Practical Guide

Azure DevOps

Complete CI/CD Pipeline

Author

Mukesh Kumar

C#Corner

Disclaimer & Copyright

The information provided within this eBook is for general informational purposes only. While we try to keep the information up-to-date and correct, there are no representations or warranties, express or implied, about the completeness, accuracy, reliability, suitability or availability with respect to the information, products, services, or related graphics contained in this eBook for any purpose. Any use of this information is at your own risk

The methods described in this eBook are the author's personal thoughts. They are not intended to be a definitive set of instructions for this project. You may discover there are other methods and materials to accomplish the same end result.

Mukesh

Kumar Author

About The Author

Mukesh Kumar is a **Software Developer** and **Microsoft MVP** who has a Master's degree in Computer Science. He is also **C# Corner MVP, Blogger, Writer** and has more than seven years of extensive experience designing and developing enterprise-scale applications on Microsoft Technologies like C#, Asp.Net, Web API, Asp.NET Core, Microsoft Azure, Angular 4+, JavaScript, Node.JS, Python etc.

He is a passionate author on www.mukeshkumar.net and believes in the motto **"Think for the new."**

You can also follow him on Facebook, Twitter, LinkedIn and Google+.

Table of Contents

Chapter 1. About DevOps

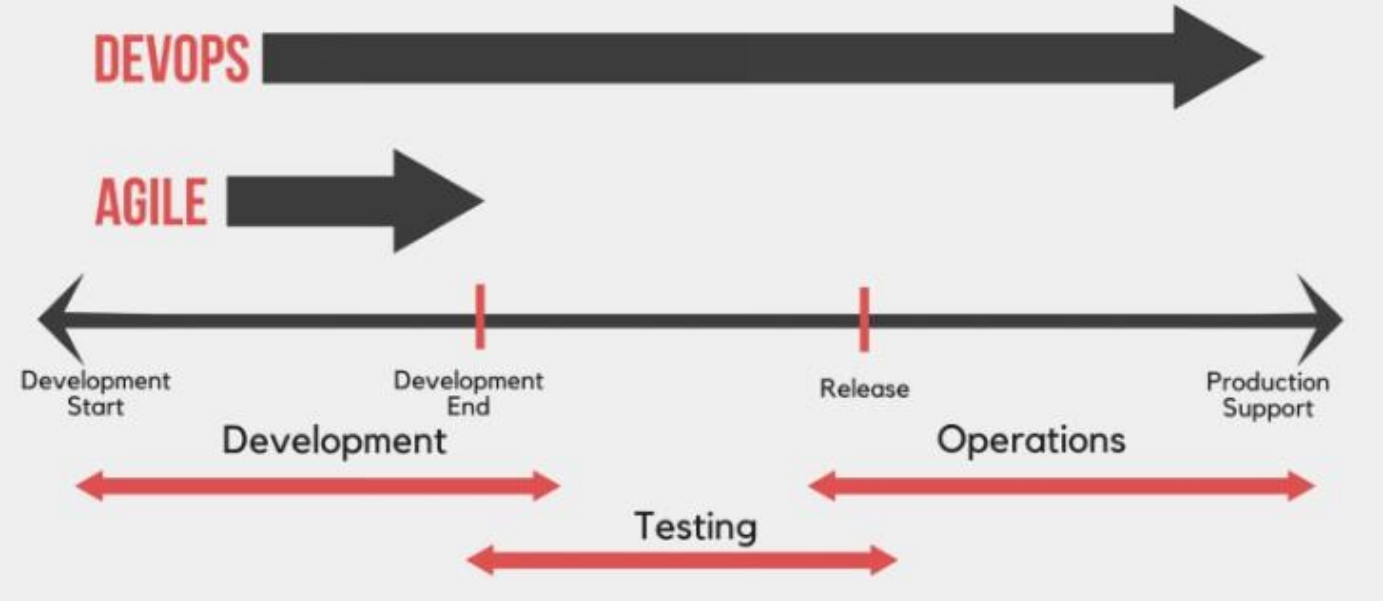
1.1 Definition

DevOps is a combination of Development and Operations. It means Dev + Ops = DevOps. It is a culture which automates systems and improves and accelerates delivery to the client in a repeated manner. It's basically a collaboration between the Development team and the Operations team for serving up a better quality application. It is a culture for continuous integration and continuous delivery where we make the automated build system as well as automated deployment system. In other words, DevOps is practice collaboration between Development and Operations from the planning of a project to deployment on production. It involves the complete SDLC life cycle as well as monitoring.

Understanding DevOps, you should consider the following points as well,

- DevOps means only combining the two teams of Development and Operations.
- It is not a separate team.
- It is not a product or tool.
- DevOps people do not hire from outside, they are internal team members who are working either in the development phase or in the operations phase.

It is a group of people, processes and tools. It basically brings two or more different teams, like development and operations, together with a well-defined process, using some great tools for automatic software delivery to the client. It is also a set of practices which are used by the DevOps teams to speed up quality delivery. There are different kinds of tools or sets of tools which are used in **Continuous Integration (CI) and Continuous Delivery (CD)** where it performs restoring the code, building the processes, executing the test cases, and deploying on the stage environment, etc.



1.2 Why DevOps

To understand why DevOps is required, let's first understand, what happens without DevOps.

As an IT consulting firm, while initiating a new project, we have two different kinds of teams. First is the Development team, which is involved completely in developing and testing, including writing code and unit test cases. The other team is the Operations team which is involved in operating and monitoring the product.

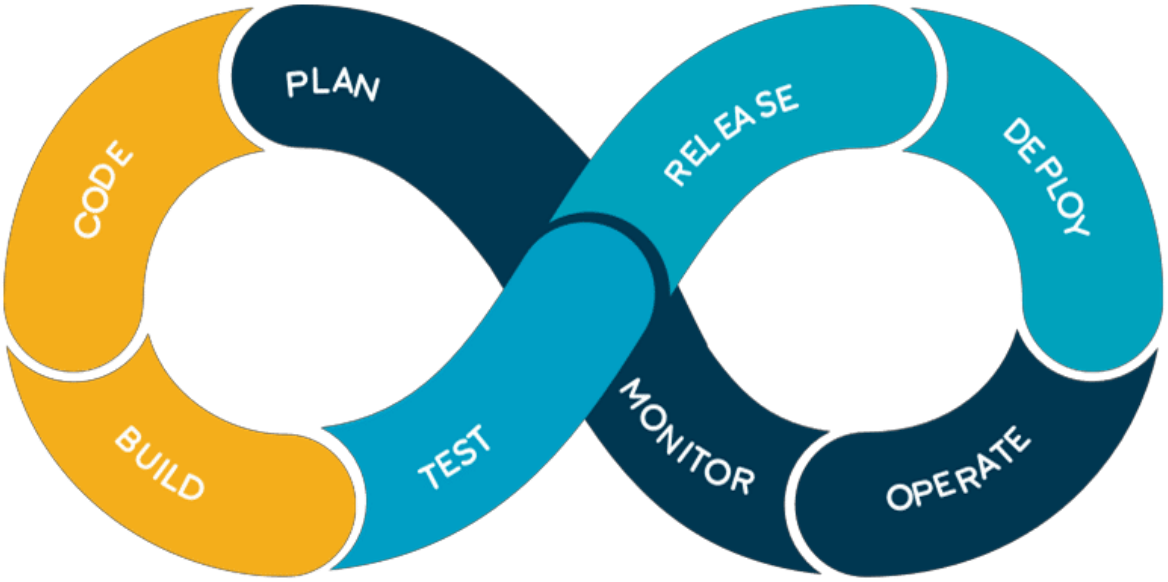
- Without DevOps, both teams **(Development and Operations)** work completely in an isolated manner.
- If DevOps is not there then the team spends most of the time in building the code and deploying on multiple environments.
- Each team waits for others to be done. This means, if development is going on then the testing team waits for the deployment of the code (Artifact) on the QA environment and in the same manner the operations team waits for the deployment on the Production environment. So, due to this, a large amount of time gets wasted.
- As a human being, we make mistakes. Building the code and deploying on specific environments in a manual fashion can increase the issues and resolving it will take a lot of time. Rather than doing it manually, we can make it automated using DevOps.

So, the above points show that we face lots of human issues and system issues if we are not following DevOps. Now, let's see how we can make it more systematic using DevOps and how DevOps helps us to achieve the same tasks in less time without any errors.

- DevOps increases the higher success rate of new releases without any error.
- It helps us to simplify the whole development and deployment process.
- Automates the manual process like build process, release process, etc.
- Automates executing the test cases.
- Configures the continuous delivery and continuous deployment in the release cycle.
- Live monitoring.
- It also helps in team collaboration.
- Reduces the failures and rollbacks
- Provides continuous improvement

1.3 DevOps Lifecycle

DevOps is a culture where the Development and Operations teams get involved. The development phase has its own lifecycle and the Operations phase has its own as well. If we combine both lifecycles, we get the lifecycle of DevOps. If an organization is not following or considering some of the points from the DevOps lifecycle then we can say, they are not following a DevOps culture. From planning to deployment, we have several stages which are very important and we cannot skip any of them. So, let's understand the DevOps lifecycle.



PLAN

It is the first stage of any new project. Here, we plan for a new project from requirement gathering from the customer and planning, to delivering the final project to the customer. We find out,

- What the requirements are.
- What types of product we have to create.
- What the timelines are for different sprints and the final product.
- What technologies we will use.
- What tools we will use.
- How many team members will be available for this project.
- What process we are going to follow, like Agile.

CODE

In this stage, we do the coding for creating the product with actual functionality as discussed with the customer. We use different types of methodologies for achieving the goal, like Agile methodology. Here we group the tasks in the sprint and their time estimation as well. Sprints are basically for 2-3 weeks. Unit Test cases are also to be a part of the coding.

BUILD

In this stage, we build the code. Code building happens two times; first when a developer is writing the code, then he/she has to build the code every time in their own local system to see the functionality. The second time, when a team member checks in the code in the source control repository, then it automates the build for the code and makes the artifact for deployment.

TEST

Testing is the heart of any development process. We write the Unit Test cases along with code. In DevOps, test cases auto execute and validate the build process.

RELEASE

In this stage, it collects the build artifact which can deploy further.

DEPLOY

Here, we start the deployment on the respective stages which are configured in the Release Pipeline. Actually after testing and validating the build artifact, it auto starts the deployment on the respective environment using a continuous delivery process. But before deploying it to the production environment, it asks for approval which can be done manually.

We can also automate the whole deployment process using continuous deployment. This is basically used when we have small changes which can be deployed on production as well without any approvals.

OPERATE & MONITOR

After successful deployment on the production environment, we have to operate the whole system and monitor the application. This monitoring is not only the performance but also the functionality.

1.4 Prerequisites

In order to do this practical demonstration, we will require a few tool and some accounts.

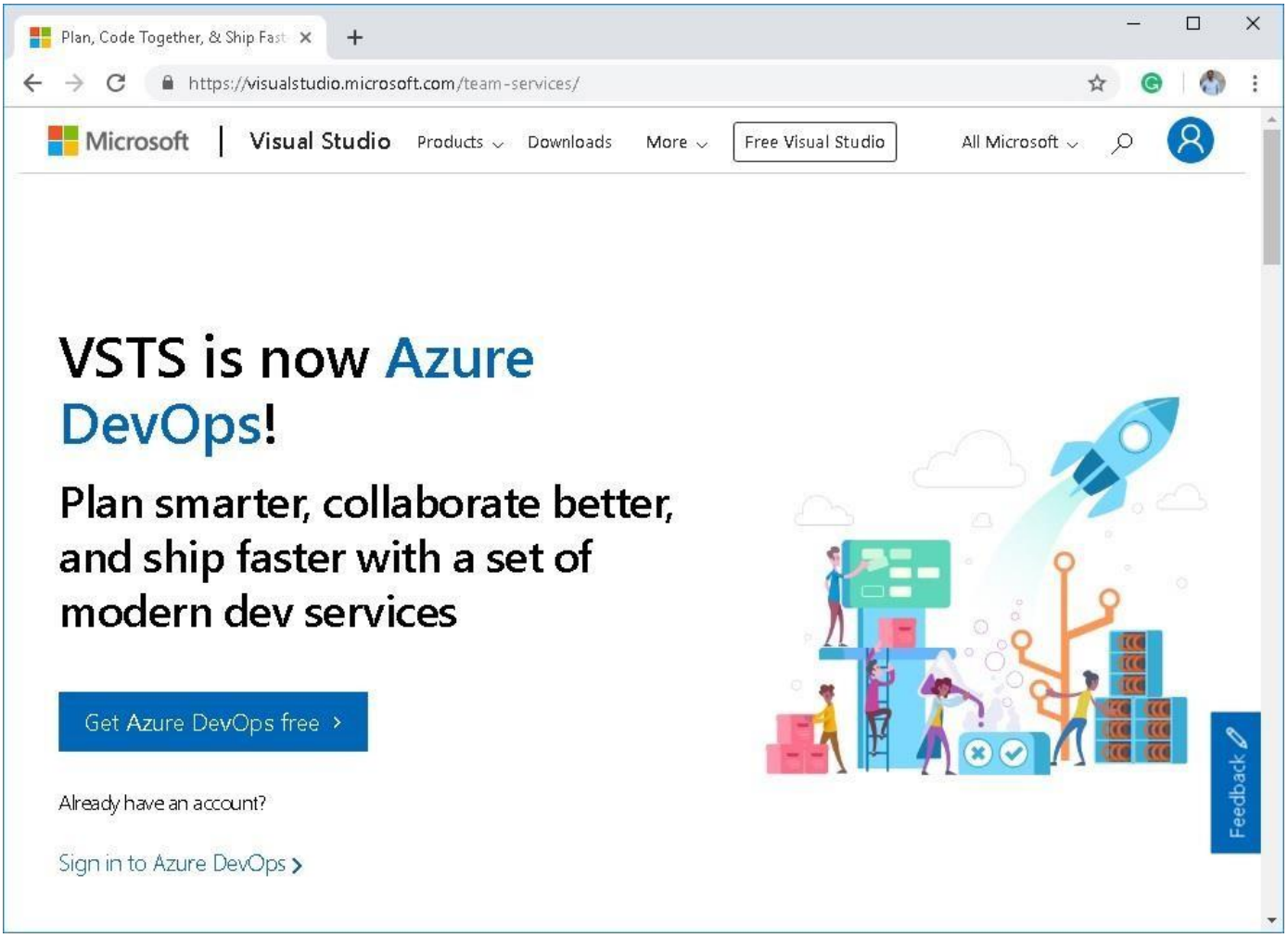
Visual Studio 2017 or Higher Version

It is a world class IDE which supports more than 40 programming languages. We will create the sample application along with a test project using Visual Studio 2017. So, before starting the demonstration, we will require Visual Studio 2017. If you would like to download Visual Studio 2017, we can download it from HERE.

Alternatively,

1. We can also use the Visual Studio Code, which is totally open source and can be download from HERE.
2. We can also create the Virtual Machine on the cloud (Cloud Account is required) and install Visual Studio 2017.

Azure DevOps



We need an Azure DevOps account. If we have an account with Azure DevOps that's fine; otherwise we can sign up from HERE. Earlier it was known as VSTS (Visual Studio Team Service). We can also get the free trial Azure DevOps account. Just click on the button 'Get Azure DevOps Free' as follows

Chapter 2.CI/CD Pipeline

Nowadays, the delivery process in software development is rapid. With the help of several tools, we try to customize our delivery process. If the delivery process is smooth then we can deliver a high quality of product to the customer within the timeline. Actually, customers want small changes or functionalities to be added in the minimum amount of time. To enable a faster delivery process, we take the help of some of the mechanisms like Agile and various tools and people, and this is called DevOps.

Let's see what the old processes in development are and what happens if there are some issues.

1. Get the requirements from the business.
2. Make a plan for converting the requirements into the actual functionality.
3. Developers do the code and check in itto a repository like TFS, GitHub etc.
4. After all code checks in to the repository, a developer executes the build process.
5. Run the test cases against the code.
6. If everything is fine, code gets deployed on the DEV server and then on QA.
7. If QA confirms it's okay then deployment begins on PROD.

Above is the general development workflows which are used mostly in small organizations. But is this process correct? Let's understand the problem with the above development and delivery process.

1. Every time a developer checks in the code into a repository, he/she doesn't know about build. Is build created successfully or not?
2. Developers should wait for other developers to do the checking in of the code and building of the code before deploying to DEV/QA server.
3. If a developer has completed his/her work and has checked in the code, they have to wait for other developers to check in the code as well.
4. If something is wrong with any part of the code then the build cycle will stop and have to wait until the issue is resolved.
5. As a human being, we make a lot of mistakes. We can also make a mistake while doing manual deployment.
6. There are chances to get more issues from development to deployment.

As we can see with the above points, there are several issues while using the old deployment process, which also needs more time for manual deployment, more resources for completing manual deployment, and obviously more money. We can resolve all of the above issues if we implement **Azure DevOps**. **In Azure DevOps**, we have **Continuous Integration (CI)** and **Continuous Delivery (CD)**. These help us to make the whole process automated. So, let's understand what these are.

Continuous Integration (CI)

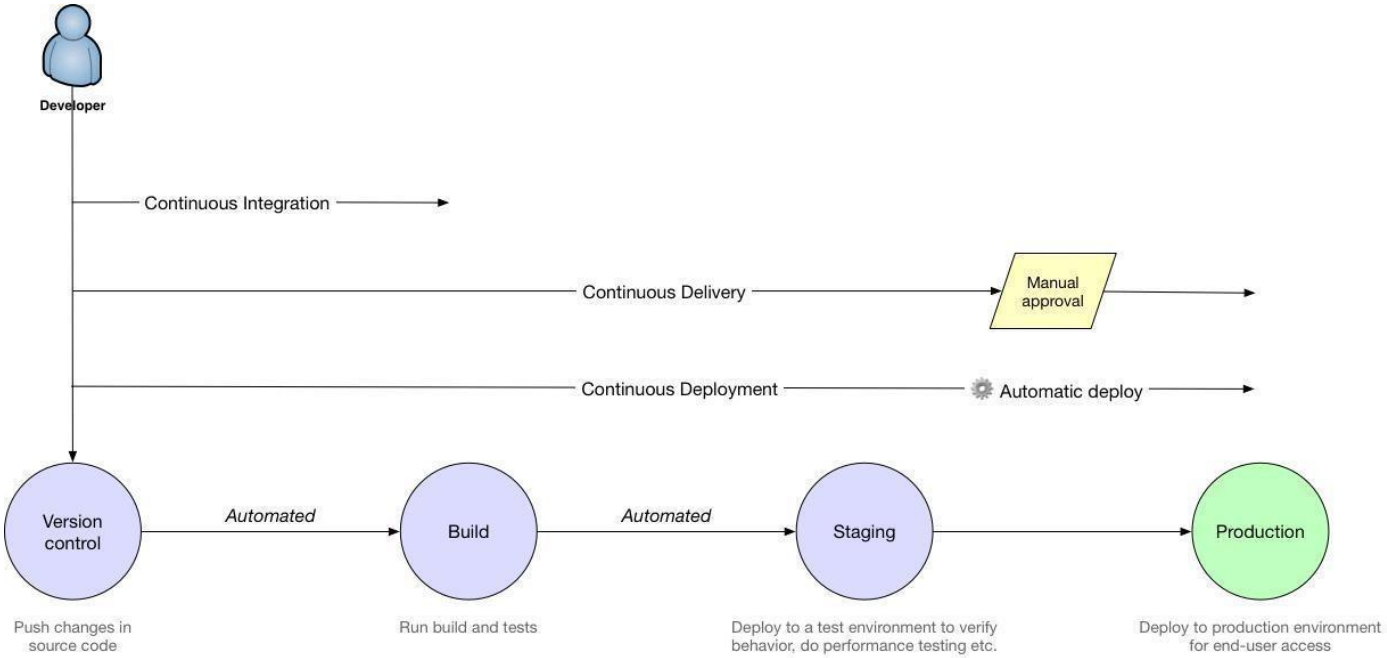
It is an automated build process which starts automatically when a developer checks in the code into the respective branch. Once the code check in the process is done, the continuous Integration process starts. It fetches the latest code from the respective branch and restores the required packages and starts building automatically. After the build process, it auto starts executing the Unit Test Cases and at the end, the build artifact is ready. This build artifact further will be used for deployment in the release process.

Continuous Delivery (CD)

It is the next process after Continuous Integration. Once the build artifact is ready for deployment then the release process gets started. As per the Azure DevOps Release Pipeline configuration, it starts deploying the build artifact on a different stage server (Environments) automatically. But in the Continuous Delivery, the deployment on the production goes manual. Manual does not mean here that we will deploy the build artifact manually, but that we have to approve it before deploying it on the PROD.

Continuous Deployment

It is the combination of **CI + CD and deployment on the production without any approval**. It means, in Continuous Deployment, everything goes automatically.



As per the above image, we can see that once a developer checks the code into the Version Control, the Azure DevOps Pipeline starts. From getting the code from Version Control, restoring it, and restoring packages from NuGet, Maven etc., to building the code and executing the unit tests cases, these are all part of the **Continuous Integration (CI)**.

Including Continuous Integration, if the build artifact gets auto-deployed on any staging server like DEV or

QA and deploys on the Production after manual approval, this is called **Continuous Delivery (CD)**. If manual approval is converted into automatic deployment then it's called **Continuous Deployment**. So, basically Continuous Deployment is the same as Continuous Delivery but without any manual approval. All goes in auto approval mode.

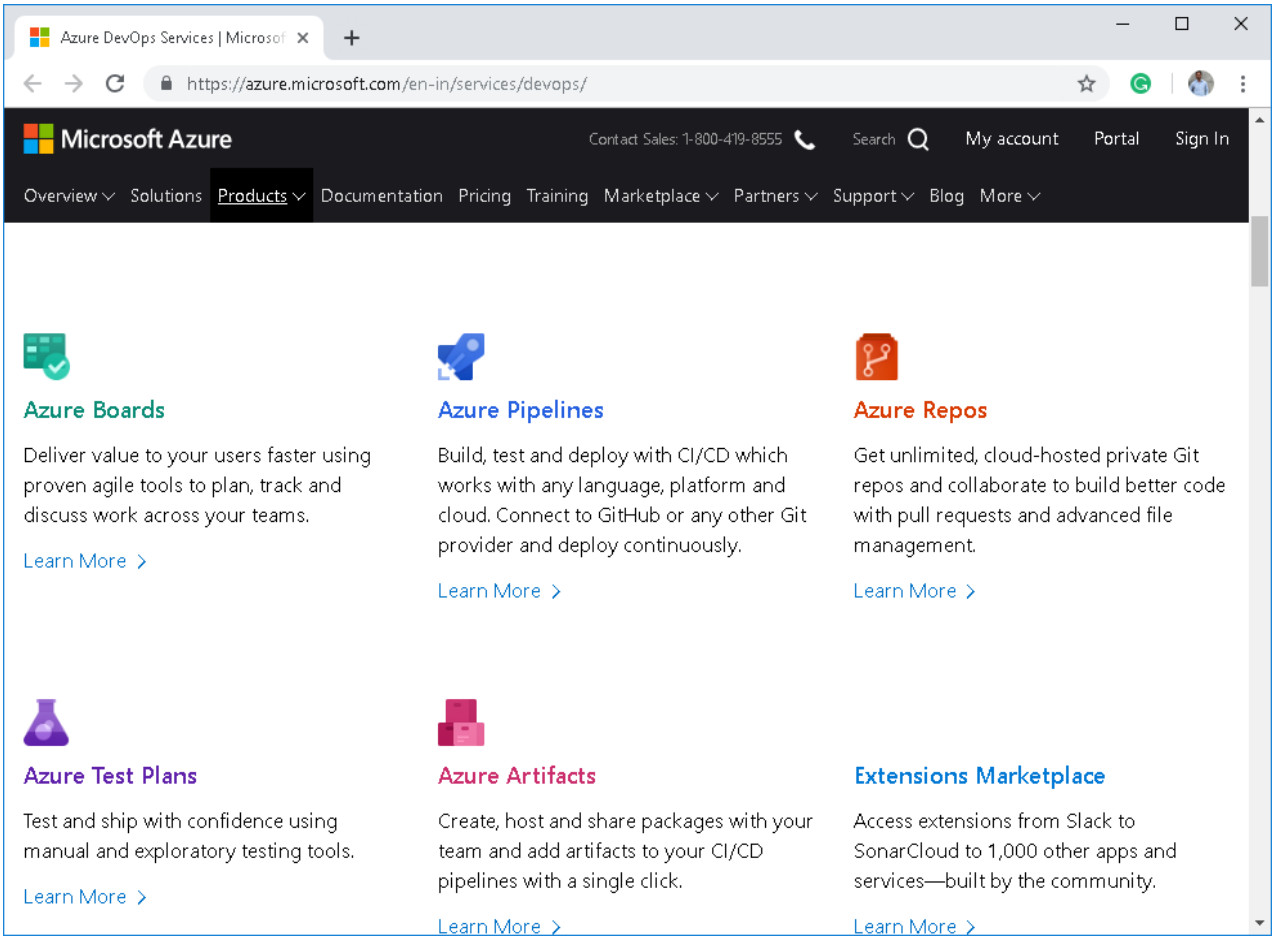
Chapter 3.Why Azure DevOps

DevOps is a group of people, processes and tools which enable and automate continuous delivery. Azure DevOps, formerly known as Visual Studio Team Service (VSTS), provides the repository management, project management, Build/Release Pipeline etc. Azure DevOps is free for a small project which has up to five users. But we can also use the paid version of it.

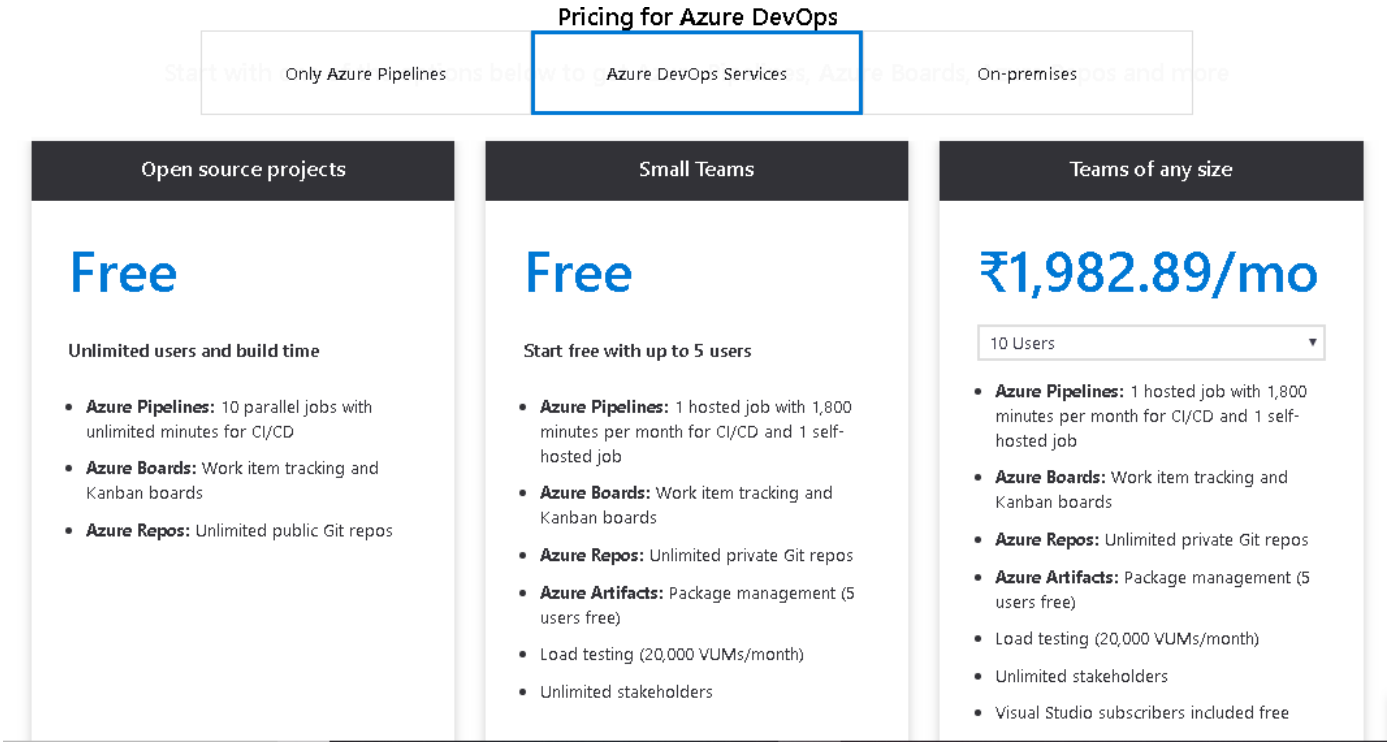
Azure DevOps provides unlimited cloud-hosted Git repos for a small or big project. Here developers can pull the code from Repos or push the code into Repos. It is basically a complete file management system.

