. Define the artifacts such as project, source, default version, etc**

**B. Define a Stage for the deployment.**

### A. Define the Artifact

The first step is to define the artifact so that it takes that artifact after the build pipeline is completed. Make sure you have a trigger placed.

When you click on the releases you see an empty page with a new pipeline button. Click on it to create a new release pipeline. Select a template called *Deploy a function app to Azure Functions*.

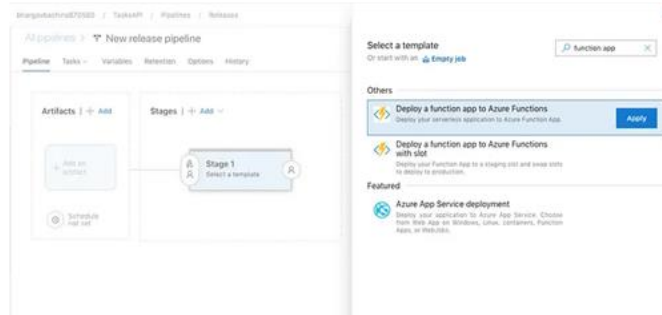


Figure 25: Selecting a pipeline

You can name the stage name. I named it Development.

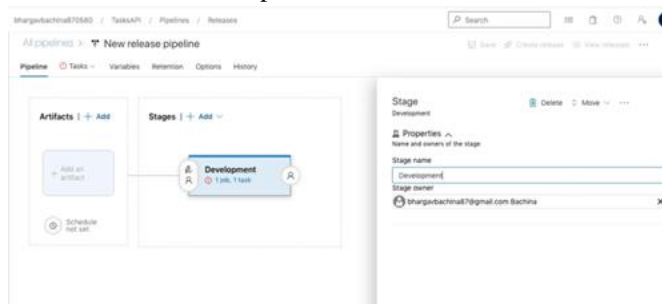


Figure 26: Stage Name

Click on the Job task section of the stage to define the tasks.

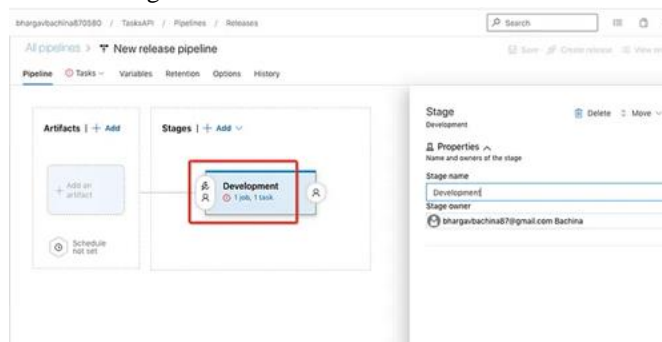


Figure 27: Job

### B. Define a Stage for the deployment.

To deploy the function app, you need a subscription and create a function app in the Azure portal as we discussed in the prerequisites section. We need to select the subscription and give the function app name that we created in the prerequisite section.