While building the source code, it requires lots of third-party packages. Using Azure Artifacts, it enables the **NPM**, **MAVEN** or **NuGet** packages to be available for private or public source code.

The most important feature of Azure DevOps is the Azure pipeline. It enables the automated build and release pipeline. Azure Pipeline helps us to achieve Continuous Integration and Continuous Delivery for the project. Know more updates about Azure DevOps HERE.



Microsoft Azure DevOps is most popular and widely-used throughout the world because **it is free for Open Source Projects and Small Teams Projects.** We can use lots of Azure DevOps features with the free version like Azure Pipeline, Azure Boards, Azure Repos, and Azure Artifacts. It means, we don't need to go for the paid version of Azure DevOps, if and only if we want to use those features which are in free versions.

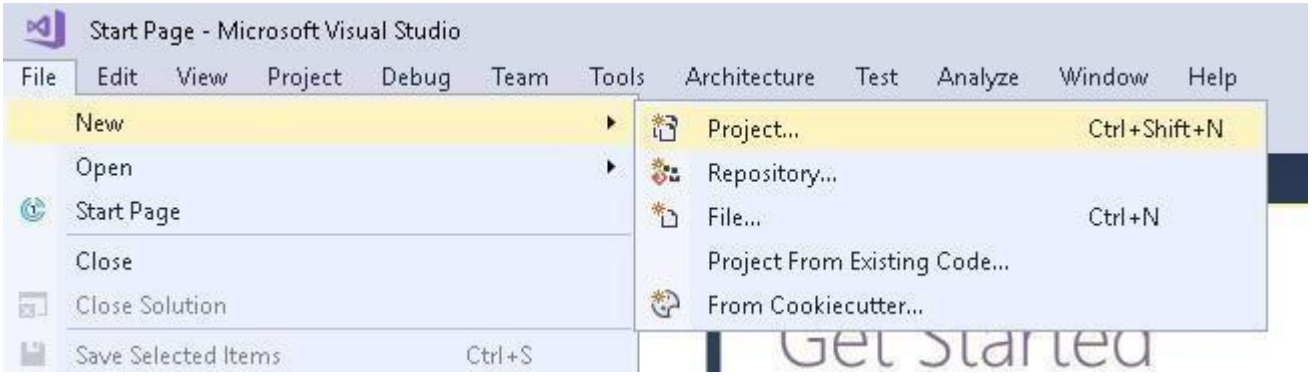


Chapter 4.DevOps Project Setup

In this session, we will create a project in Visual Studio 2017 or higher version which will be used for this demonstration. Apart from the main project, it will also include a testing project, where all required Unit Test Cases will write. For this demonstration, we are using Visual Studio 2017 but anyone can use a higher version of Visual Studio for creating the Asp.NET Core. So, let's start by creating the Asp.Net Core project along with xUnit testing project for DevOps Demo.

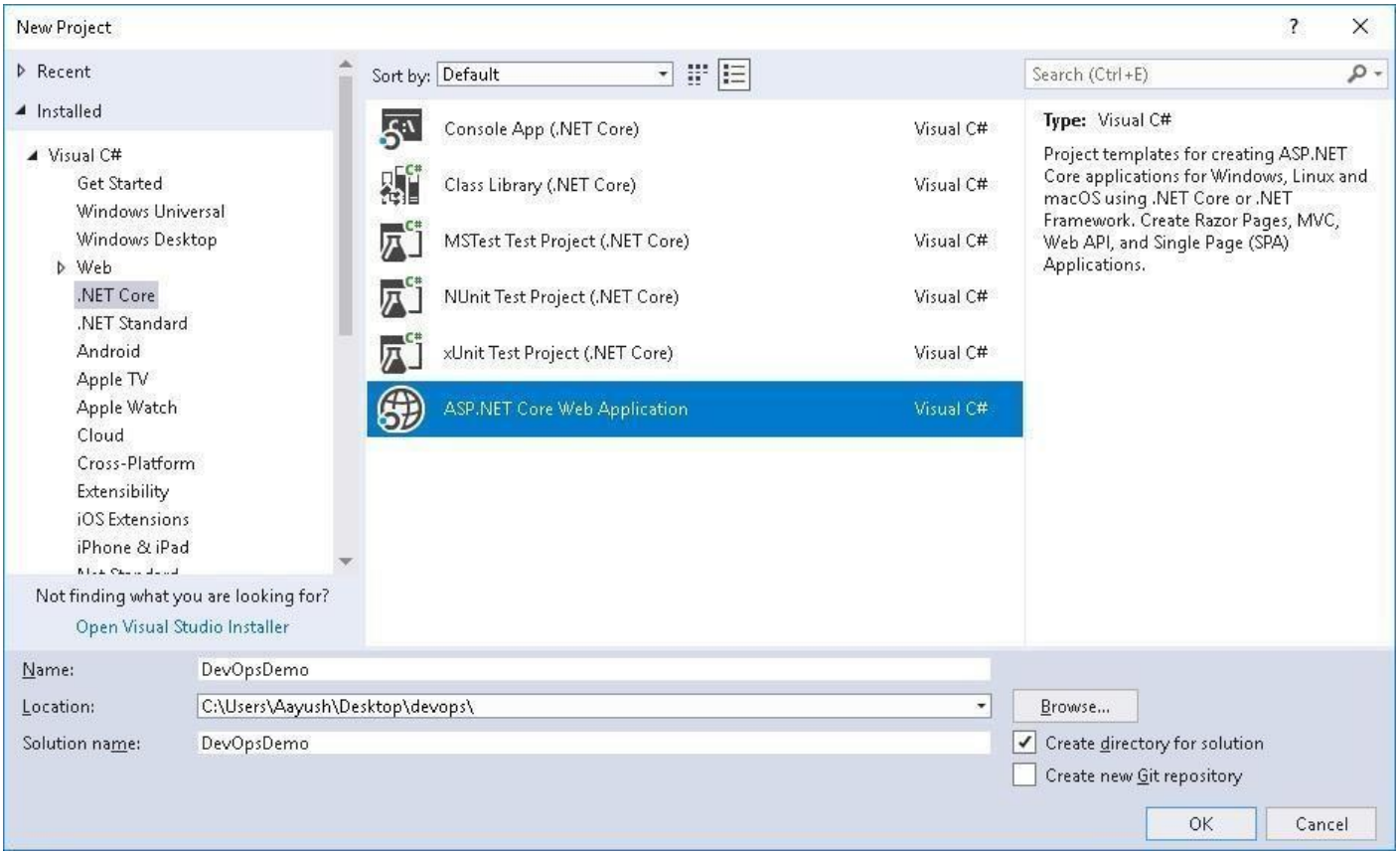
4.1 Create Asp.Net Core Project

Let's create an **Asp.Net Core Project** in Visual Studio. Open **Visual Studio 2017 or higher version** and go to the **File** menu and choose **New** and then **Project**.

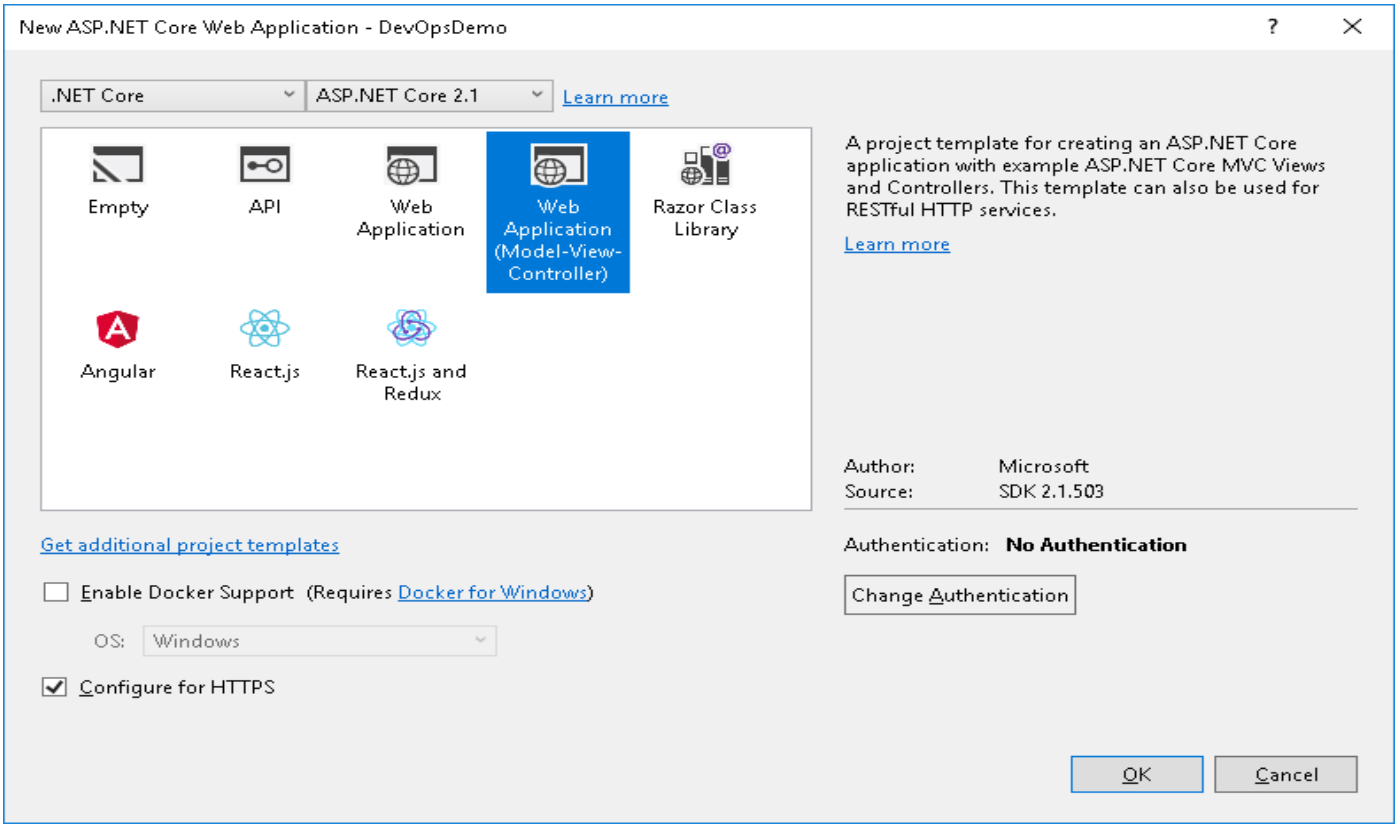


A "**New Project**" dialog window will open as follows, as we are going to create a **.Net Core application**. So, move to the first panel and from **Installed**, choose **Visual C# > Web > .Net Core**. Here we will find different kinds of **.Net Core application** templates. We will select **ASP.NET Core Web Application**.

After selecting the **.Net Core application** template, provide a **suitable name for the project** as well as a solution name like **DevOpsDemo**. This project is being created for a DevOps demonstration, so let's keep the name simple, as DevOpsDemo. Don't forget to provide the project **location** and **click on OK**.

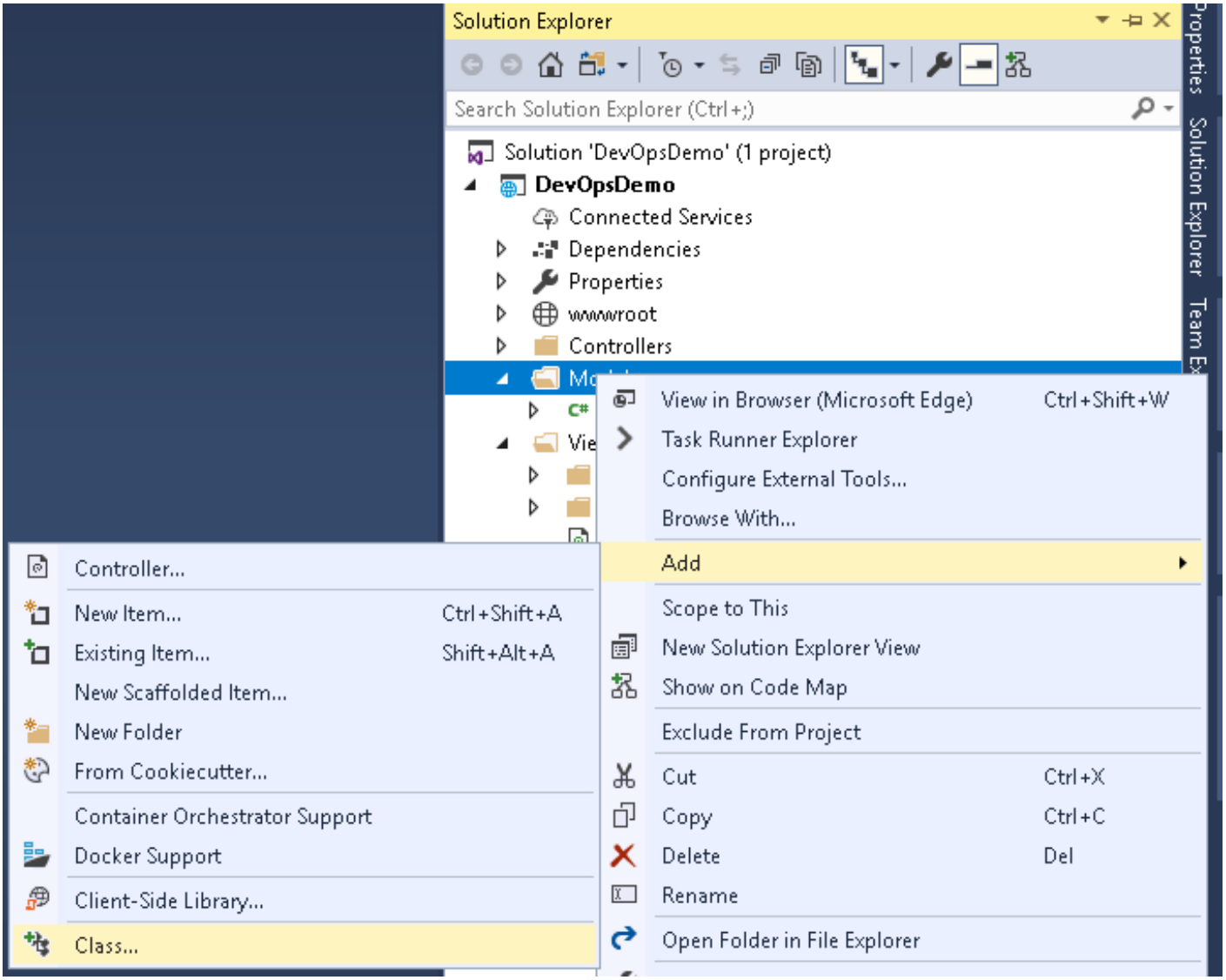


The next dialog window will ask to choose the **Project template**, here, we are working on **ASP.NET Core 2.1**. We have different kinds of project templates available like **API, Web Application, Model View Controller** etc. Here we will select **Web Application (Model-View-Controller)** which enables the functionality of MVC. Apart from this, we require two more things; first check on the checkbox for configuring the HTTPS and change authentication and choose **No Authentication. Now click on OK**.

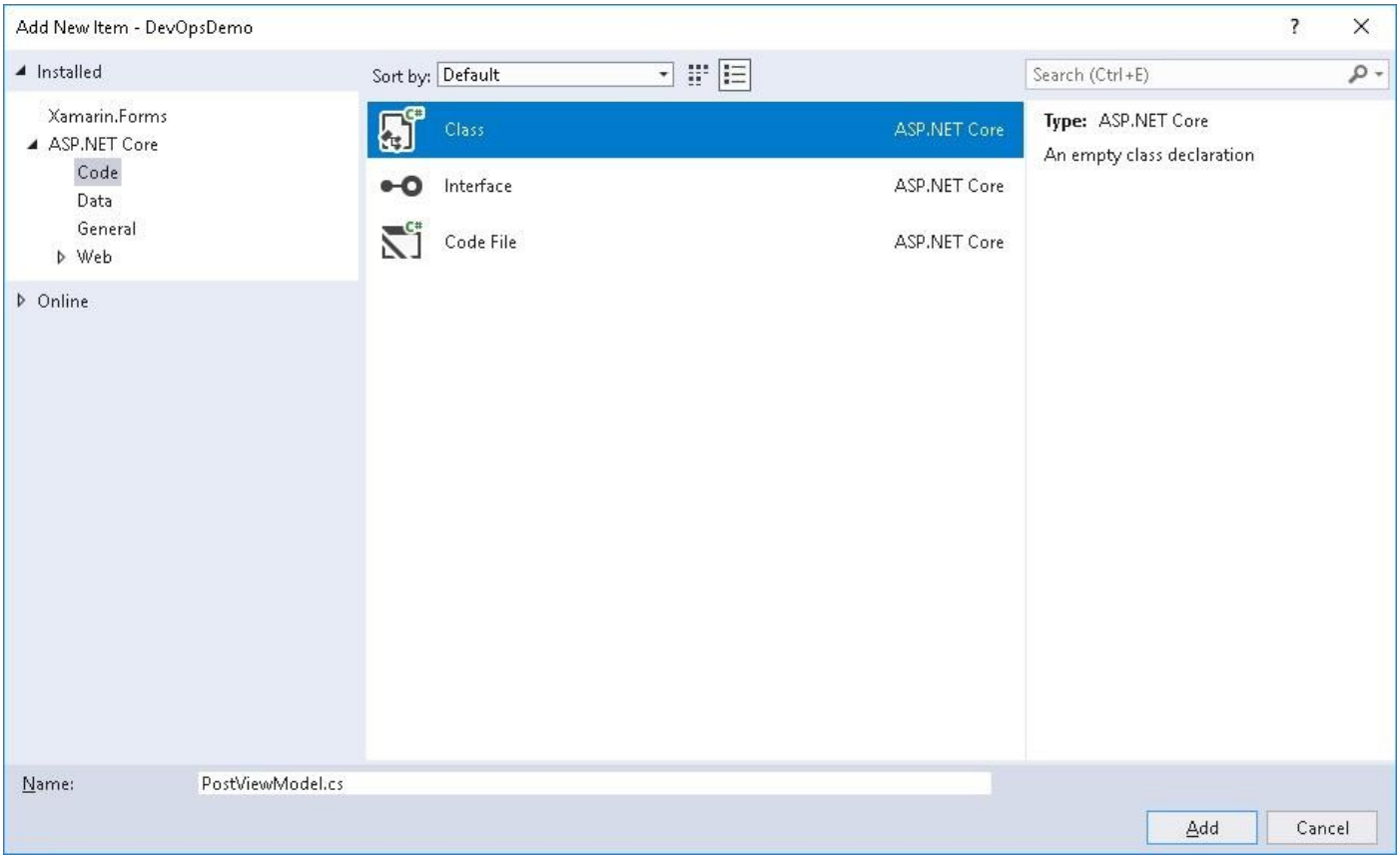


It will take a few moments to create and configure the project and the final project will be ready. Once the project is ready, we can see the basic MVC structure application in ASP.NET Core Web Application.

Let's add some Model-View-Controller functionality. So, let's create a **Model** class for **'Post'**. **Right click** on the Model folder from the project and choose **Add** and then choose **Class**.



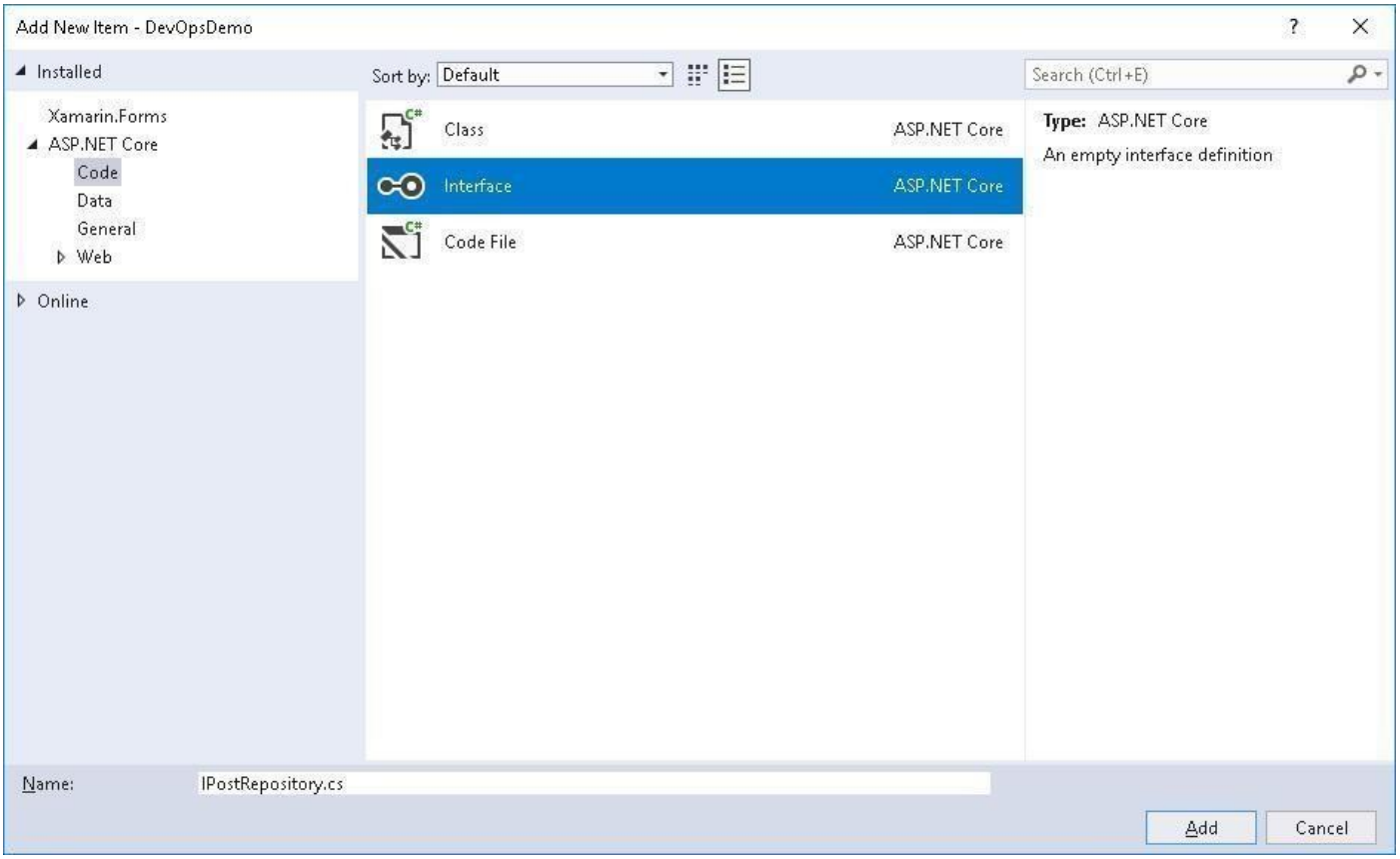
It will open the **Add New Item** dialog window, and from here we can choose different kinds of Model data files like class, interface etc. So, just choose Class and provide the name for the class as **'PostViewModel.cs'** and click on **Add**.



Open the **PostViewModel** and update the class's code as follows. Here we are adding four properties like **PostId, Title, Description and Author** for PostViewModel class**.**

```
namespace DevOpsDemo.Models
{
  public partial class PostViewModel
  {
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Description { get; set; }
    public string Author { get; set; }
  }
}
```

The next step is to create repository information. So, let's create one more folder for repository classes and interfaces as Repository. Once **Repository** folder is created then right click on folder and select **Add > New Item**. It will open **Add New Item** dialog window from where we can select the file type. So, first select an **Interface** and provide the name for interface as **IPostRepository.cs** and click on **Add**.

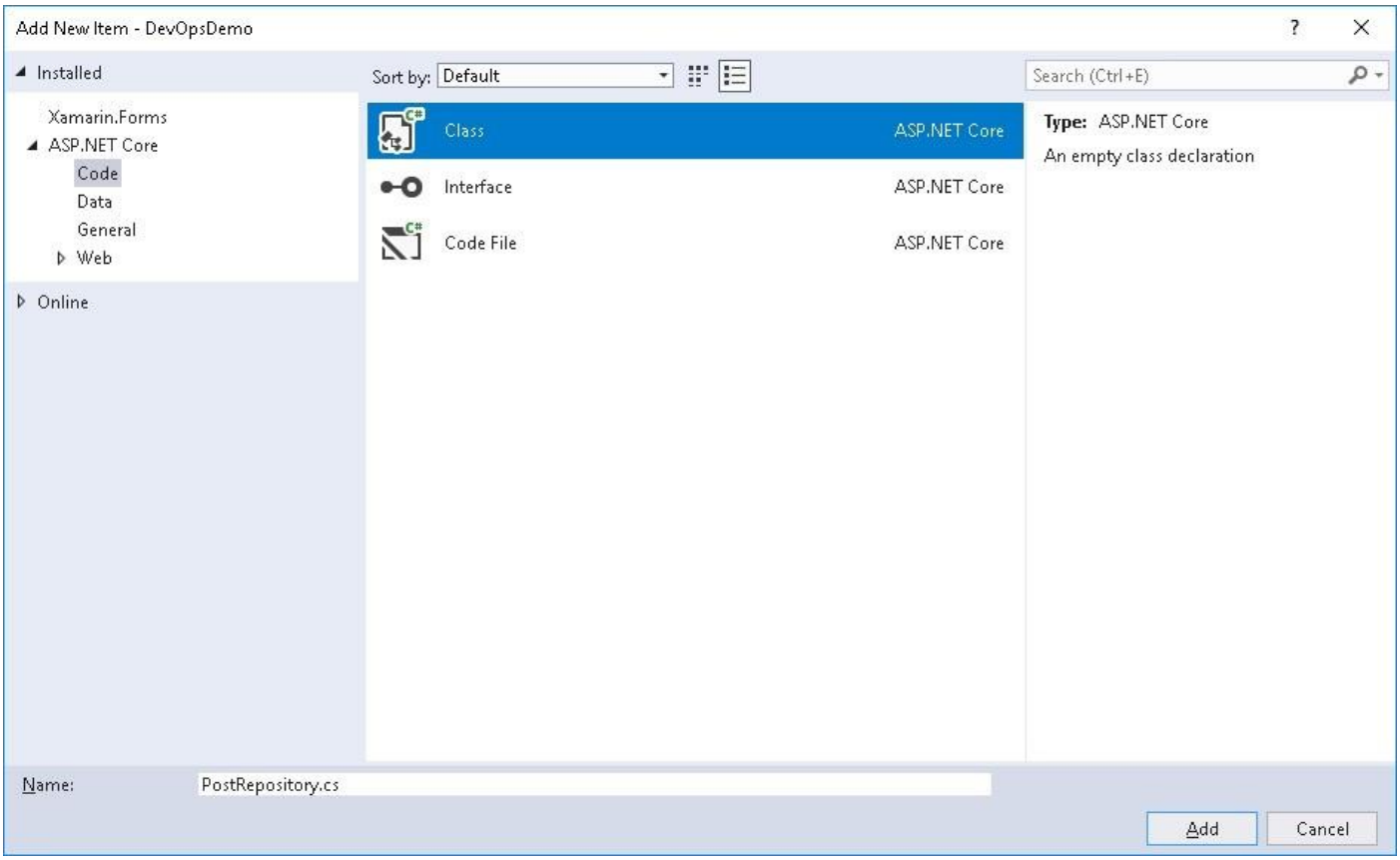


Update the code for the **IPostRepository** interface as follows. Here we are adding one method which will return the list of Post.

IPostRepository.cs

```
using DevOpsDemo.Models;
using System.Collections.Generic;

namespace DevOpsDemo.Repository
{
   public interface IPostRepository
   {
      List<PostViewModel> GetPosts();
   }
}
```

The same process needs to be followed to add one new class as **PostRepository.cs** in Repository folder as follows.



Here is the **PostRepository** class which implements the **IPostRepository**. We are creating some dummy data for Posts. We are keeping it simple and not implementing the database-driven functionality for getting the real time data. It is because this demonstration is only for understanding how DevOps works and how we can implement Continuous Integration and Continuous Delivery.

PostRepository.cs

```
using DevOpsDemo.Models;
using System.Collections.Generic;

namespace DevOpsDemo.Repository
{
   public class PostRepository : IPostRepository
   {
      public List<PostViewModel> GetPosts()
      {
         var posts = new List<PostViewModel> {
            new PostViewModel(){ PostId =101, Title = "DevOps Demo Title 1", Description ="DevOps Demo Description 1", Author="Mukesh Kumar"},
            new PostViewModel(){ PostId =102, Title = "DevOps Demo Title 2", Description ="DevOps Demo Description 2", Author="Banky Chamber"},
```

```
            new PostViewModel(){ PostId =103, Title = "DevOps Demo Title 3", Description ="DevOps Demo Description 3", Author="Rahul Rathor"},
        };

        return posts;
    }
  }
}
```

Now, it's time to show the data which are returning from repository on View. So, let's open the **HomeController** and implement the **constructor dependency injection** for getting the instance of the **PostRepository** class and create a **ActionResult** as **Index**. Here, in the Index method, we will fetch the data using **PostRepository** instance and return the data into the View.

HomeController.cs

```
using DevOpsDemo.Models;
using DevOpsDemo.Repository;
using Microsoft.AspNetCore.Mvc;
using System.Diagnostics;

namespace DevOpsDemo.Controllers
{
  public class HomeController : Controller
  {
    IPostRepository postRepository;
    public HomeController(IPostRepository _postRepository)
    {
      postRepository = _postRepository;
    }

    public IActionResult Index()
    {
      var model = postRepository.GetPosts();

      return View(model);
    }

    public IActionResult About()
    {
      ViewData["Message"] = "Your application description page.";

      return View();
    }

    public IActionResult Contact()
    {
      ViewData["Message"] = "Your contact page.";
```

```
            return View();
        }

        public IActionResult Privacy()
        {
            return View();
        }

        [ResponseCache(Duration = 0, Location = ResponseCacheLocation.None, NoStore = true)]
        public IActionResult Error()
        {
            return View(new ErrorViewModel { RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier });
        }
    }
}
```

We will populate the data on View in tabular format. So, once data will be there, we will iterate on it and display the data as follows.

Index.cshtml

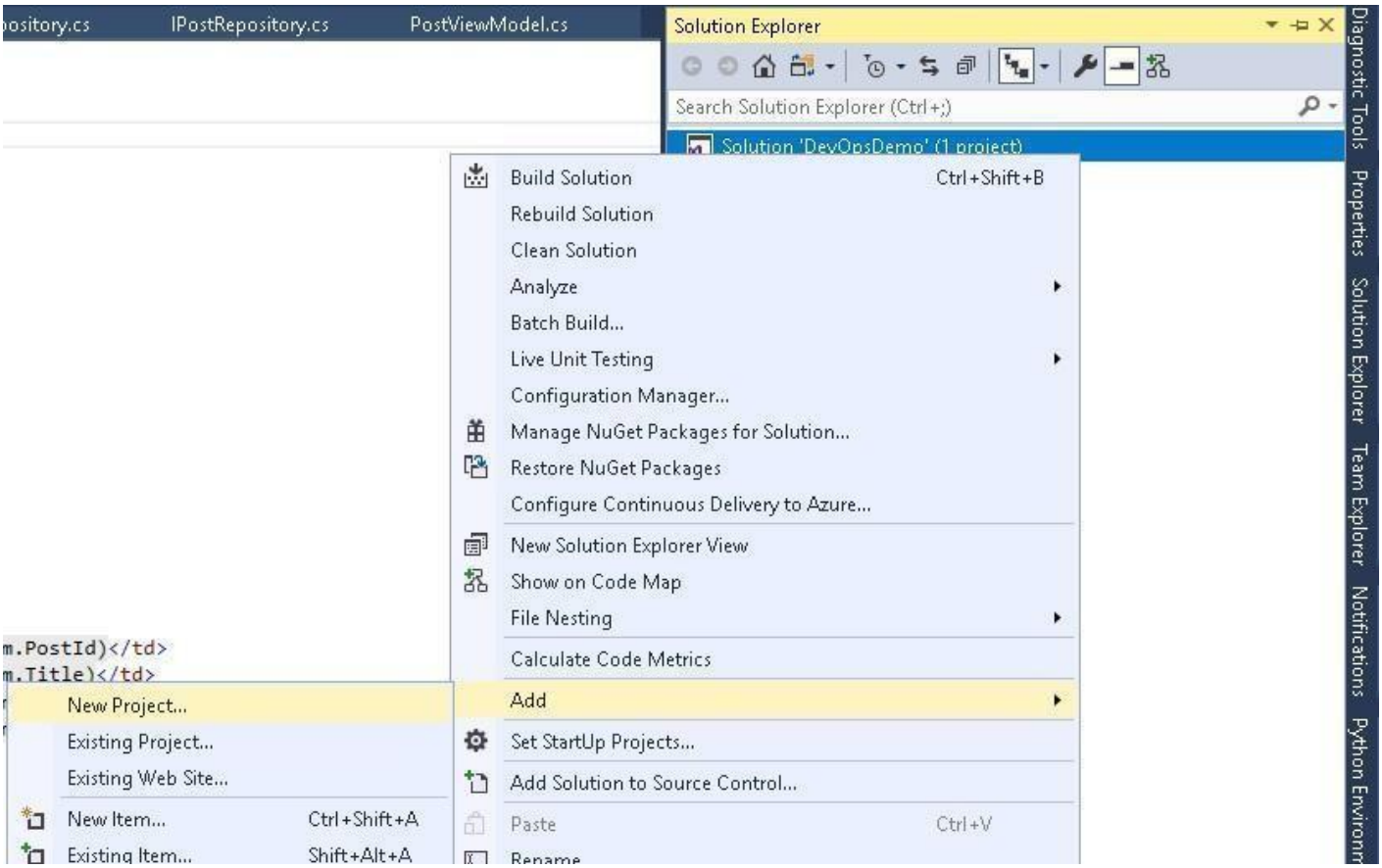
```
@model IList<DevOpsDemo.Models.PostViewModel>
@{
    ViewData["Title"] = "Home Page";
}
<div class="row">
    <h2>Post List</h2>
    <table class="table">
        <thead>
            <tr>
                <th>Post Id</th>
                <th>Title</th>
                <th>Description</th>
                <th>Author</th>
            </tr>
        </thead>
        <tbody>
            @foreach (var item in Model)
            {
                <tr>
                    <td>@Html.DisplayFor(modelItem => item.PostId)</td>
                    <td>@Html.DisplayFor(modelItem => item.Title)</td>
                    <td>@Html.DisplayFor(modelItem => item.Description)</td>
                    <td>@Html.DisplayFor(modelItem => item.Author)</td>
                </tr>
            }
        </tbody>
    </table>
</div>
```

4.2 xUnit Test Project

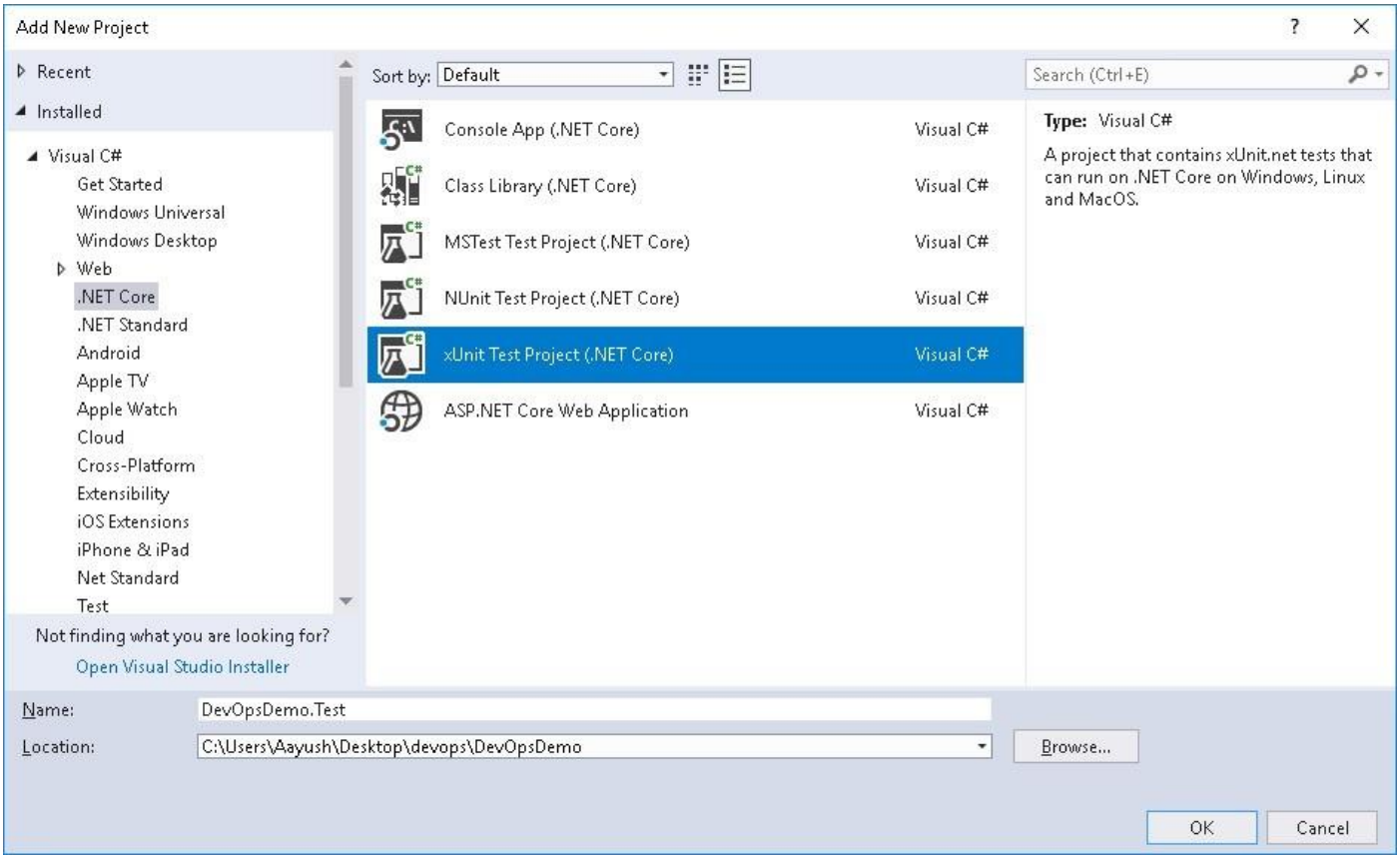
Testing is an important aspect of any product. Without testing, we cannot think that a product is ready to be delivered. We do the testing in many ways but in the development phase, while writing the code for specific functionality, we write the test cases against the functionality and check if everything is working fine as expected or not.

Unit Test Cases help us to find the bugs or issues in the code in the earlier stage while building the code. At the time of building the code and creating the artifact, it also executes the test cases and if test cases are failing, it means, something is wrong and functionality is not as expected.

Here, we will create a separate testing project for ASP.NET Core Web Application. So, let's right click on the **DevOpsDemo** solution and select **Add > New Project** as shown in the following image.

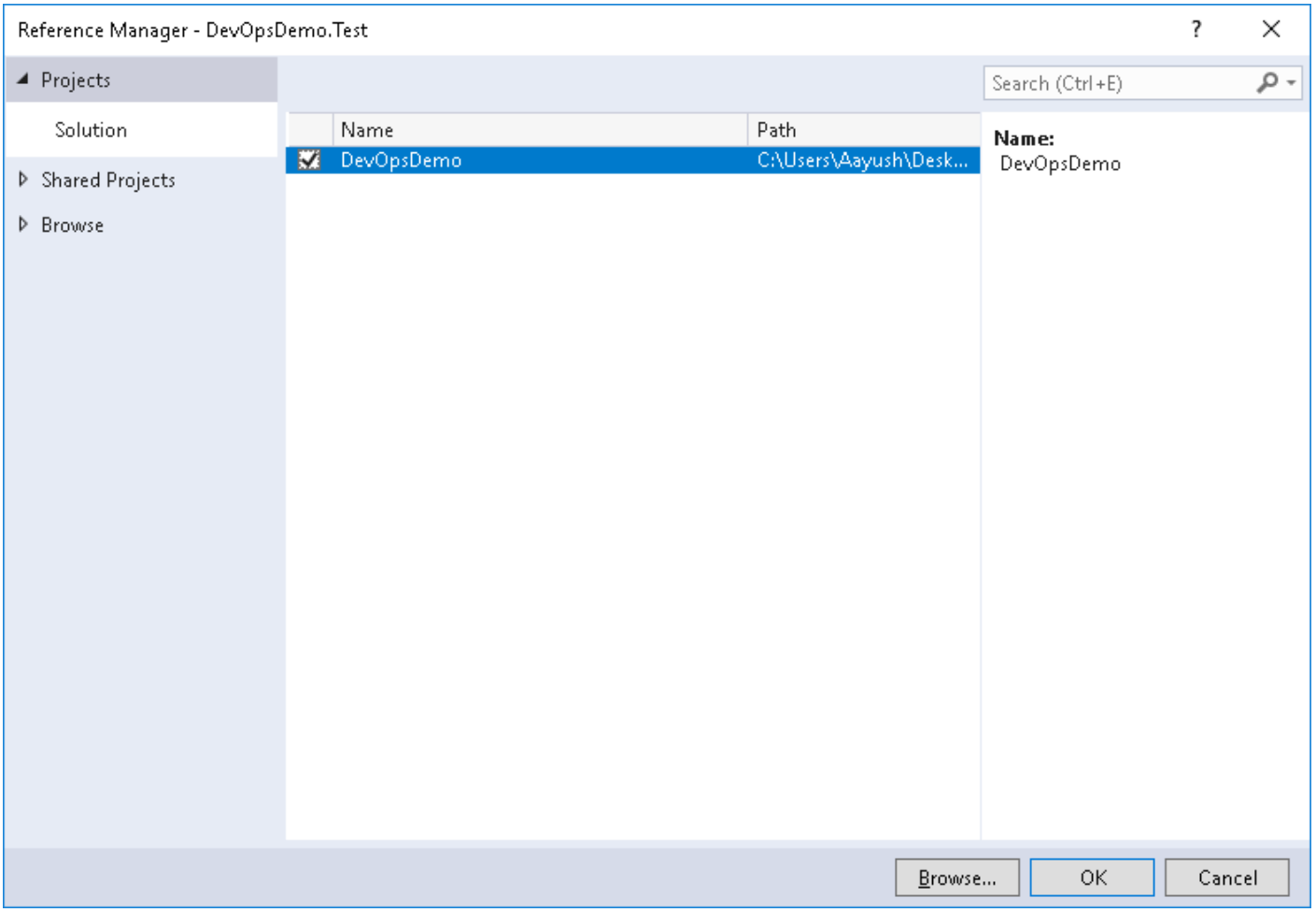


It will open the New Project dialog window. Here we have to follow the same process as we have done above for creating the new ASP.NET Core Web Application project. Only since we have to change the application template for a testing project, we will choose the **xUnit Test Project (.Net Core).** After selecting the project template, provide the name of the testing project as **DevOpsDemo.Test** and click on **OK**.



Within a few seconds, the xUnit testing project will be ready. It will contain one unit test class as **UnitTest1.cs**. Before moving to the next page, just rename **UnitTest1** to **PostTestController**.

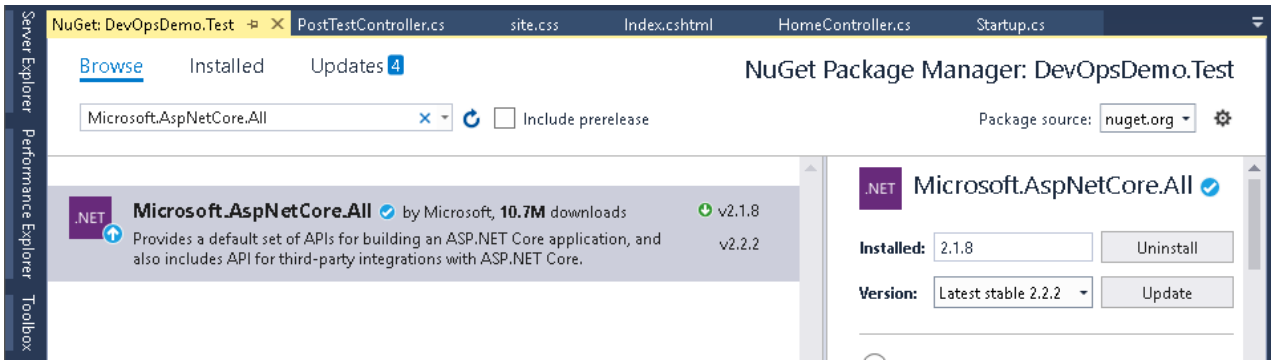
We will now test the main project functionality. So, it is time to add the main project reference in the test project so that we can access the repository and controller for writing the test cases. So, **Right click on the Dependencies in testing project and select Add Reference.** It will open the Reference Manager for **DevOpsDemo.Test** project. From the Projects section in left panel, select the **DevOpsDemo** (mark checked) and click to **OK**.



Install Microsoft.AspNetCore.All from NuGet (version 2.1.8)

This is the xUnit testing project, and by default we cannot get all the Asp.Net Core functionality. So, let's first add the packages which will provide a complete set of APIs for building the Asp.NET Core application. Let's **right click on Dependencies** and select **Manage NuGet Packages**. It will open the **NuGet Package Manager** from where new packages can be searched for installation, or see the installed packages, or see if any update is available for any package.

So, go to the **Browse** section and search for **Microsoft.AspNetCore.All** in the search section and install it. For this demonstration, we are using version **2.1.8 for Microsoft.AspNetCore.All**.


NuGet: DevOpsDemo.Test
PostTestController.cs
site.css
Index.cshtml
HomeController.cs
Startup.cs
Server Explorer
Performance Explorer
Toolbox
Browse
Installed
Updates 4
NuGet Package Manager: DevOpsDemo.Test
Microsoft.AspNetCore.All
Include prerelease
Package source:
nuget.org
Microsoft.AspNetCore.All by Microsoft, 10.7M downloads
Provides a default set of APIs for building an ASP.NET Core application, and also includes API for third-party integrations with ASP.NET Core.
v2.1.8
v2.2.2
Microsoft.AspNetCore.All
Installed:
2.1.8
Uninstall
Version:
Latest stable 2.2.2
Update

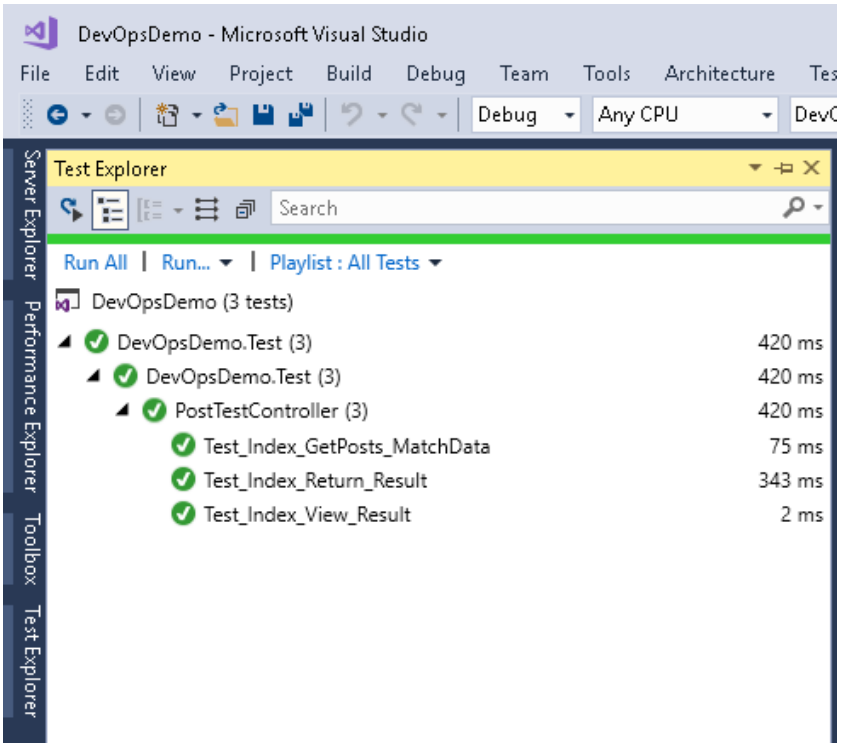
Open the **PostTestController** and write the few unit test cases for **HomeController** as follows.

PostTestController.cs

```
using DevOpsDemo.Controllers;
using DevOpsDemo.Models;
using DevOpsDemo.Repository;
using Microsoft.AspNetCore.Mvc;
using System;
using System.Collections.Generic;
using Xunit;

namespace DevOpsDemo.Test
{
   public class PostTestController
   {
      private PostRepository repository;

      public PostTestController()
      {
         repository = new PostRepository();

      }

      [Fact]
      public void Test_Index_View_Result()
      {
         //Arrange
         var controller = new HomeController(this.repository);

         //Act
         var result = controller.Index();

         //Assert
         Assert.IsType<ViewResult>(result);
      }
      [Fact]
      public void Test_Index_Return_Result()
      {
         //Arrange
         var controller = new HomeController(this.repository);

         //Act
         var result = controller.Index();
```

```
            //Assert
            Assert.NotNull(result);
        }


        [Fact]
        public void Test_Index_GetPosts_MatchData()
        {
            //Arrange
            var controller = new HomeController(this.repository);
            //Act
            var result = controller.Index();
            //Assert
            var viewResult = Assert.IsType<ViewResult>(result);
            var model = Assert.IsAssignableFrom<List<PostViewModel>>(viewResult.ViewData.Model);
            Assert.Equal(3, model.Count);
            Assert.Equal(101, model[0].PostId);
            Assert.Equal("DevOps Demo Title 1", model[0].Title);
        }
    }
}
```

Let's open the **Test Explorer** and click to **Run All** as shown in following image to start executing the unit test cases, which start building the solution and start executing the unit test cases. Once all unit test cases run then you can see the status as green. As per the following image, we can see that all test cases are passed.