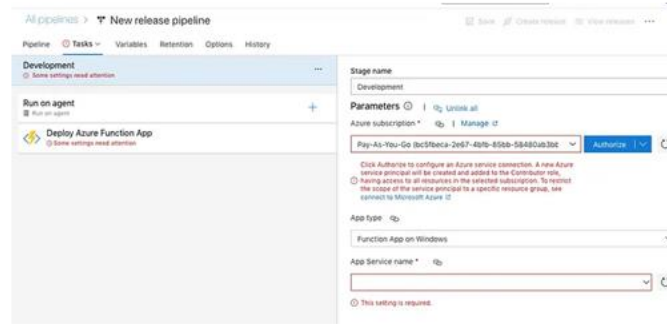


Figure 28: Subscription and App Service Name

Once it is Authorized this is how it looks like

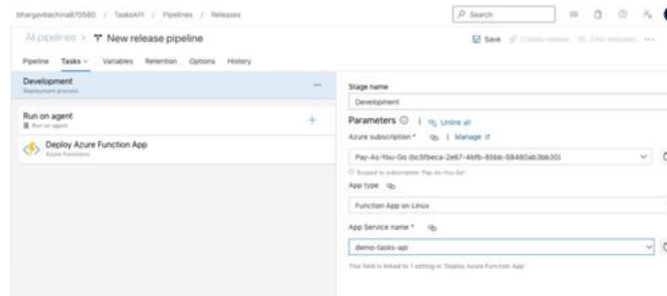


Figure 29: Release Pipeline

We need to add an artifact by selecting the source pipeline

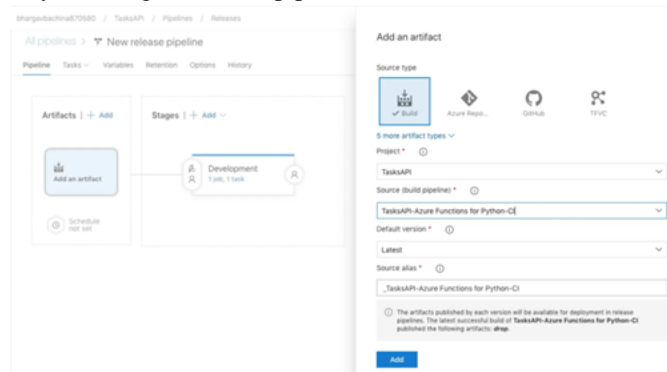


Figure 30: Adding an Artifact

Once it is added, you can have a trigger defined. Click on the flash icon

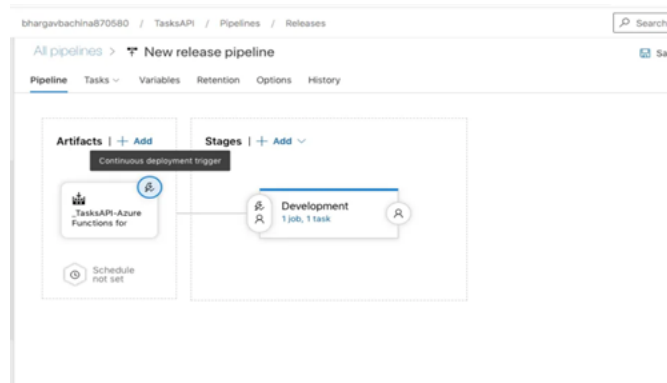


Figure 31: Enable Trigger

Once it is enabled, don't forget to save the pipeline.

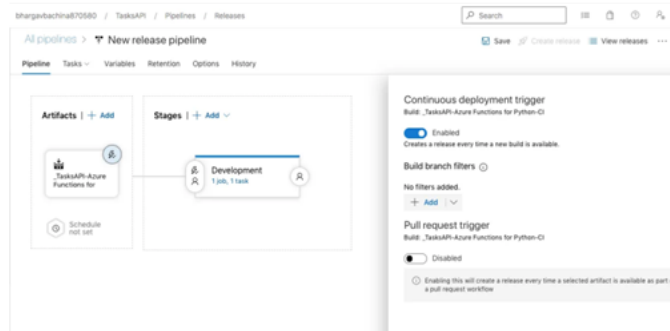


Figure 32: Continuous deployment trigger

**7. Demo**

It's time for the demo. As soon as you check-in the code to mine it triggers the build pipeline. Usually, you don't check in the code directly into main instead you create a pull request. I just used a main here for simplicity.

**A. Build Pipeline Demo**

I just added one more task to the array and push it to the main branch. As soon as we pushed the code, the build pipeline triggered.

[https://miro.medium.com/v2/resize:fit:1400/1\\*ORVkc6t1-GyvxA5LVmjW2g.gif](https://miro.medium.com/v2/resize:fit:1400/1*ORVkc6t1-GyvxA5LVmjW2g.gif)

**B. Release Pipeline Demo**

As soon as the building is completed and a new artifact will trigger the release pipeline.

[https://miro.medium.com/v2/resize:fit:1400/1\\*5xbT\\_9IE-nCaGtzAdWhJIA.gif](https://miro.medium.com/v2/resize:fit:1400/1*5xbT_9IE-nCaGtzAdWhJIA.gif)

Once it is succeeded, you can go to the function app in the portal and verify the functions.

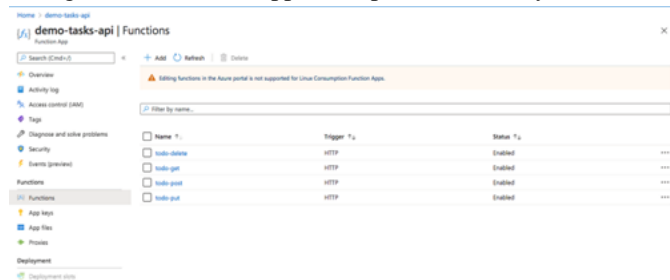


Figure 33: Functions Deployed

You can test the functions either directly in the portal or you can the function URL and hit it in the browser or in the postman. You can see the 5 tasks in the array response.