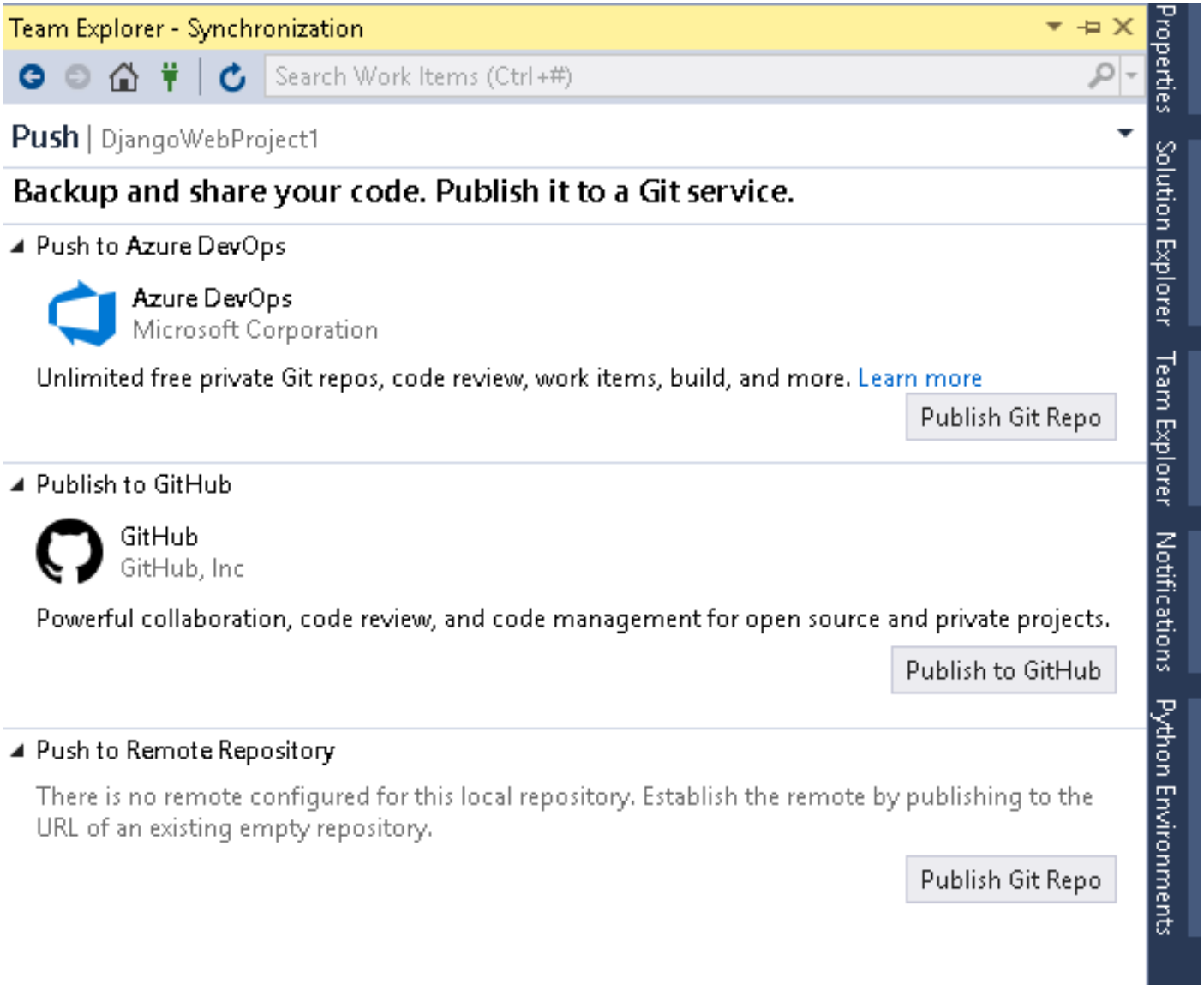


4.3 Add Project to GitHub

Now it's time to add this above project to any repository like TFS, GitHub, Bitbucket etc. so that we can access it while setting up Azure DevOps pipeline. Let's choose the **GitHub** so that it will be accessible publicly. So, right click on the Solution **(DevOpsDemo)** and select **Add Solution to Source Control**. It will add your file in a local repository

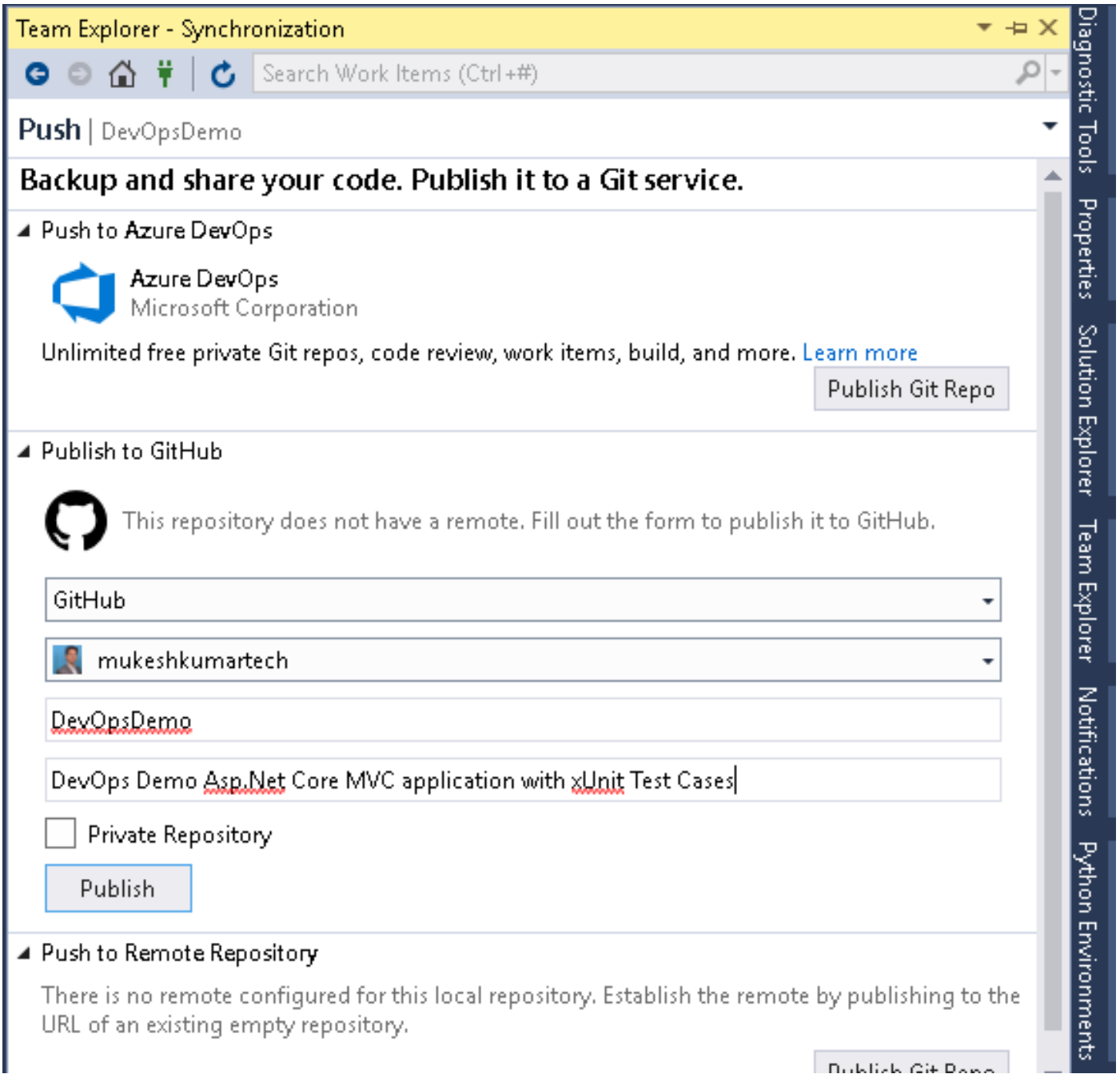
Now, go to Team Explorer, here we can find the **'Sync'** option, just click on it. It will open the window from where we can publish code to a particular repository like **GitHub** as follows.

Here, we will choose the **Publish to GitHub**, so that we can publish this code to **GitHub** public repository.



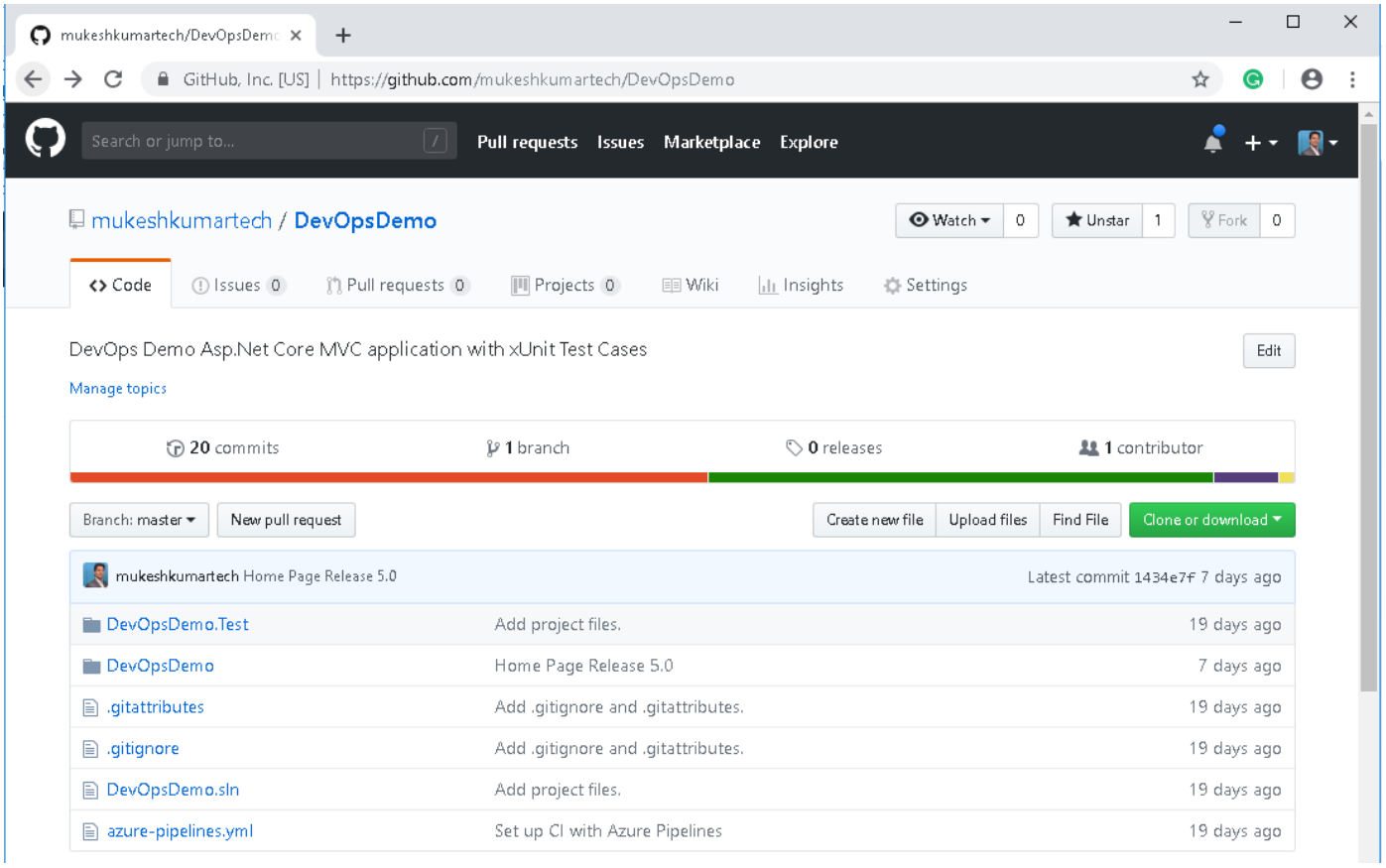
Once we click on the **Publish to GitHub**, it will ask for **Authentication with your GitHub account**. Let's provide the credentials to log in with **GitHub** Account. After logging in with **GitHub**, we will get the following screen. Here, we can provide the **name** and **description** of the repository which will create in **GitHub** for adding this project. We can make it private to check the checkbox for Private Repository. For this demonstration, we will keep it public, so let's click on the Publish button.

It will take few minutes to create the new repository in **GitHub** with provided name and publish the whole code into this repository.



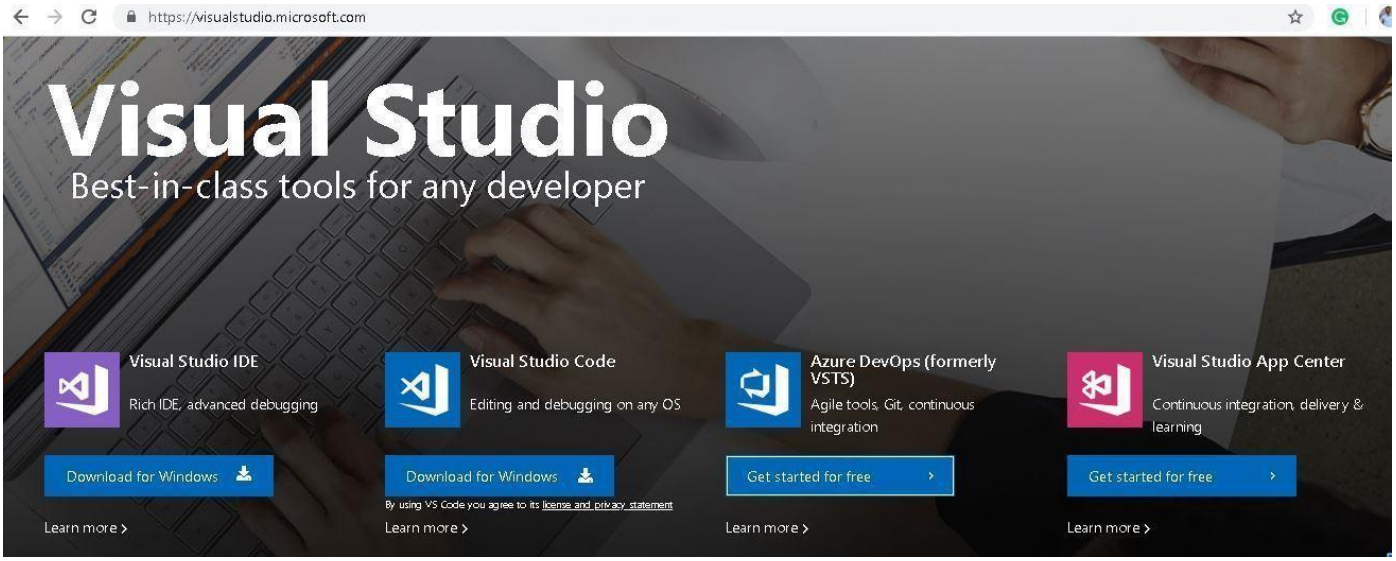
Now, go to following GitHub URL where we can find the published code for this whole demonstration.

https://github.com/mukeshkumartech/DevOpsDemo

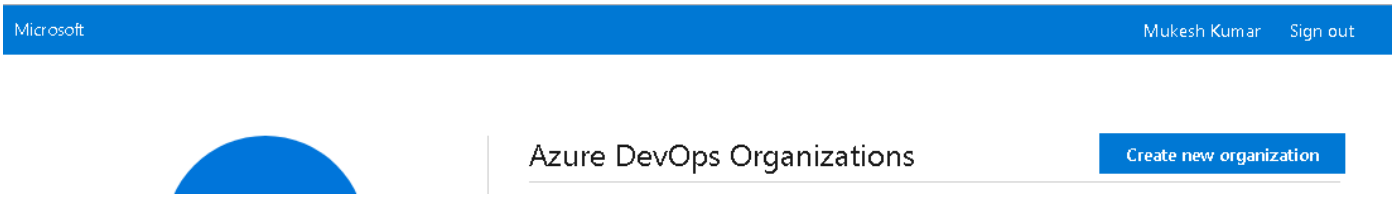


Chapter 5.Create Organization and Project

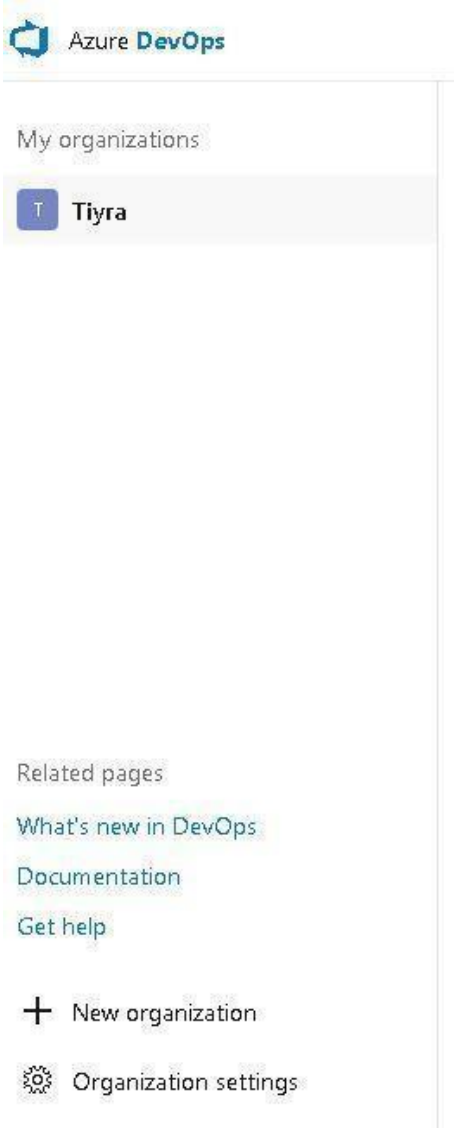
Let's move to **https://visualstudio.microsoft.com** and create new account or **sign in** with existing credentials.



Once we are logged in successfully, then we have a new screen as follows. From here we can create a new organization on clicking **'Create new organization'** button.

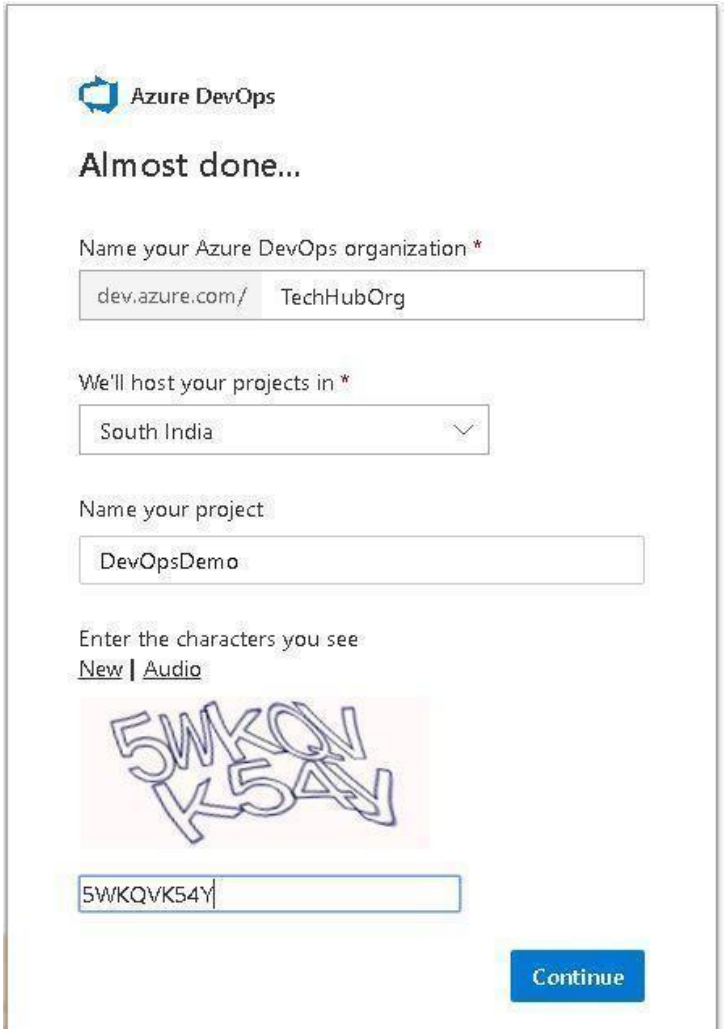


We have another option to create the organization, if we have already an organization and we would like to add a new one then we can create one using the **Create Organization** option as follows.

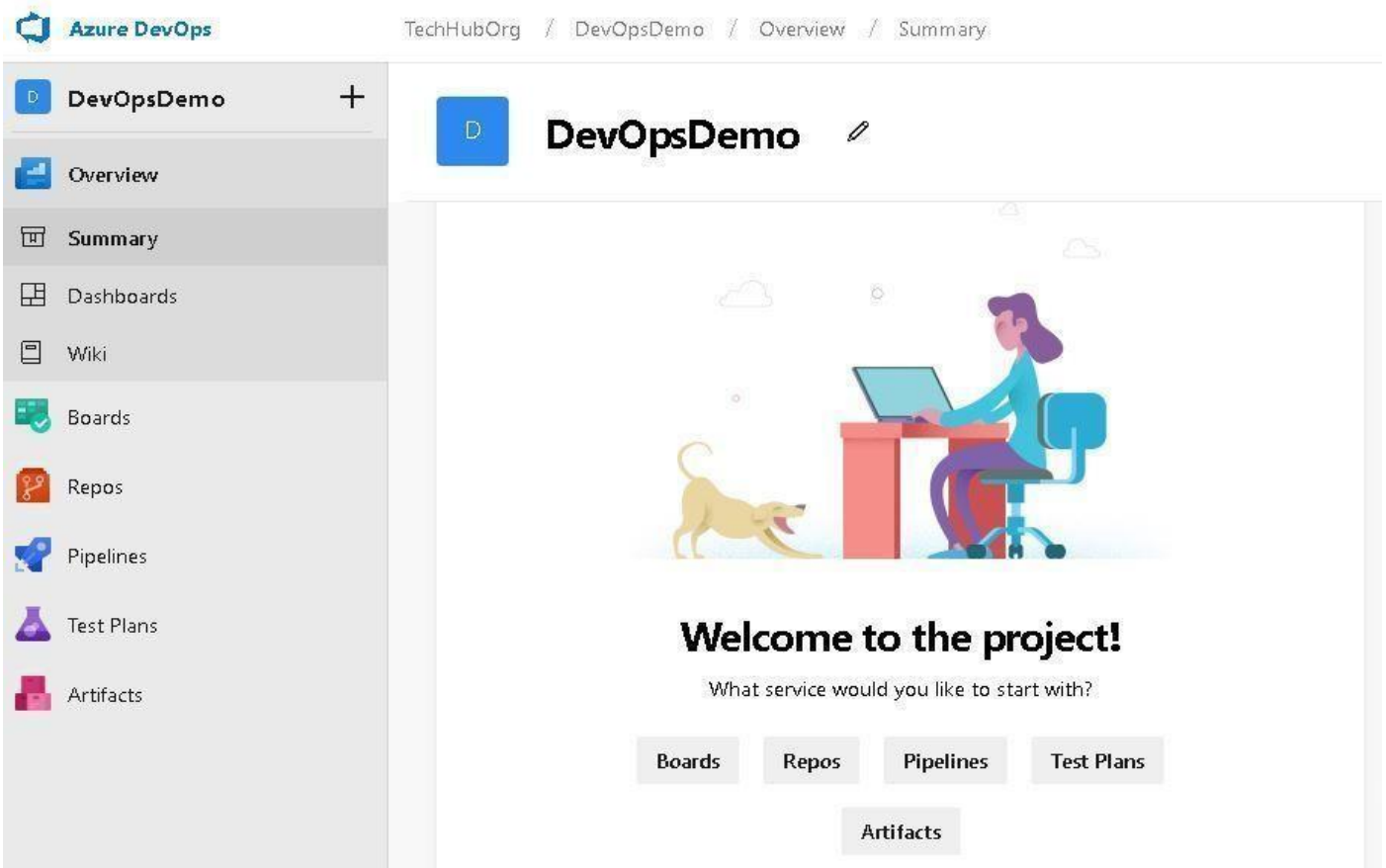


It will take a few seconds to configure the Azure DevOps new organization

Next screen will ask about the name of the **Azure DevOps organization**, here we are giving the name as '**TechHubOrg'**, we also have to provide the hosting location as '**South India'** for our organization. At the time of creating the new organization, we can also provide the name of the project. This project will auto create inside this new organization. Now, fill in the security question and click the **Continue** button.



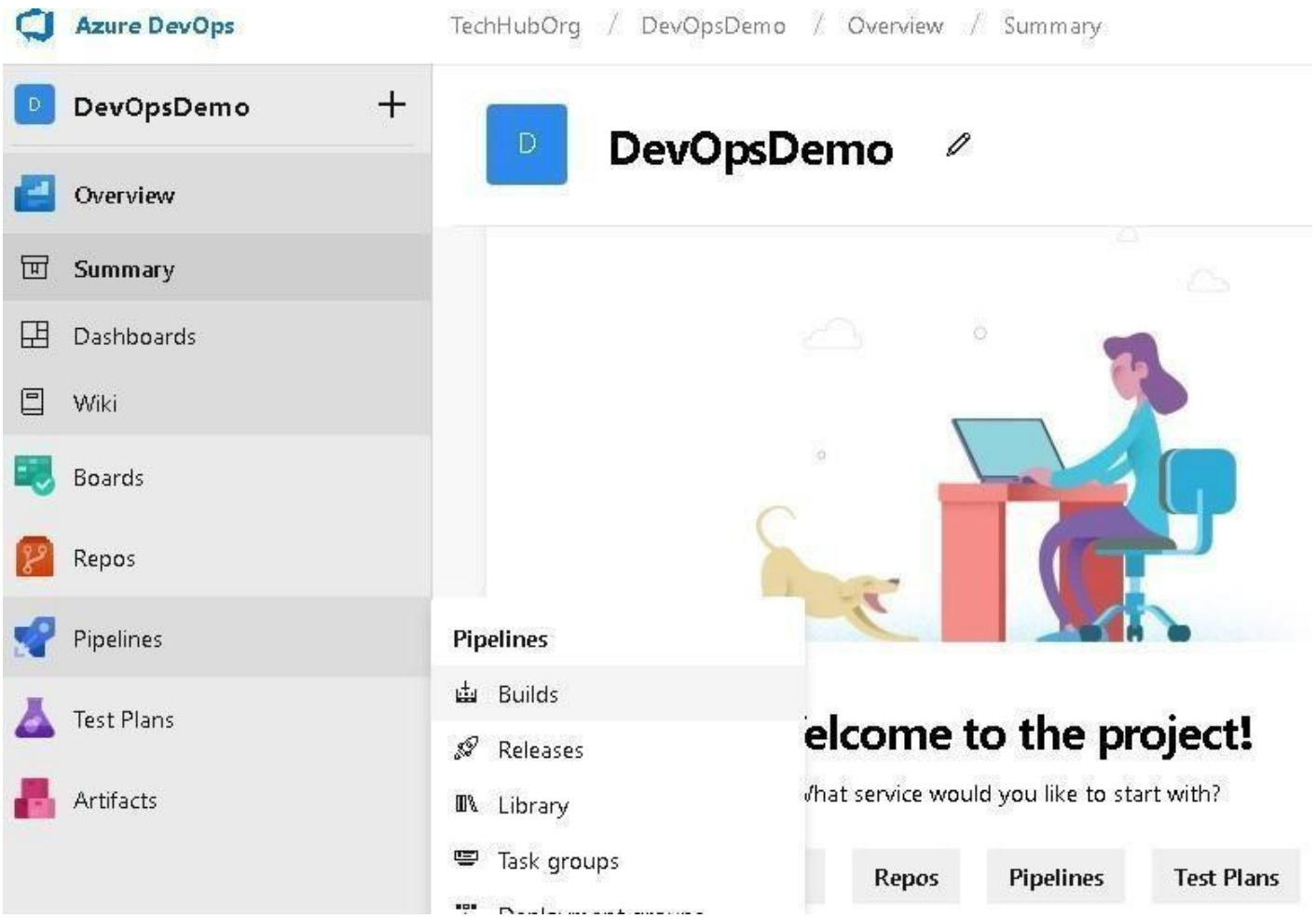
The next screen will open the project which has been created recently at the time of creating the organization. Here, we have created **DevOpsDemo** project inside the **TechHubOrg** organization. Following is the welcome screen for the DevOpsDemo project. From here, we can manage all the things like Boards, Repos, Pipeline and Test Plans etc. which will be project specific.



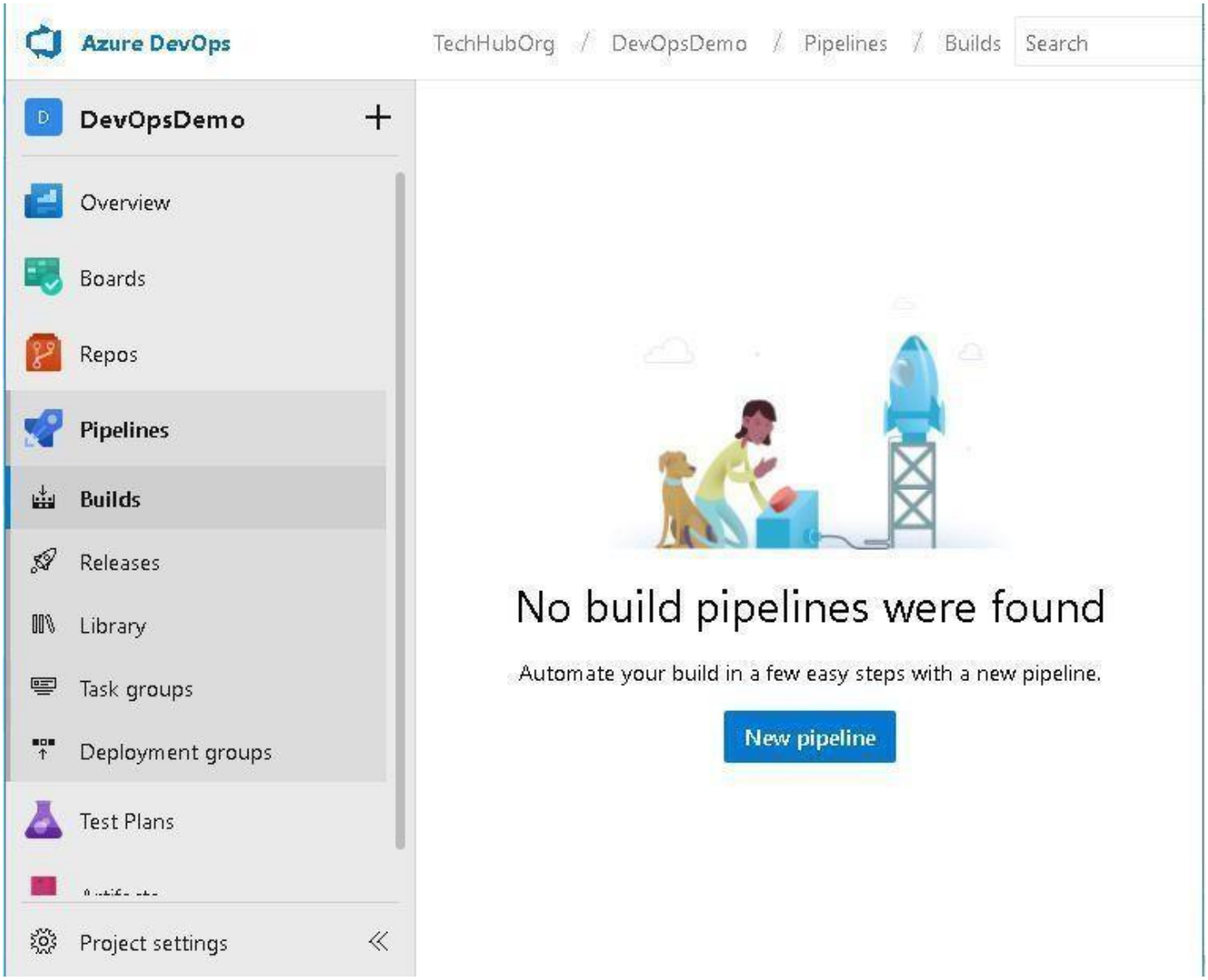
Chapter 6.Continuous Integration

Let's create the Azure Build pipeline. Build pipeline is basically responsible for building the code and testing the corresponding Unit Test Cases once the developer checks in the code into the repository. If everything will be fine, then it will create the Artifact which can be used for deployment.

For creating the build pipeline in **Azure DevOps**, click on **Pipeline** and select **Build** as shown in the following images.

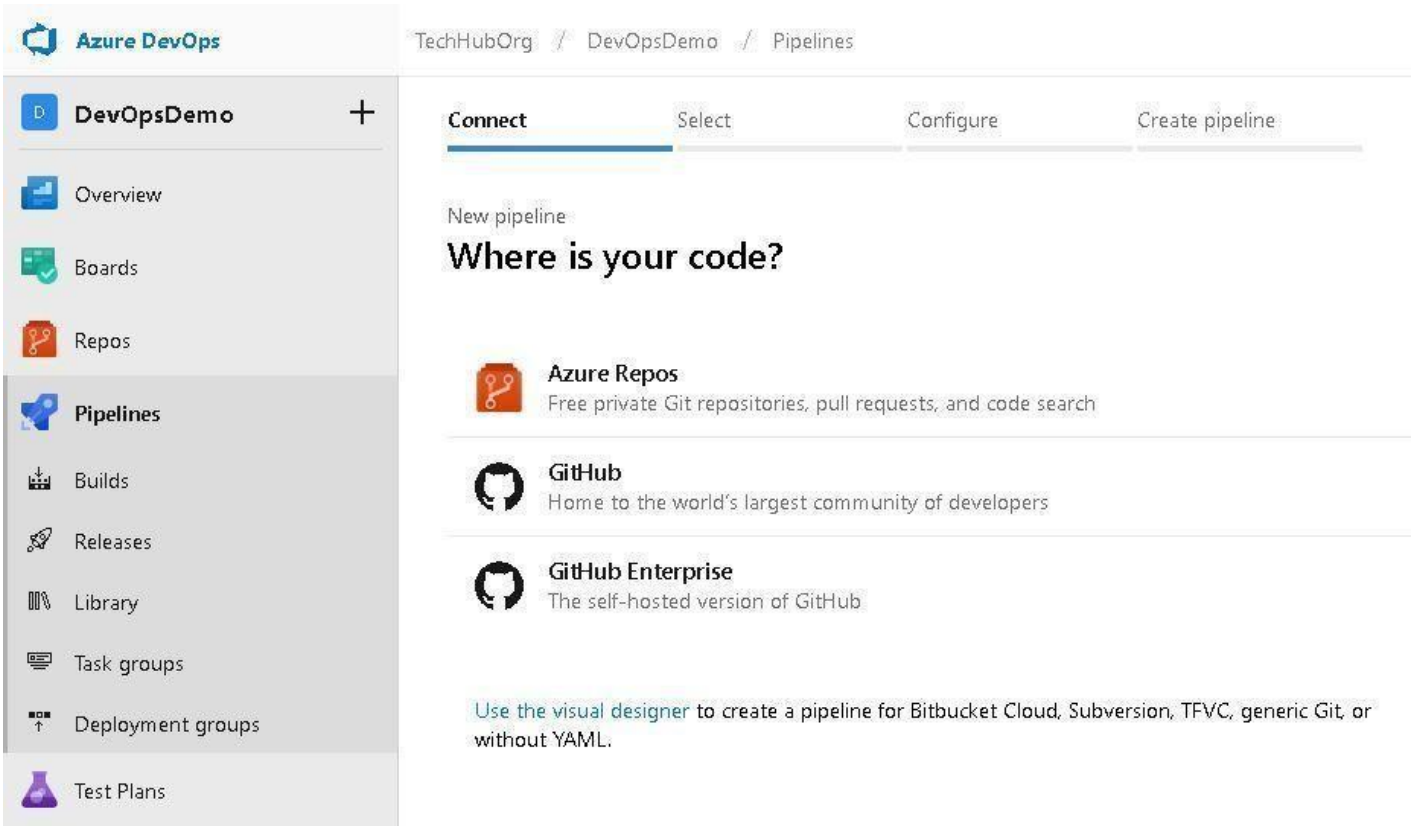


The next screen says that we don't have any build pipeline setup yet and it has one button as new pipeline for creating a new one. So, let's click on **New Pipeline**. It will only create the build pipeline.

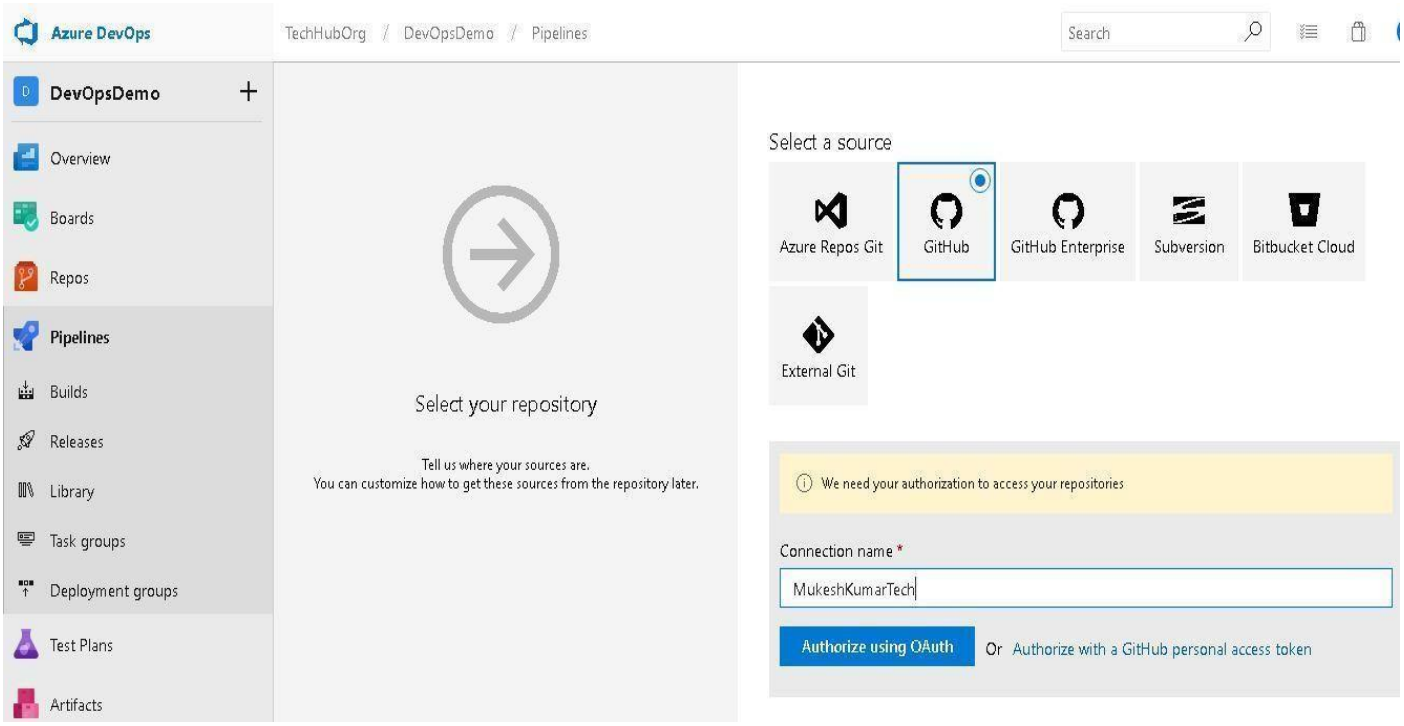


After clicking on **New Pipeline**, it will start the wizard for setting up the build pipeline. It will basically ask for the location of the code repository file and do some configuration.

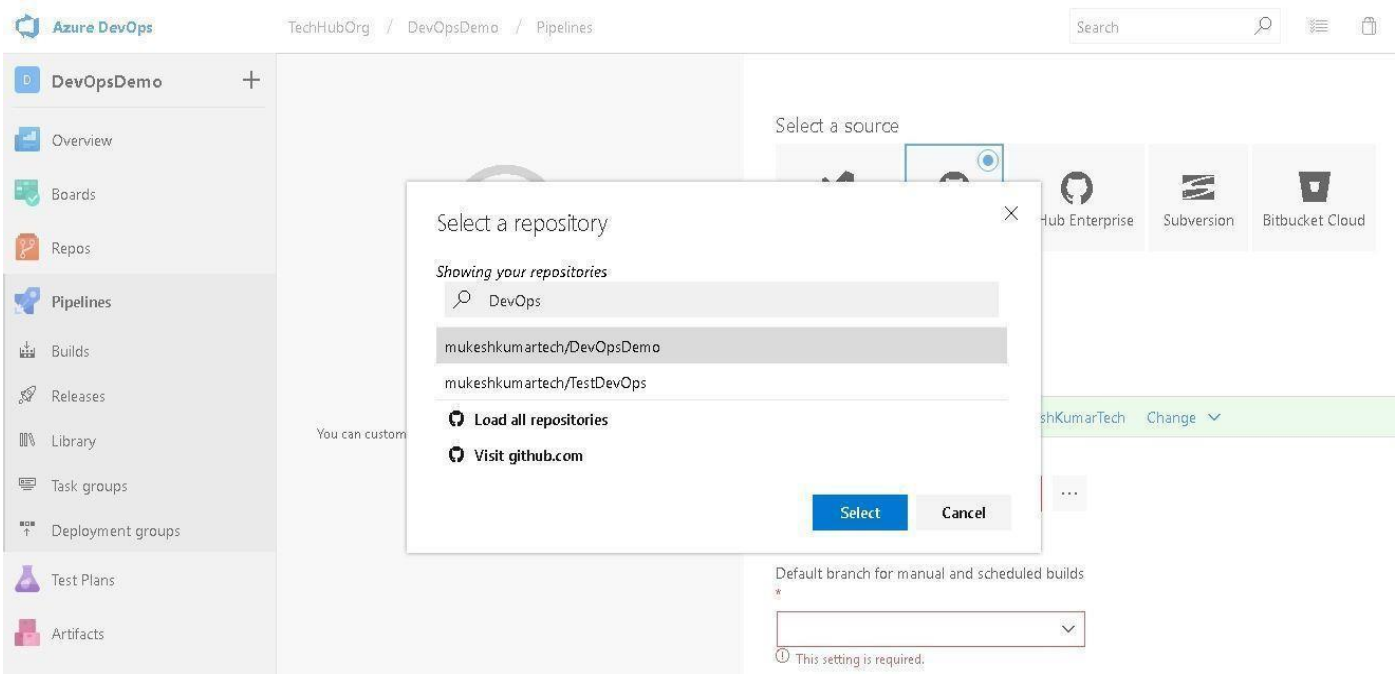
Here, we will not go with Wizard process, we will use the **Visual Designer** for creating and configuring the Azure DevOps Build pipeline. So, let click on the link **'Use the Visual Designer'** as shown in below image.



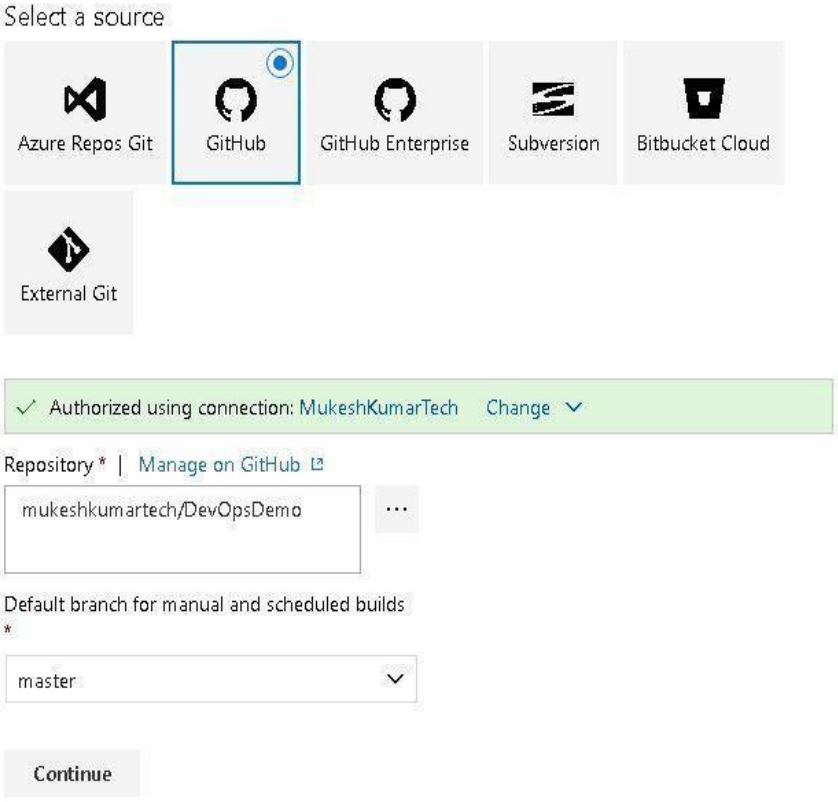
As we select the Visual Designer, it will ask to choose the repository. All available repositories are part of the Azure DevOps like Azure Repo Git, GitHub, Bitbucket etc. We have already pushed our project code which we have already created above in **GitHub**. So, select the **GitHub** and provide the connection name, which we will use further. Now, click to **Authorize using OAuth**. It will open a dialog window for logging in to **GitHub**. Here we have to provide the GitHub credentials for Authorization with **GitHub**.



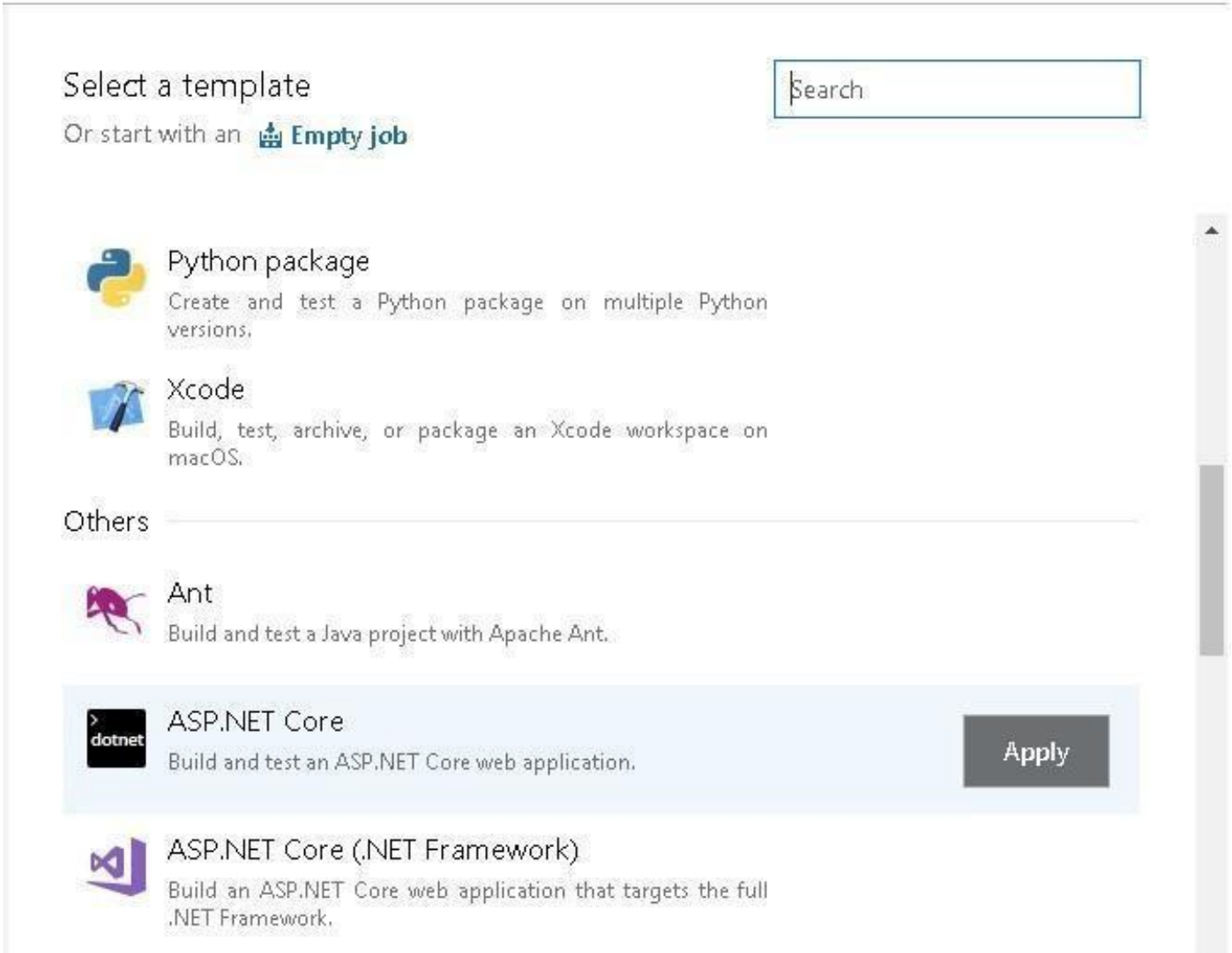
Once we authorize with the GitHub account then we can see all **GitHub** repositories. Here we have to select **DevOpsDemo** because we have created and added our project code into DevOpsDemo repository.



Let's select the repository and then the branch, by default it will be **master [Other branches can also be selected]** and click to **Continue**.


Select a source
Azure Repos Git
GitHub
GitHub Enterprise
Subversion
Bitbucket Cloud
External Git
Authorized using connection: MukeshKumarTech
Change
Repository *
Manage on GitHub
mukeshkumartech/DevOpsDemo
Default branch for manual and scheduled builds *
master
Continue

Next screen will ask about selecting the project template, as we have created the **Asp.Net Core application**. So, here we will choose the **Asp.Net Core** and click to **Apply**.

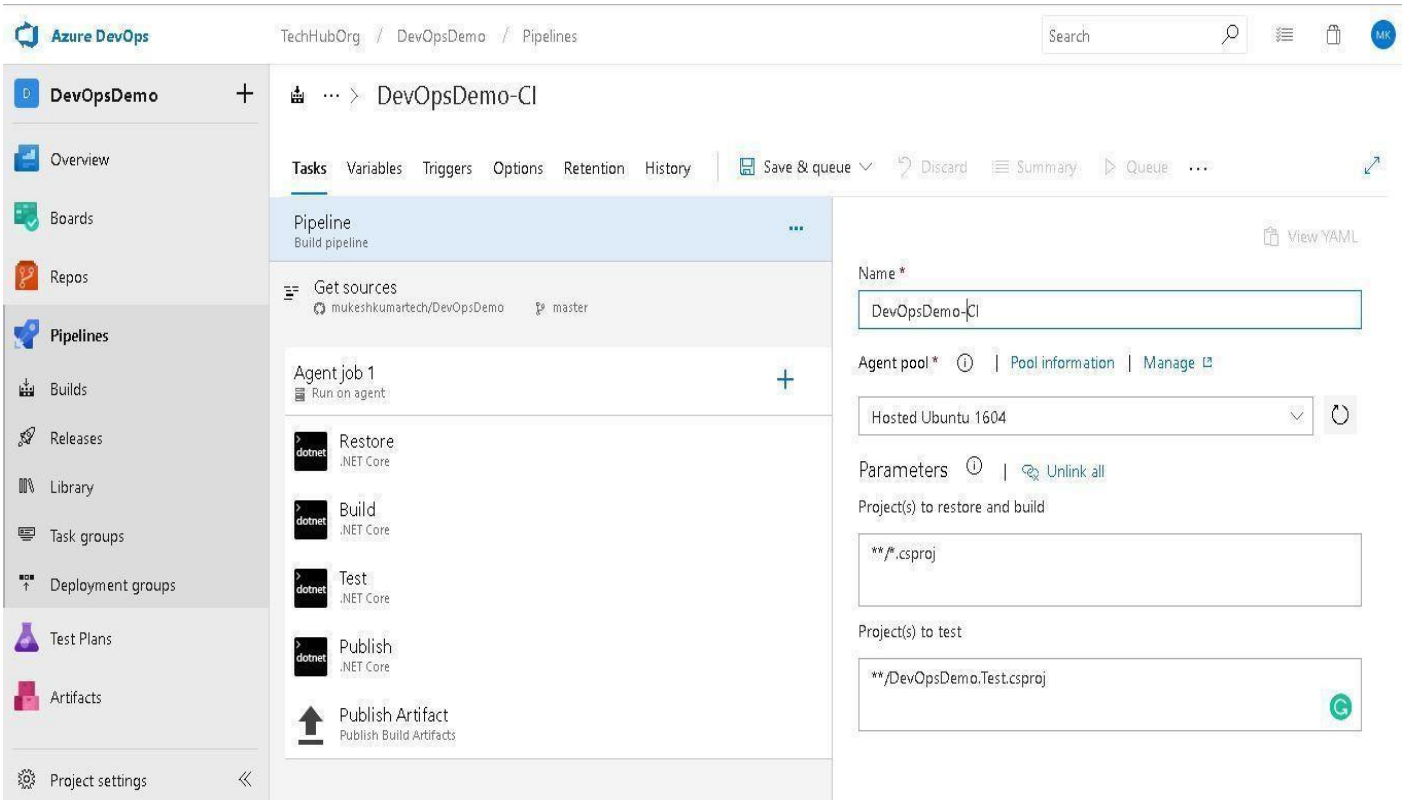


Next screen will be about **Azure DevOps Build pipeline configuration** like the name of the build pipeline, restoring the code from repository, building the code, testing the unit test cases etc.

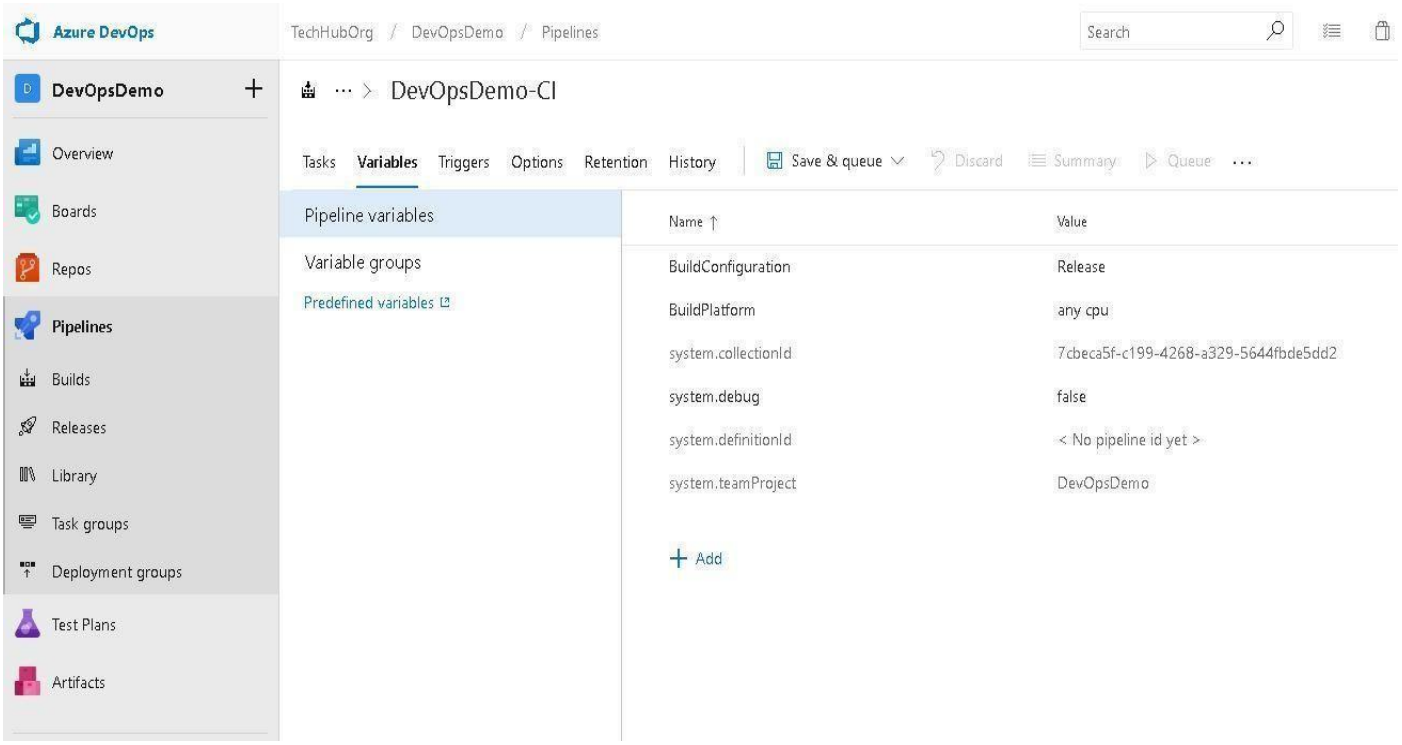
By default, **Tasks** tab will be selected. Here we can provide the **name of build pipeline** as we have given it as **'DevOpsDemo-CI'**.

We have two options for the project, first as 'Projects to restore and build'. For our case, we don't need to do anything. Let's keep the default one. Second is 'Project to test', here, our testing project name was **DevOpsDemo.Test**. So, we will mention it inside **'Project to Test'**.

By default, we have available jobs like **Restore, Build, Test and Publish**. If anything else is required and we would like to perform in between in build process, we can add a new job using the **+ sign**.