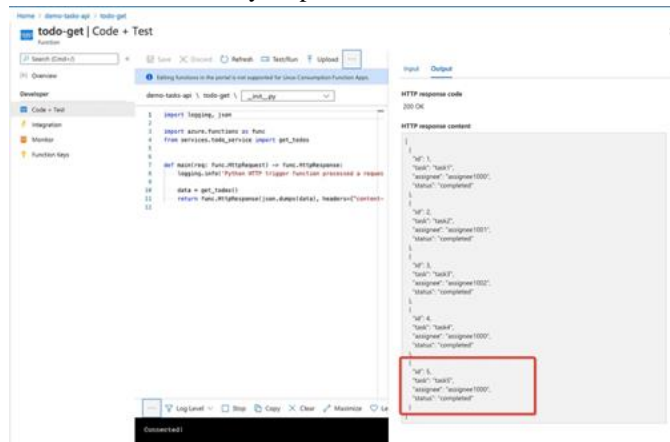
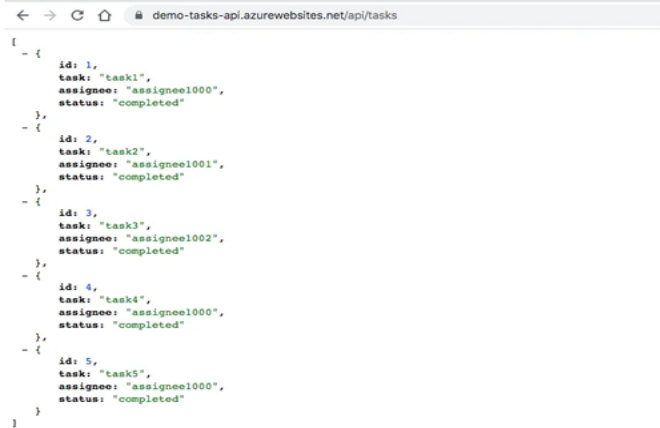


Figure 34: Testing the function

You can hit the below URL in the browser and see the response as below.



<https://demo-tasks-api.azurewebsites.net/api/tasks>



```
demo-tasks-api.azurewebsites.net/api/tasks
[
  - {
    id: 1,
    task: "task1",
    assignee: "assignee1000",
    status: "completed"
  },
  - {
    id: 2,
    task: "task2",
    assignee: "assignee1001",
    status: "completed"
  },
  - {
    id: 3,
    task: "task3",
    assignee: "assignee1002",
    status: "completed"
  },
  - {
    id: 4,
    task: "task4",
    assignee: "assignee1000",
    status: "completed"
  },
  - {
    id: 5,
    task: "task5",
    assignee: "assignee1000",
    status: "completed"
  }
]
```

Figure 35: Testing the function

## 8. Summary

- One way of building Python REST API is to use the Azure function app.
- There are a lot of deployment strategies when you deploy your Azure functions to production and your deployment strategy entirely depends on your application architecture and the DevOps tools you are using.
- Using Azure DevOps, you have a ready-made template to deploy your function app.
- We must build two pipelines to deploy this application using Azure DevOps: Build pipeline and Release pipeline.
- The build pipeline generates the artifact as soon as there is a commit in the source repository.
- The Release pipeline takes the artifact and releases it to the appropriate environment.
- Authentication should be made before you can use pipelines to access your Repos and upload files to the Azure subscription account.
- You can use task groups to put the common tasks in one place across the environments.
- You can use Azure key vaults to store the access keys for your storage account.
- You can make use of Pipeline Variables while creating tasks. It's a convenient way to get key bits of data into various parts of the pipeline.

## 9. Conclusion

In conclusion, this paper explored the process of building a Python REST API using Azure Functions, emphasizing deployment strategies with Azure DevOps. The discussion highlighted the significance of tailoring deployment strategies to match the application architecture and DevOps tooling. Leveraging Azure DevOps, developers can utilize pre-configured templates for function app deployment, streamlining the deployment process. Additionally, the paper outlined the necessity of creating two pipelines—Build and Release—to automate artifact generation and deployment across environments. Authentication measures were underscored as essential for accessing repositories and Azure subscription accounts. Furthermore, the use of task groups and Azure Key Vaults was recommended to enhance efficiency and security in deployment workflows. Lastly, the utilization of Pipeline Variables was highlighted as a practical approach to managing crucial data within the deployment pipeline, ensuring smooth and reliable deployment processes.

## References

- [1] Azure Cloud Documentation <https://azure.microsoft.com/en-us>
- [2] Azure DevOps Documentation <https://azure.microsoft.com/en-us/products/devops/boards/>
- [3] Azure Functions <https://azure.microsoft.com/en-us/products/functions>
- [4] Python Documentation <https://docs.python.org/3/>