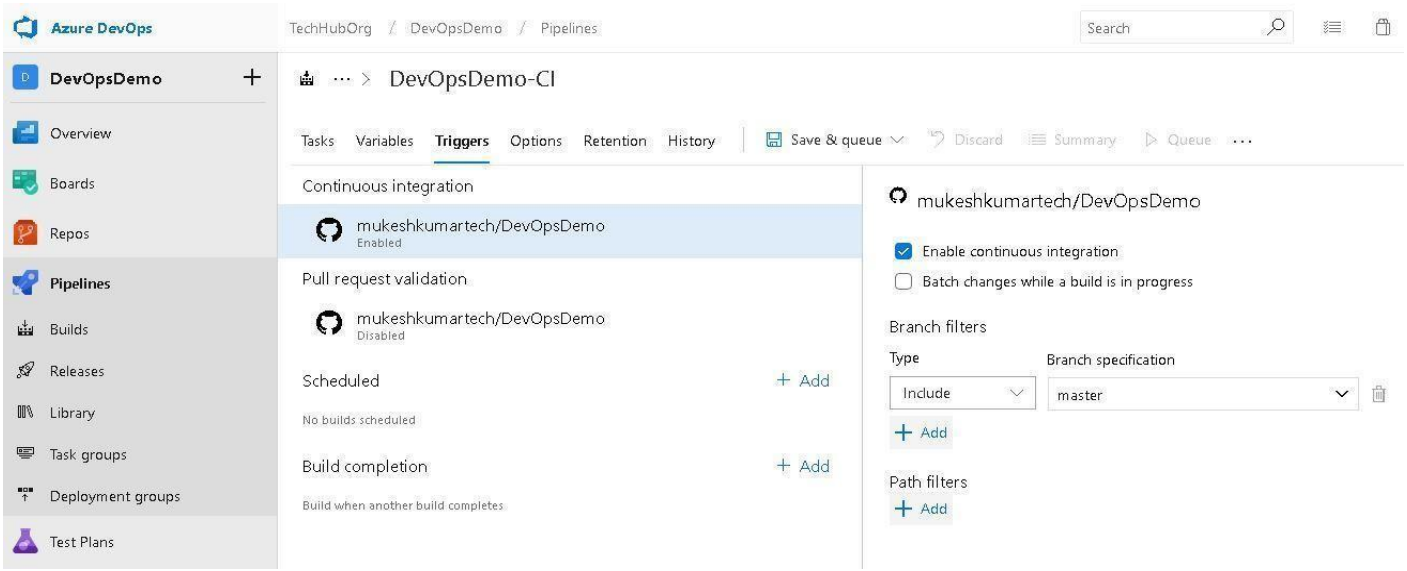


Let's move to **Variables** tab, here we can configure the build setting like **Build Configuration, Build Platform**.



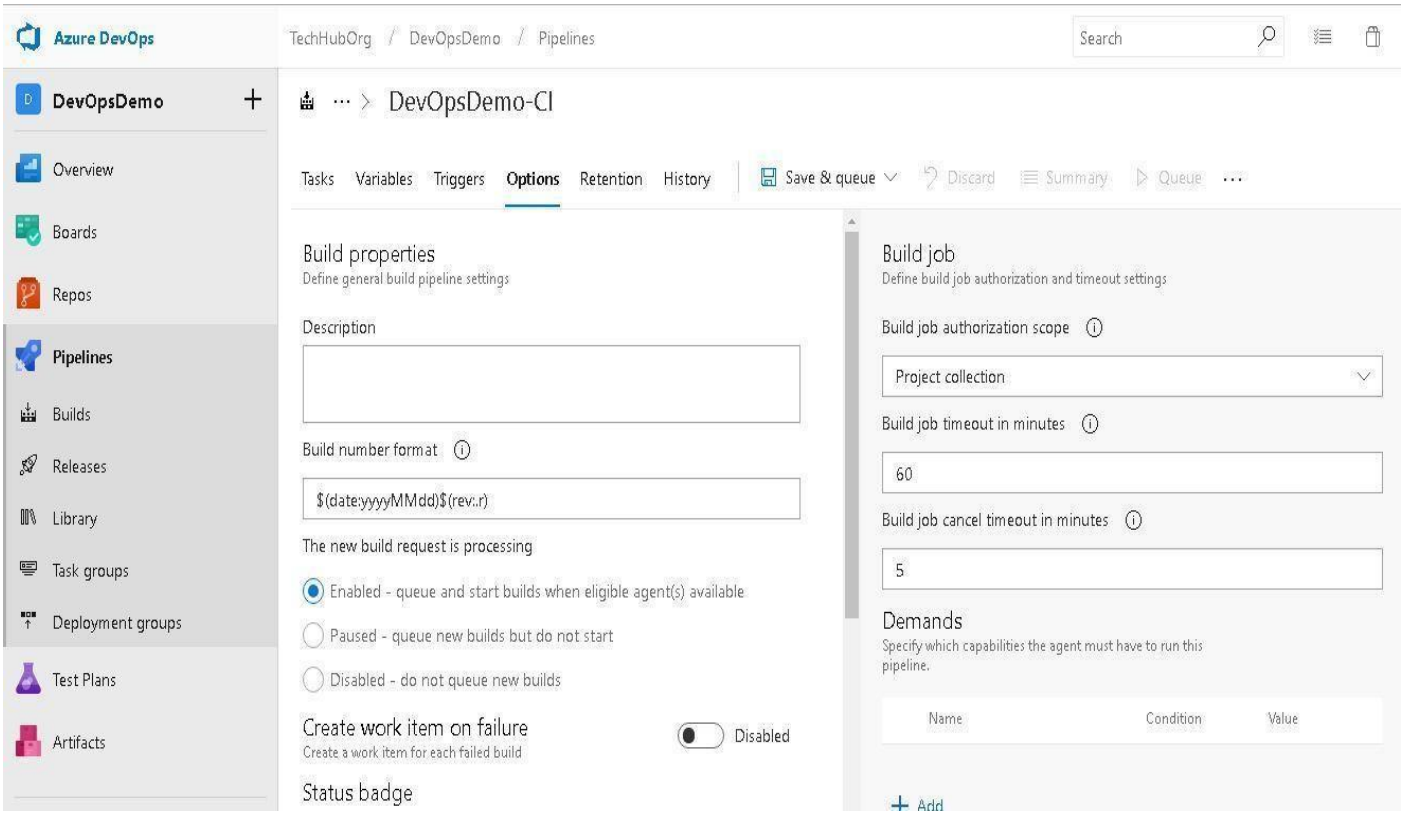
Move to the next tab, **'Triggers'**. It is a very important tab from where we can enable the **Continuous Integration**. So, just check the checkbox for **Enable Continuous Integration**. It means, as we will check in our code into repository, it will auto start the building code and creating the build artifact which will be deployed in the release cycle.

If we have multiple branches and we would like to filter any specific branch then we can define it into **Brach Filters** section.

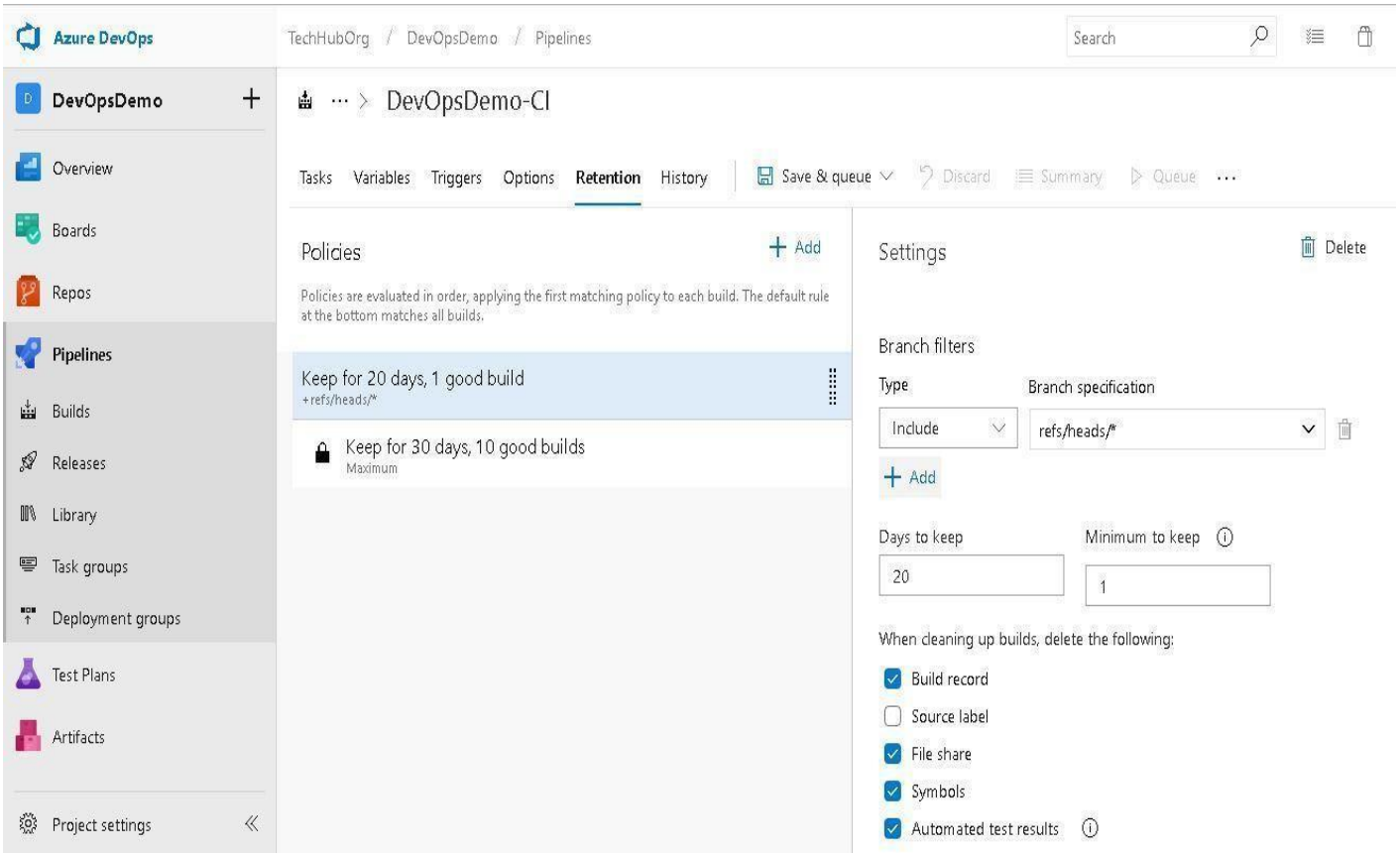


Let's move to **Options** tab, here we get the options to define the build version number using **'Build number format'** section. This one is in a default format, but we can modify it as per our requirement.

Create work item on failure is also a great feature which is used to create a new work item once the build fails. In a Build Job section, we can define the build job timeout in minutes.


Azure DevOps
TechHubOrg / DevOpsDemo / Pipelines
Search
DevOpsDemo
Overview
Boards
Repos
Pipelines
Builds
Releases
Library
Task groups
Deployment groups
Test Plans
Artifacts
DevOpsDemo-CI
Tasks
Variables
Triggers
Options
Retention
History
Save & queue
Discard
Summary
Queue
Build properties
Define general build pipeline settings
Description
Build number format
$(date:yyyyMMdd)$(rev:.r)
The new build request is processing
Enabled - queue and start builds when eligible agent(s) available
Paused - queue new builds but do not start
Disabled - do not queue new builds
Create work item on failure
Create a work item for each failed build
Disabled
Status badge
Build job
Define build job authorization and timeout settings
Build job authorization scope
Project collection
Build job timeout in minutes
60
Build job cancel timeout in minutes
5
Demands
Specify which capabilities the agent must have to run this pipeline.
Name
Condition
Value
Add

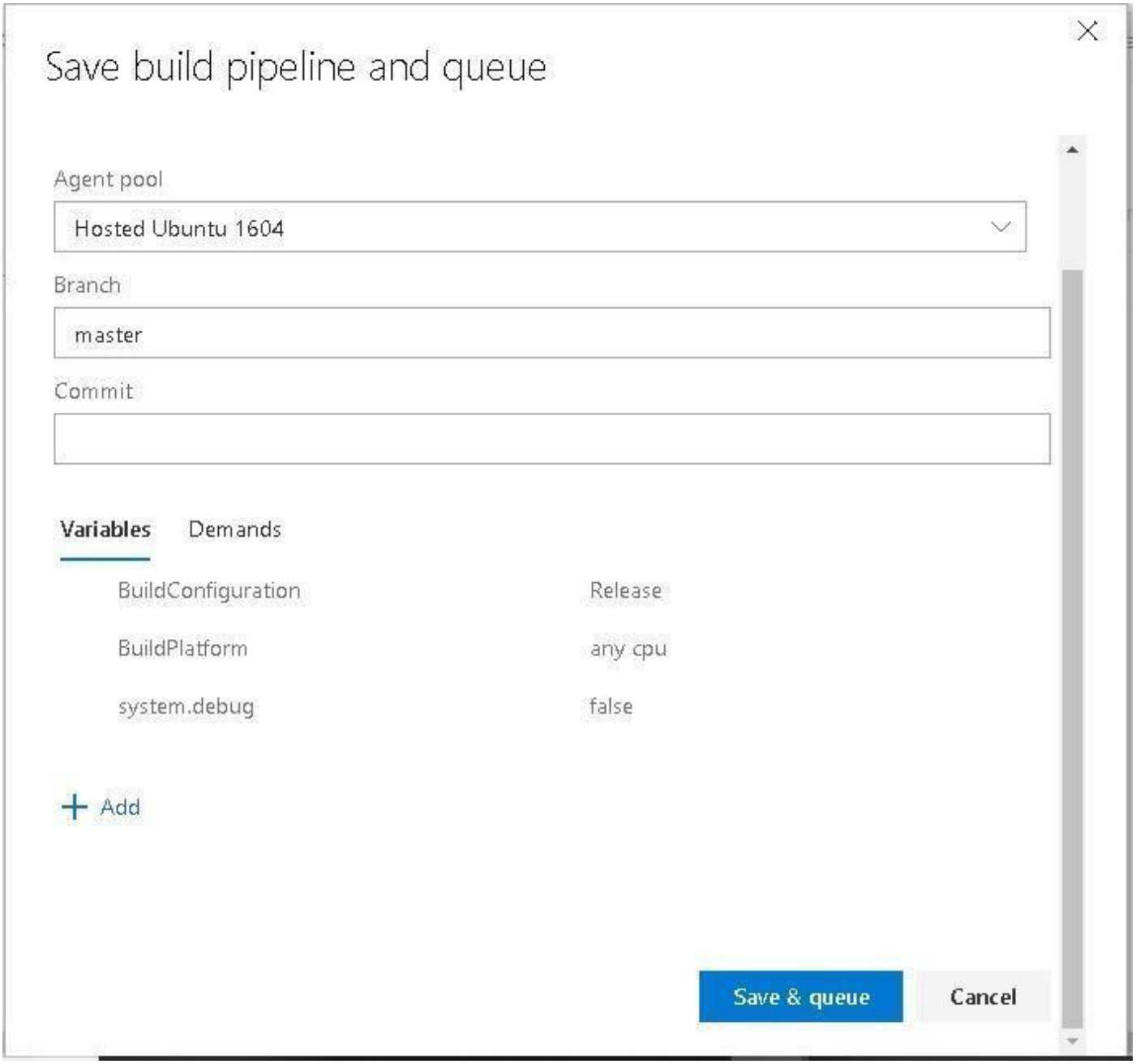
Now, time to move on to the **Retention** tab. Here we can define for how many days,we would like to keep the build information and how many good builds we would like to keep for those days. Everything can be set up here. Apart from these, we can also define what information should be deleted and what should not when clearing the build information.



Let's move to the **History** tab; here we are doing the setup of the **Azure DevOps build cycle**, so we don't have any history information. But in the future, when any build cycle performs then we can see all the history here.

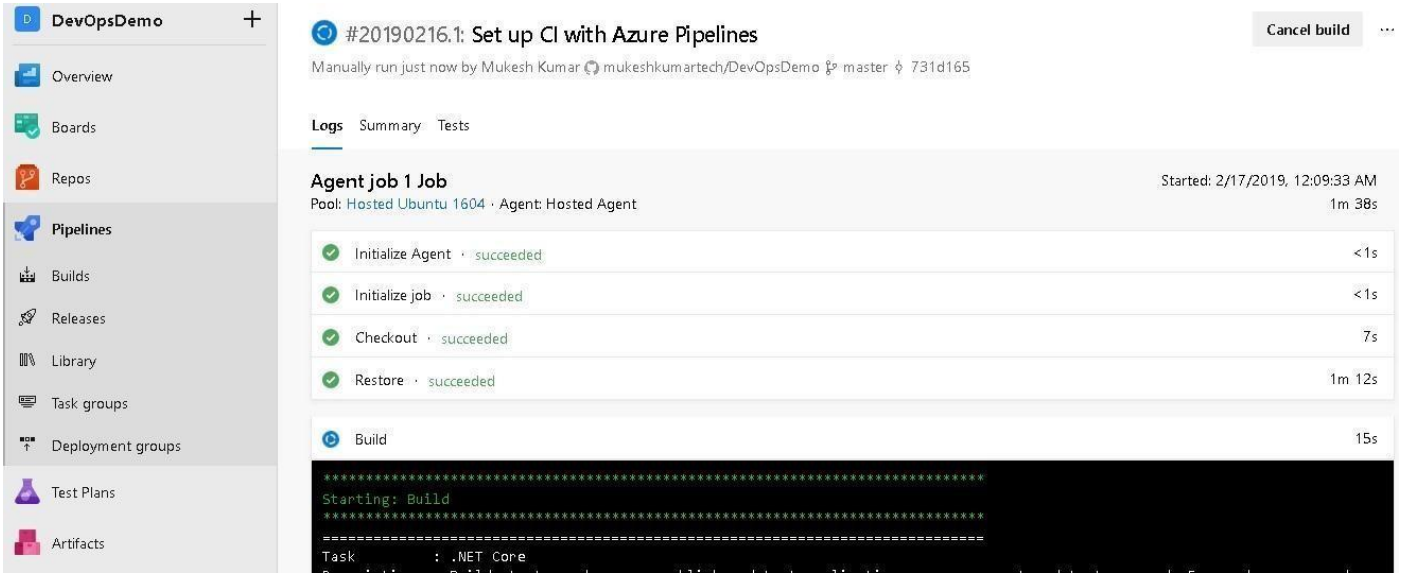
We have finished configuring the Azure DevOps Build Pipeline, now let's click to Save and Queue.

Once we will click to **Save and Queue** then it will ask for confirmation. So, just click to **Save and Queue** button once more as follows.



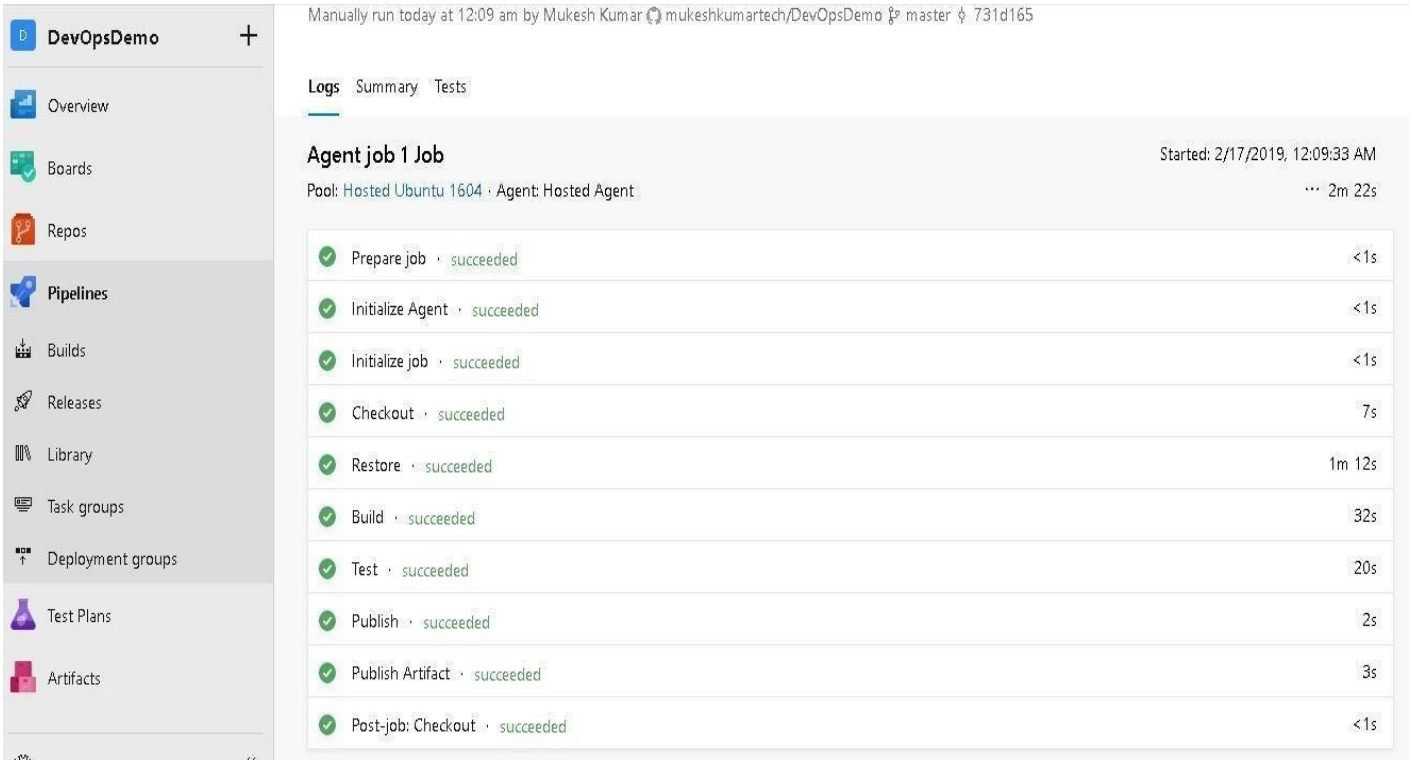
After clicking on Save and Queue button. first it will save the configuration for build pipeline and start queuing a build. As per the following image, we can see a message **'Build #20190216.1 has been queued'**.

Let's click on the **build # number** and we can see the log for jobs which are performing. As per the following images, build has already processed the Initialize Job, Checkout Job, Restoring the Code Job and now has moved on to Build. It is performing the Build.

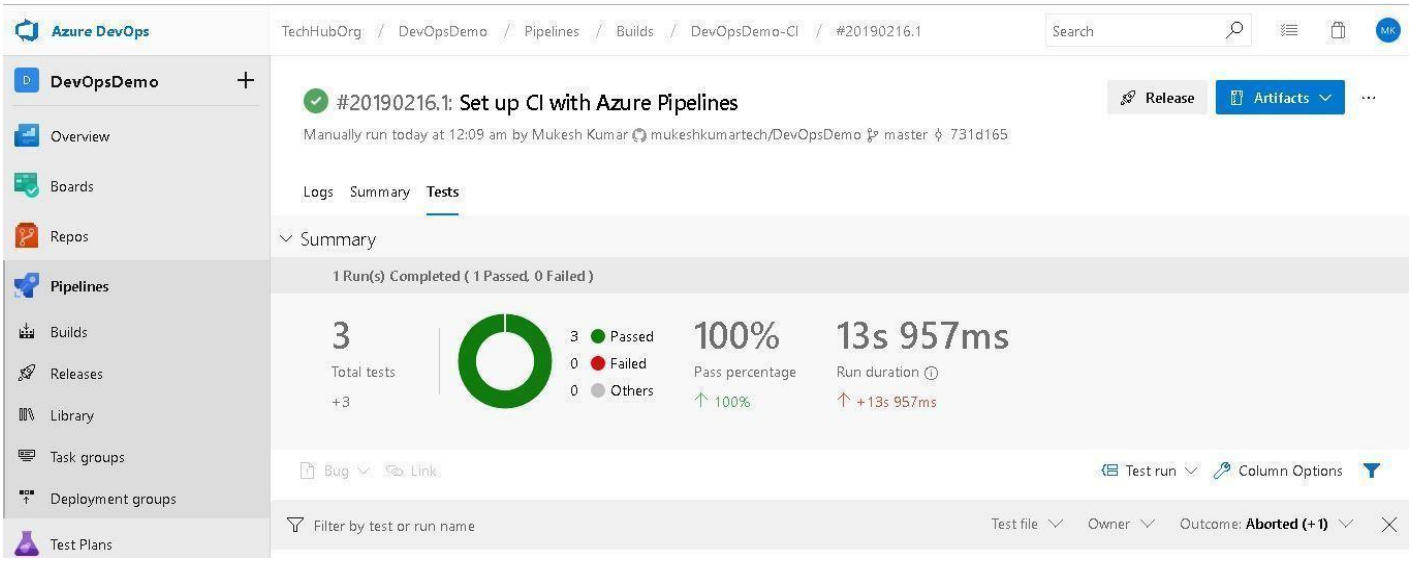


After build completes, it will move to Test the Unit Test Cases, which we have already defined at the time of creating the Build Pipeline.

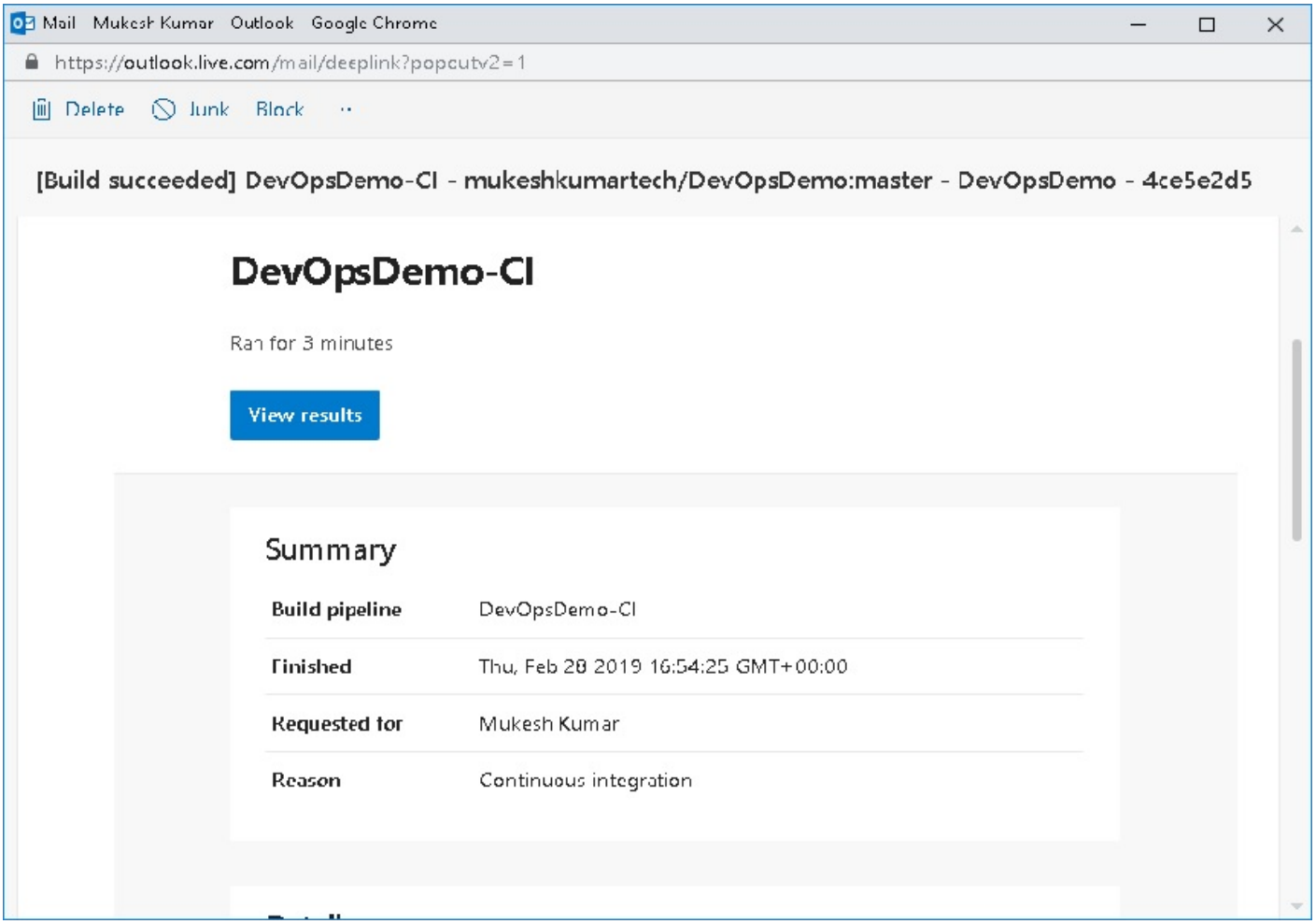
After few minutes, we can see that all jobs complete successfully as follows with **succeeded message.** If something is wrong with our code then it will fail with proper information.



Here, we have to confirm that **our Unit Test Cases** have executed and passed successfully or not. So, let's move to the Tests tab, here we can see the summary of executed Unit Test Cases. We had created 3 Unit Test Cases and all have been passed.



We will also learn about any build process that completes in the Azure DevOps Build pipeline through Email as follows.



So, we have performed several things as follows,

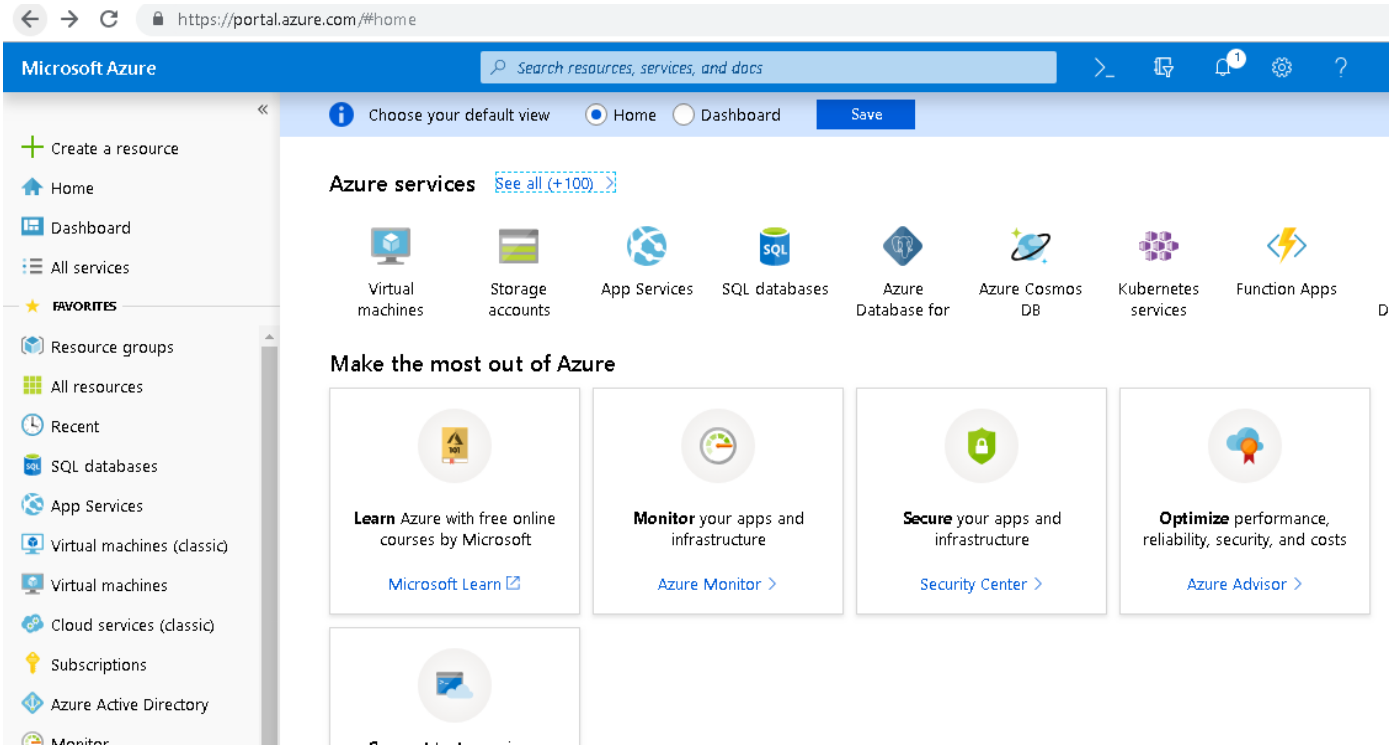
- Created the Azure DevOps Build Pipeline.
- Configured the Build Pipeline and enabled Continuous Integration.
- Saved the Build and Queued the Default Build.
- Azure DevOps Build has executed successfully.
- Unit Test Cases have passed successfully.

Chapter 7.Create Azure App Services

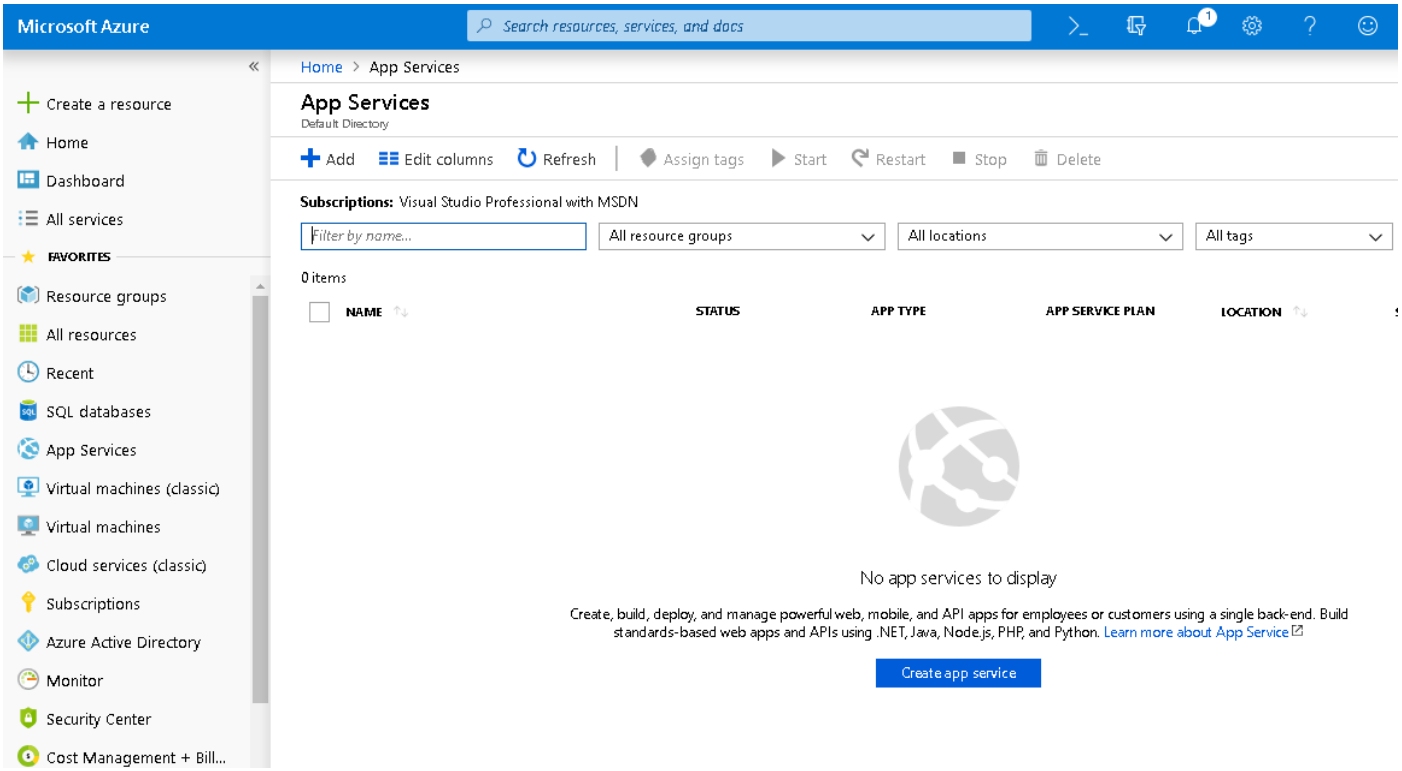
As we have finished creating the Build Artifact in Azure DevOps Build Pipeline which can be deployed, now we have to see how we can deploy it to a web app. But before moving to the Release lifecycle, first we will create the 3 web apps **(DEV, QA and PROD)** where we can deploy our artifact in different stages.

Before moving on, as we have explained above we will require an Azure Subscription. So, open **https://portal.azure.com** and log in with your credentials. Once we log in, we can see the default Dashboard for Azure Portal as follows. Here we have different options available to create the VM, Web App, Storage etc.

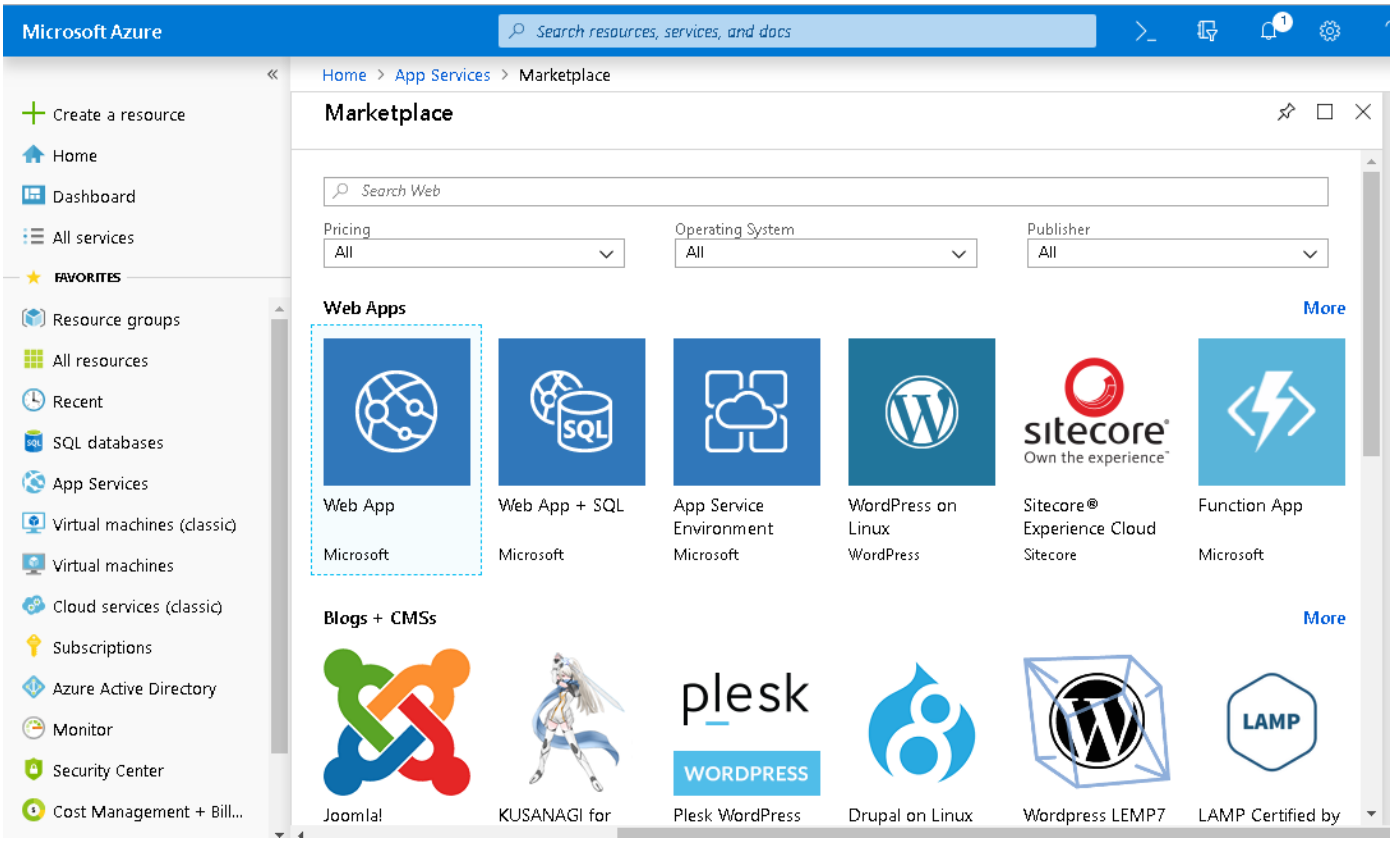
Azure App Service is a fast and simple way to create web apps using Java, Node, PHP or ASP.NET, as well as supporting custom language runtimes using Docker. A continuous integration and continuous deployment (CI/CD) pipeline that pushes each of your changes automatically to Azure app services allows you to deliver value to your customers faster.



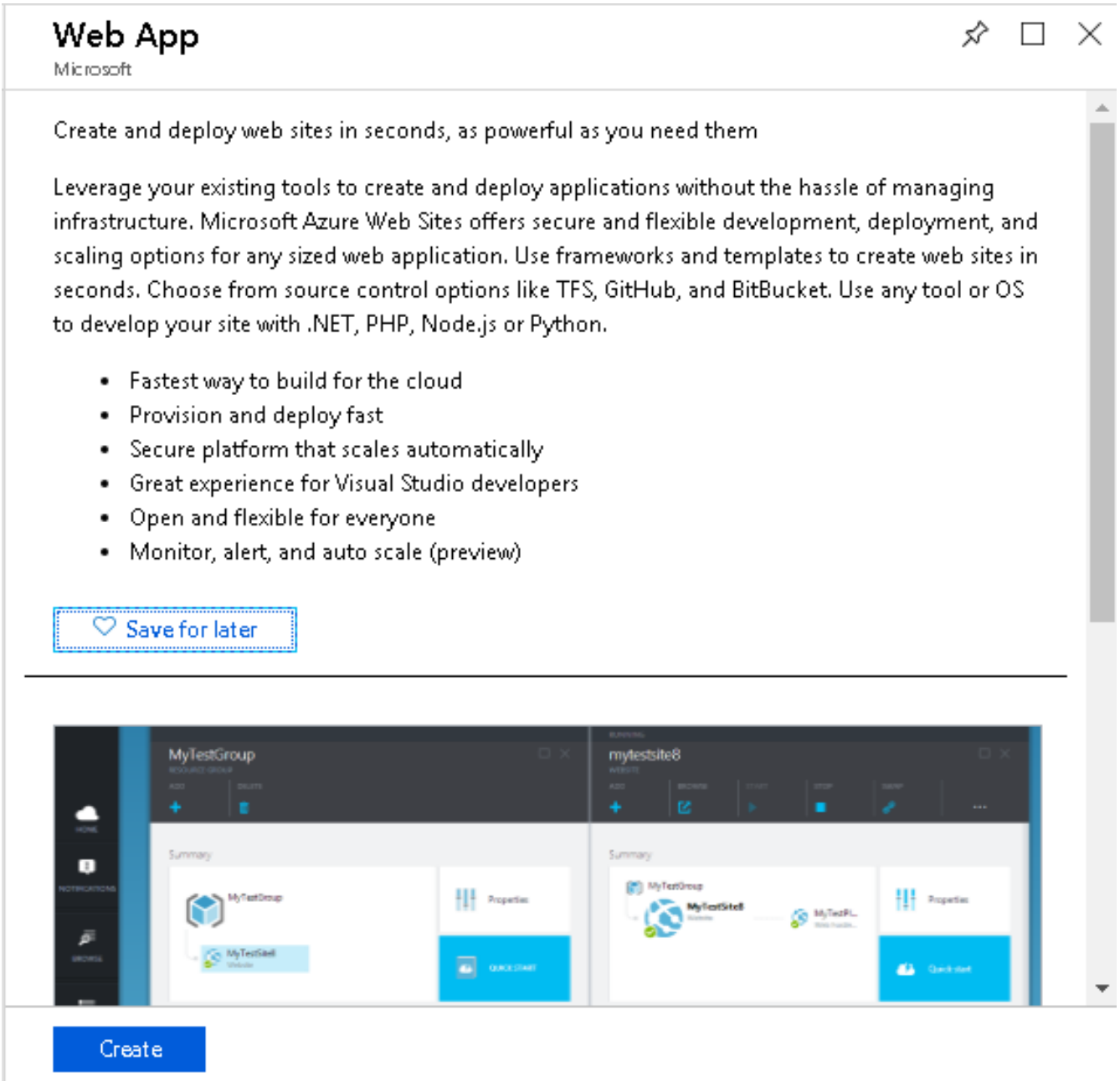
We are here to create 3 web apps. So for doing that let's click on **'App Services'** from the left panel just after SQL Database [See the above image]. It will open the App Services page from where we can create new App Services. We don't have any App Services in our bucket. So, let's create a new App Service by clicking on the button **'Create App Service'**.



The next screen will provide us with different kinds of templates available for creating the App Services. Here we will create a simple Web App. So, click on **Web App**.

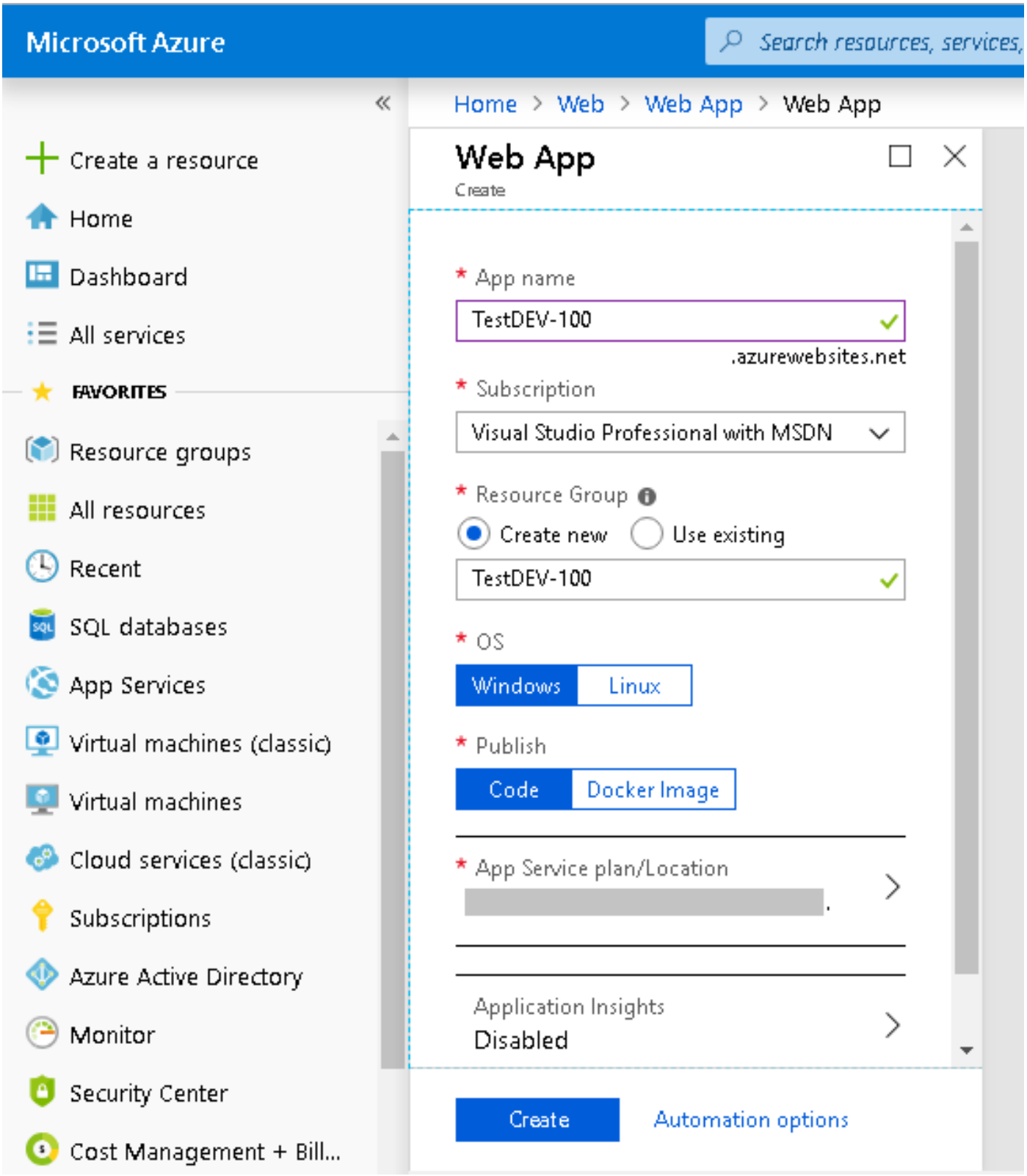


From the **Web App** section, we have to click on **Create**.



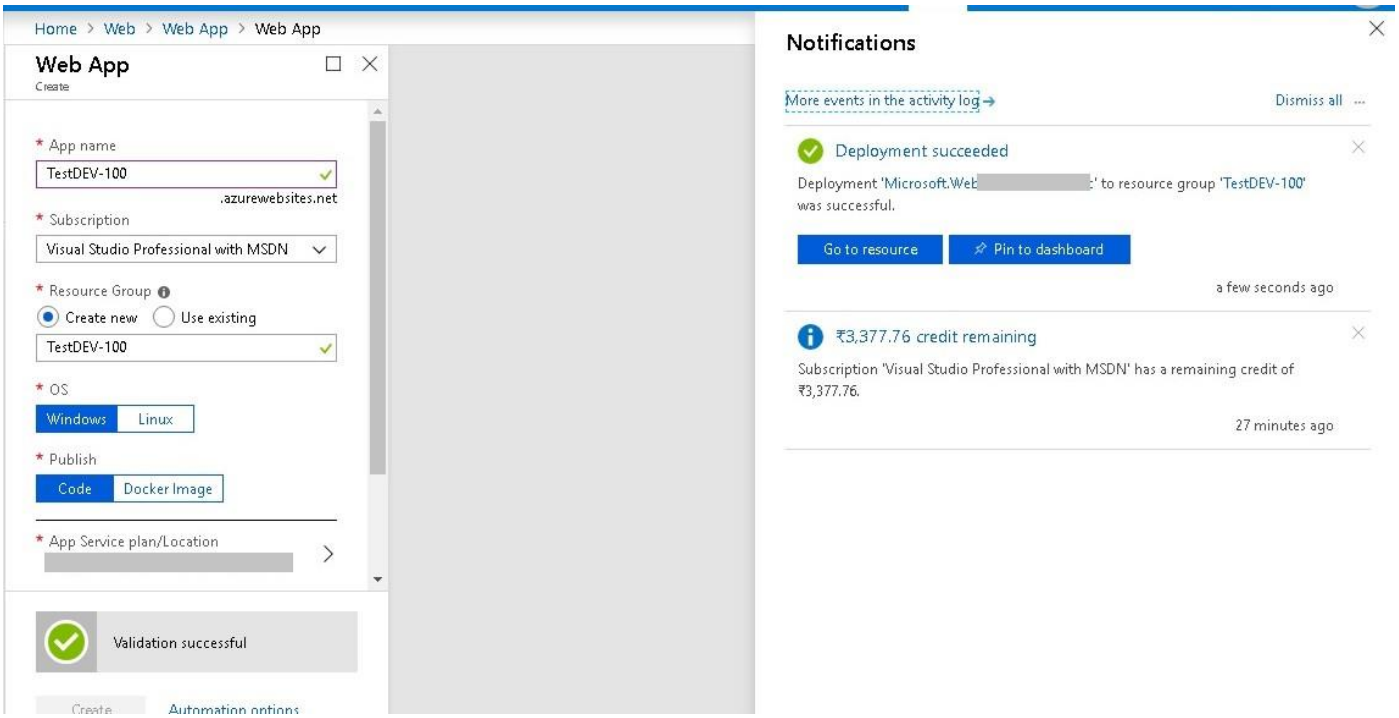
The next screen will ask for some information before creating the Web App. So, provide the name of a web app as **'TestDEV-100'**. The name should be consistent throughout the Azure Portal. Next, we have to provide the **Azure Subscription**. Next, we have to provide the **Resource Group**. We can reuse this if we have already created it, but for this demonstration we are creating a new one.

The next option is **App Service Plan and Location**. It is used to track what we have used and how much we have to pay per used resource. For the rest of the options just keep the default and click on the Create button.



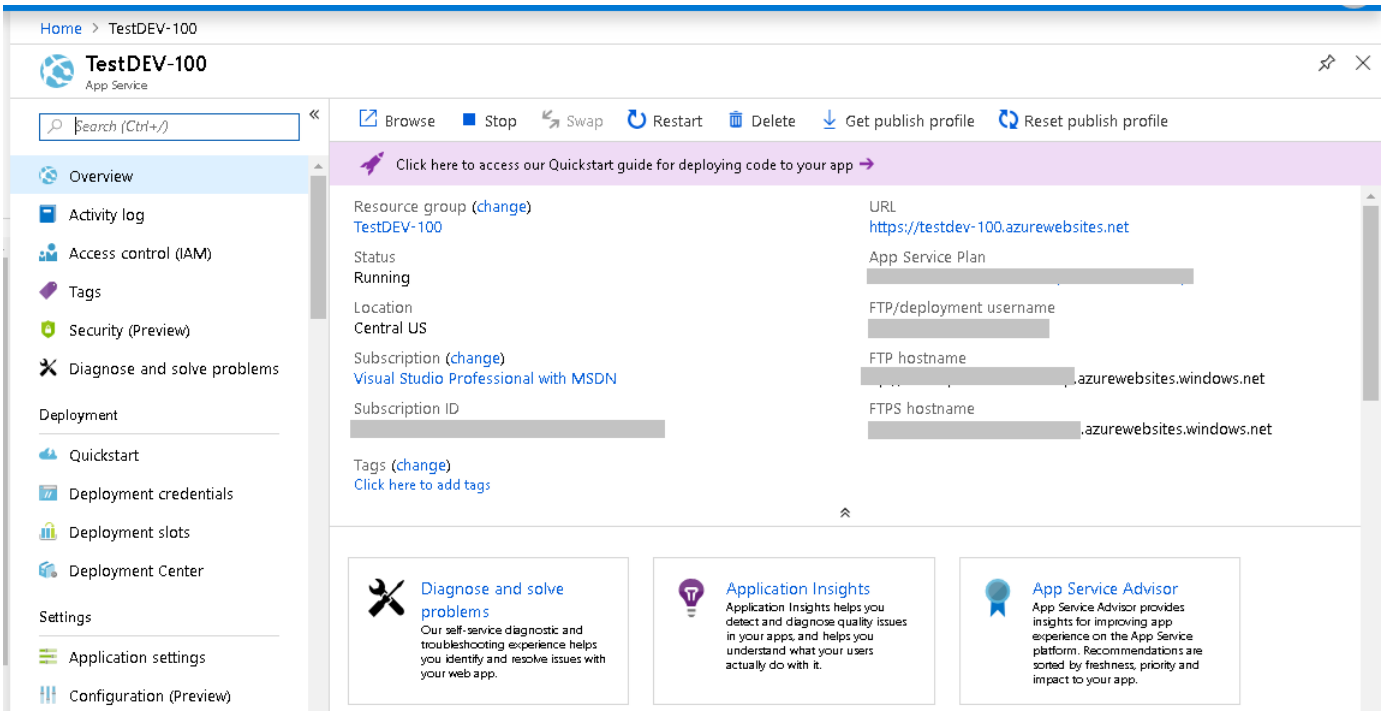
It will start the validating the information which we have provided. Once validation succeeds, it will start creating the Web App Service. It might take time to create the Web App Service depending on our network speed. We can see the progress of creating the Web App Service in the Notification bar.

Just click on the **Notification icon** and it will open the windows something like below. Here we will get notified that our resource **(Web App)** has been created and we have an option to **Go to Resource**.



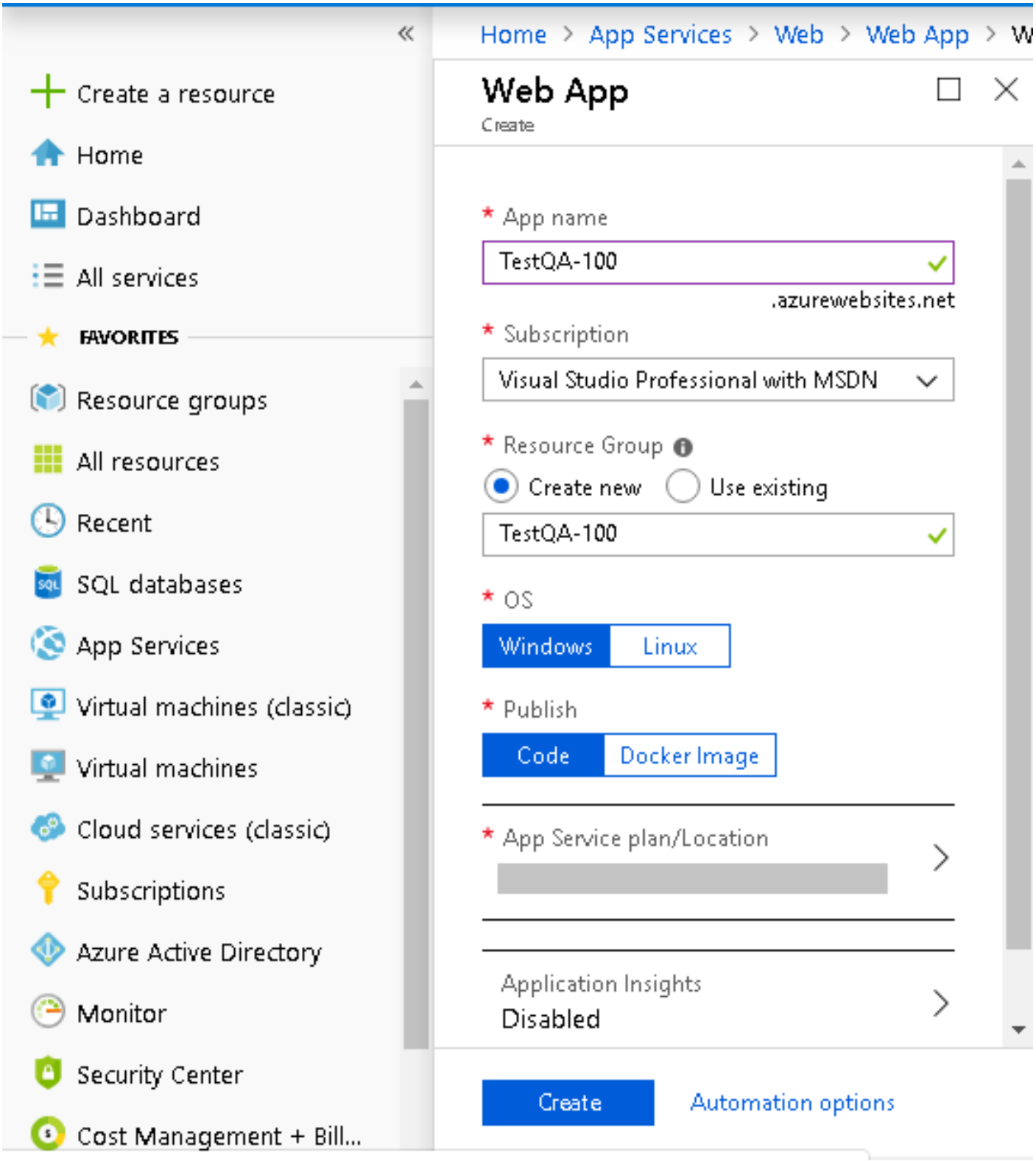
Click on **Go to Resource**. It will open the page where we can get all the information for newly created web apps **(TestDEV-100).** Here we can see the location of the resource where it was created, URL of web app for accessing it, and many more options for deployment like **username and password**.

At the top, we have couple more option for this Web App, like we can **stop** the Web App if it is running. We can **restart** it , we can **delete** it. If we would like we can get the **publish profile** which could be used for publishing the artifact on this web app.

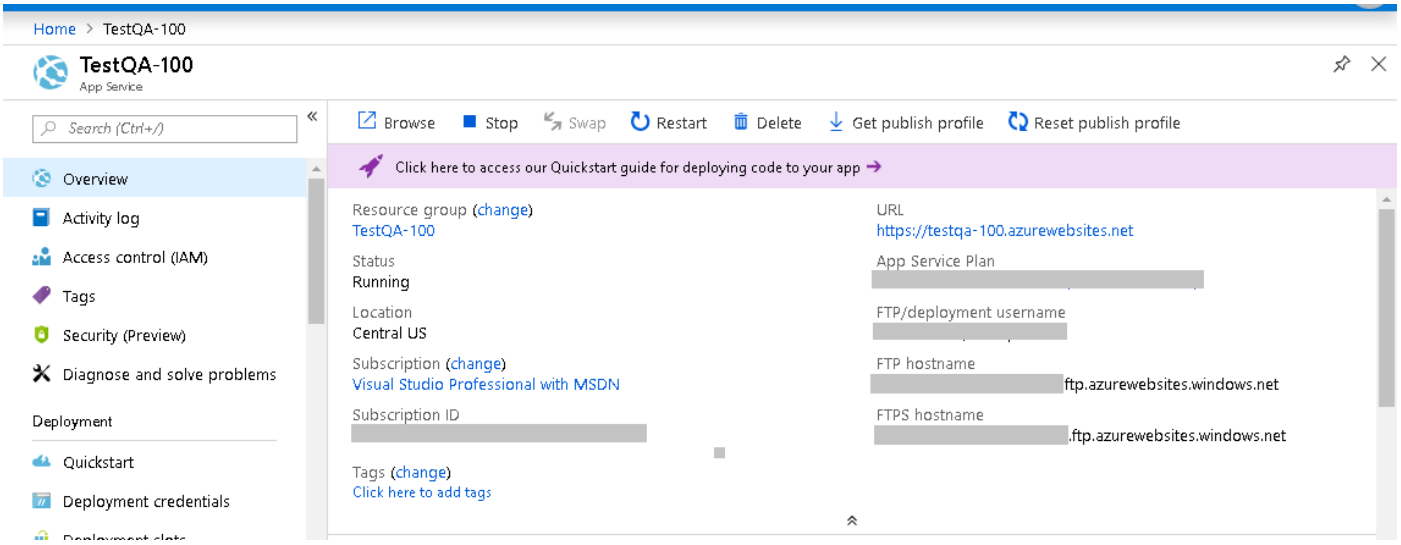


Let's move to **App Services** and follows the same process as we have followed for creating the **TestDEV-100** web app and create two more Web Apps like **TestQA-100 and TestPROD-100**.

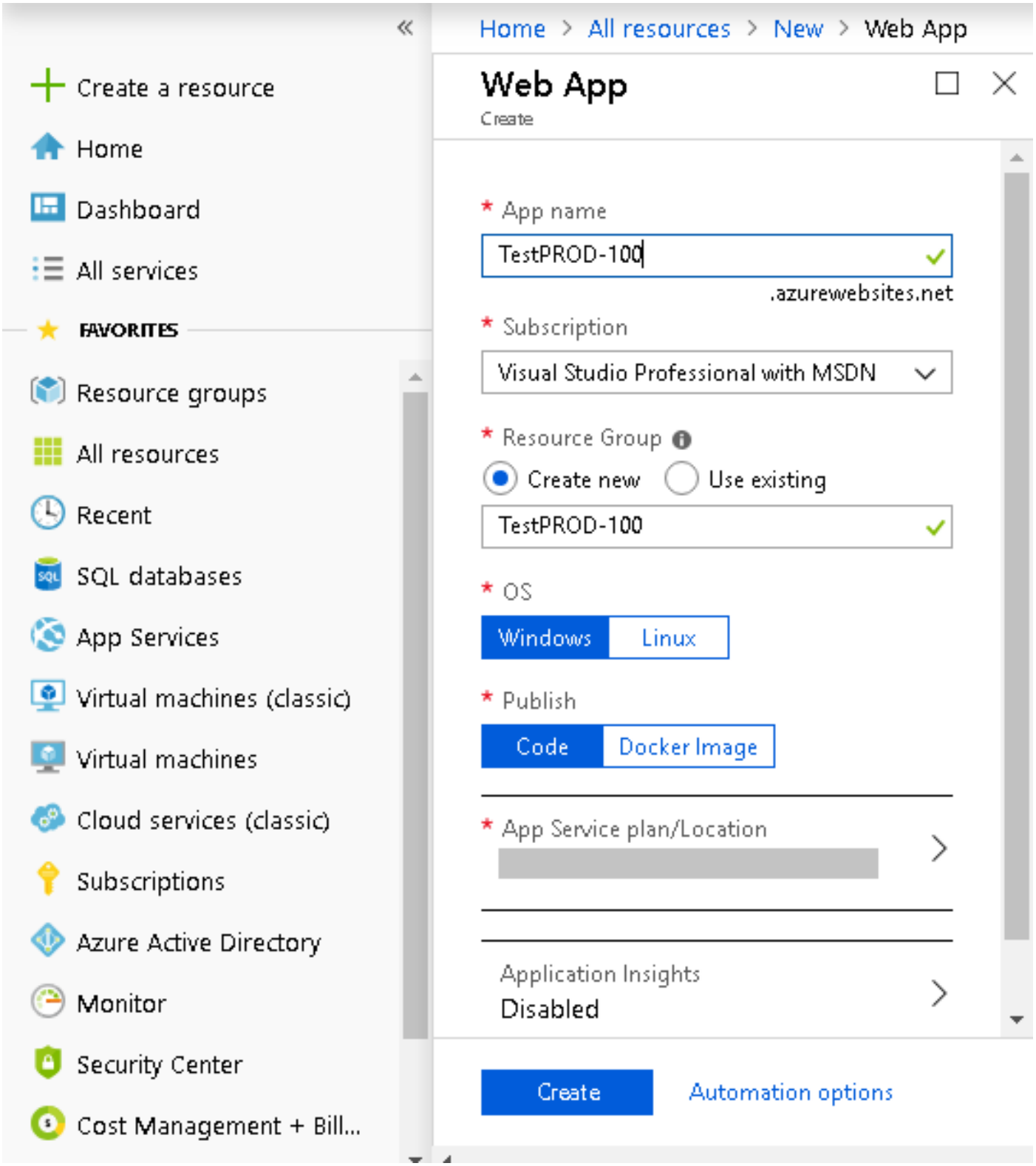
For the QA environment, we will create the **TestQA-100** Web app.



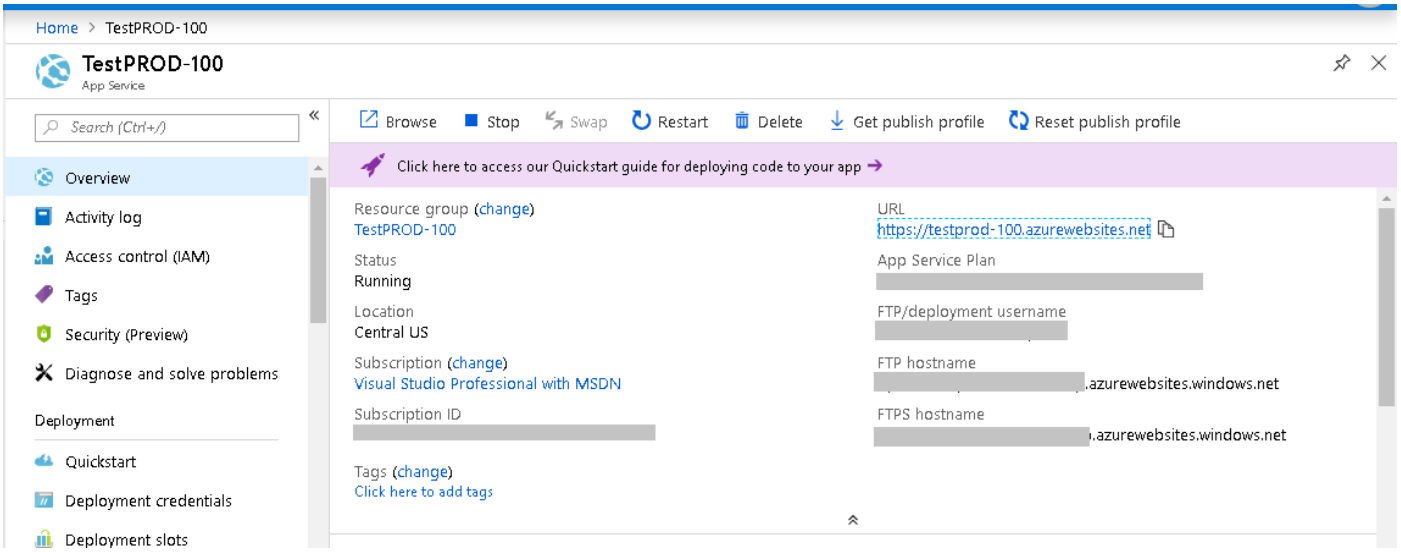
Once **TestQA-100** is created successfully, we will get the information as follows.



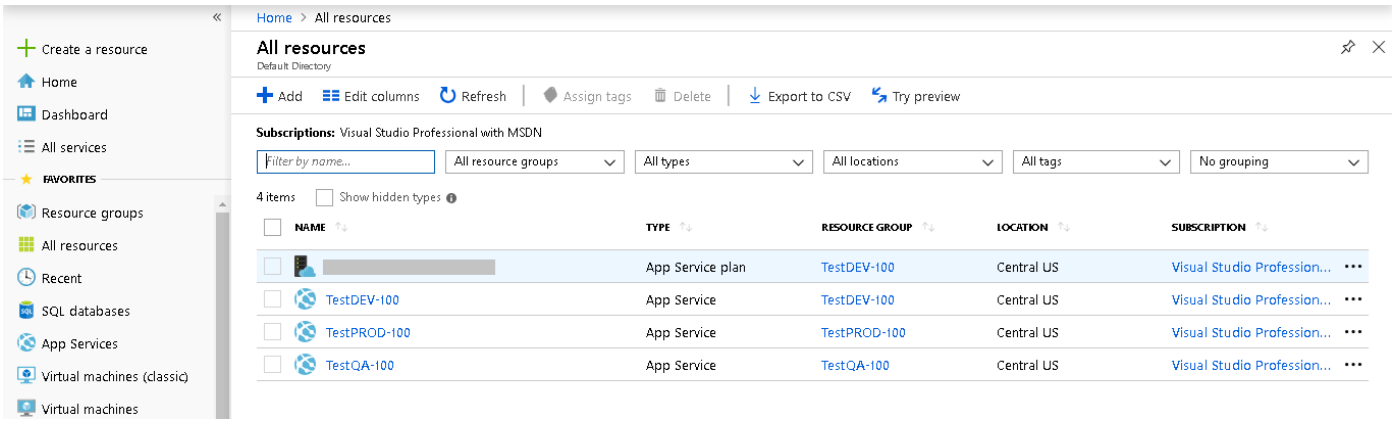
For the **PRDO** environment, we will create the **TestPROD-100** Web app.



Once **TestPROD-100** web app has been created successfully, we will get all information about this resource as follows.



Now, let's move to the **Resouce page**. Here we can find all the newly-created App Services. We have 3 web apps for the **DEV, QA and PROD environment** respectively. Apart from that we have one **App Service Plan**, under which all app services will run.



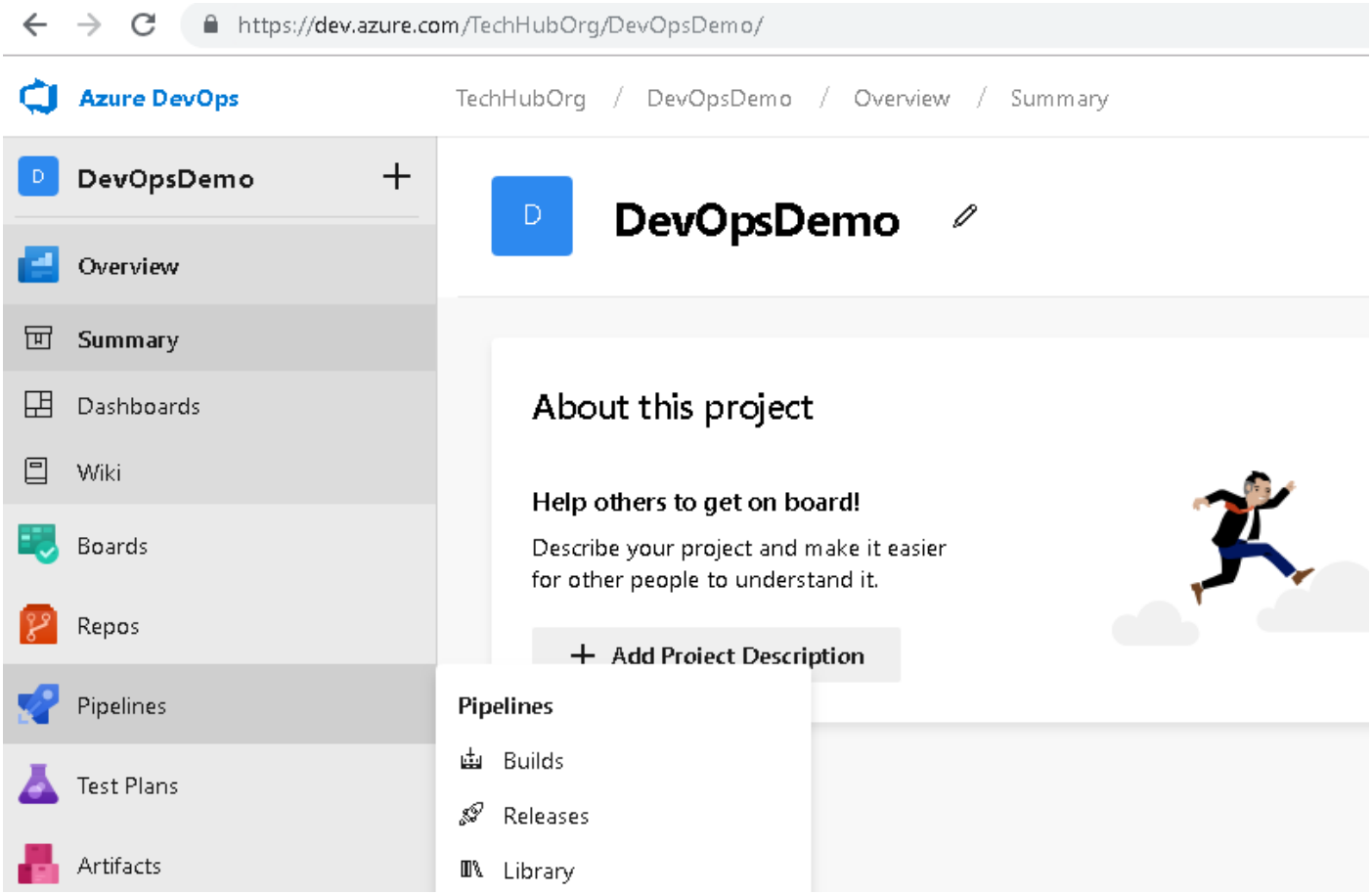
So far, we have created the **Azure DevOps Build Pipeline** successfully and were able to create the build artifact. Apart from this, we have created 3 web apps for different environments like DEV, QA and PROD. These app services will be used in Azure DevOps Release Pipeline for deployments.

Chapter 8.Continuous Delivery

Let's move on to our **DevOpsDemo** project in Azure DevOps using the following URL.

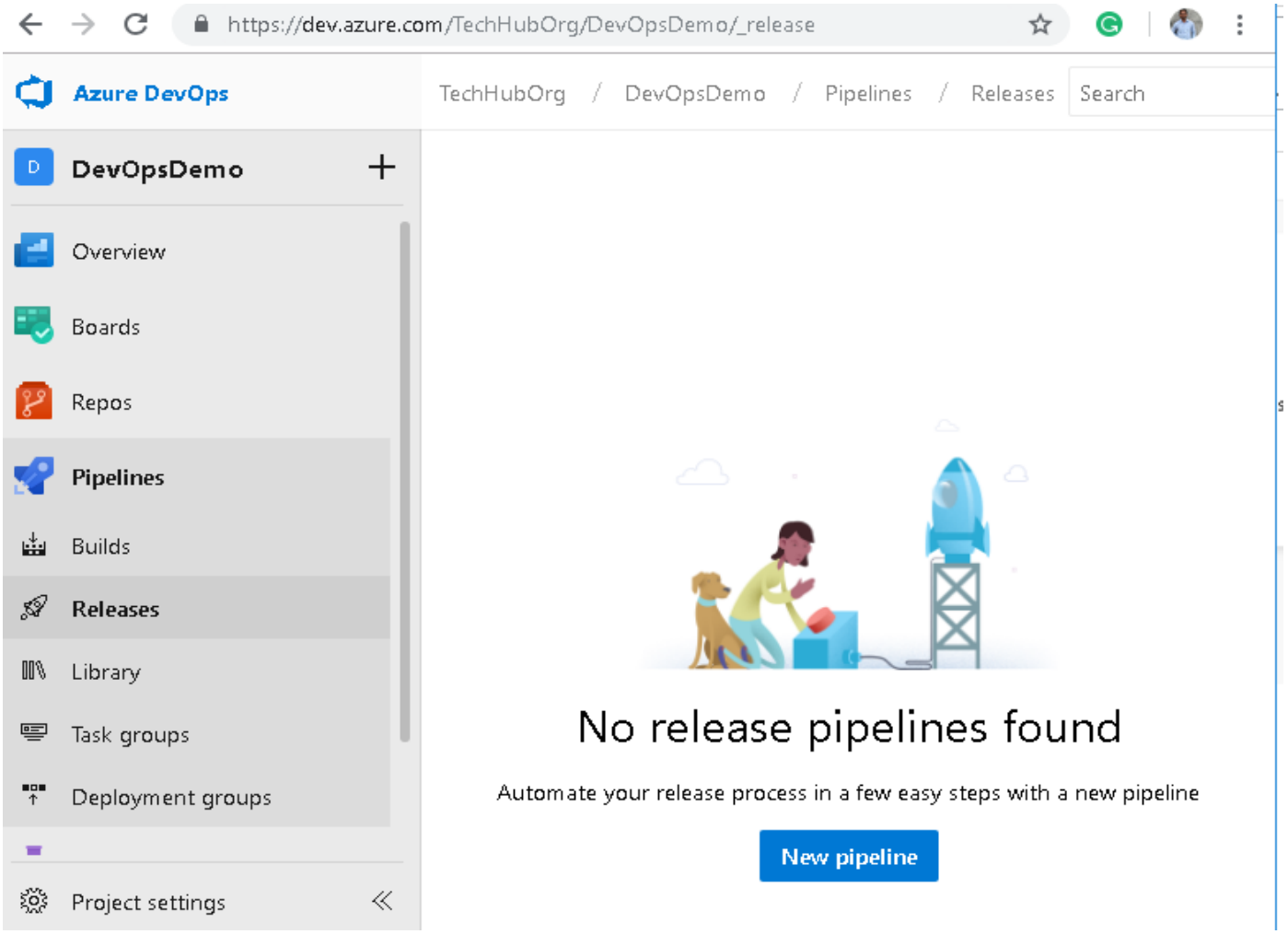
https://dev.azure.com/TechHubOrg/DevOpsDemo.

Click to **Pipeline** and choose **Releases** for creating new **Azure DevOps Release Pipeline**.



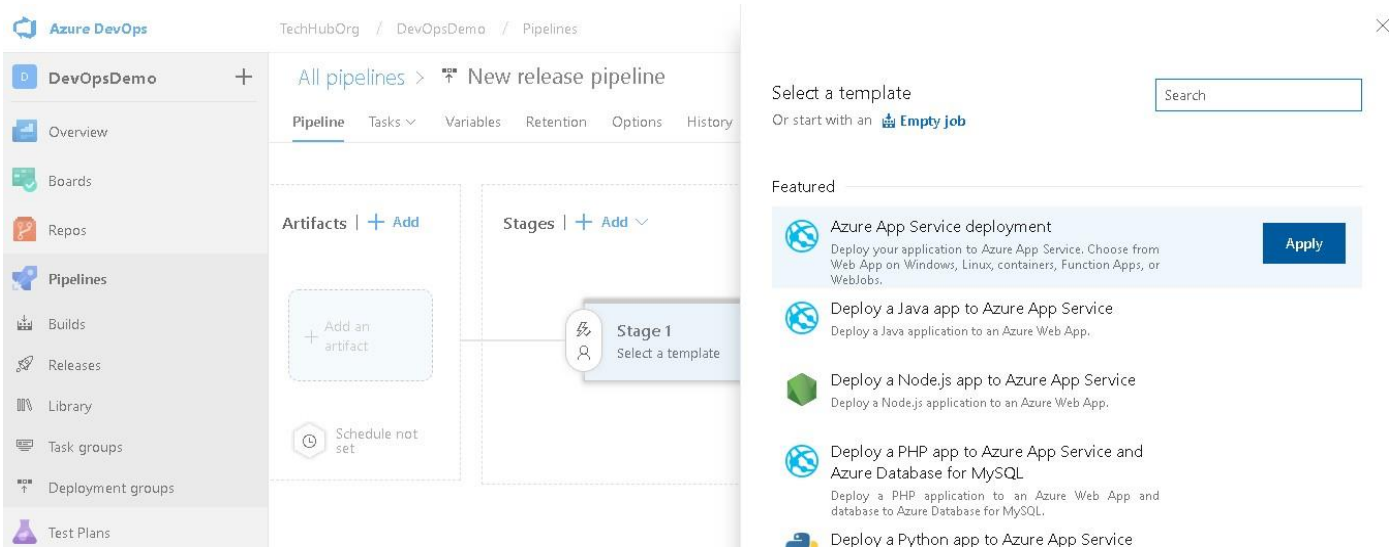
The next screen will show all the release pipelines if we have any. But we haven't created any so far. So, let's create a new one and click on **New Pipeline**.

8.1 Create Dev Stage

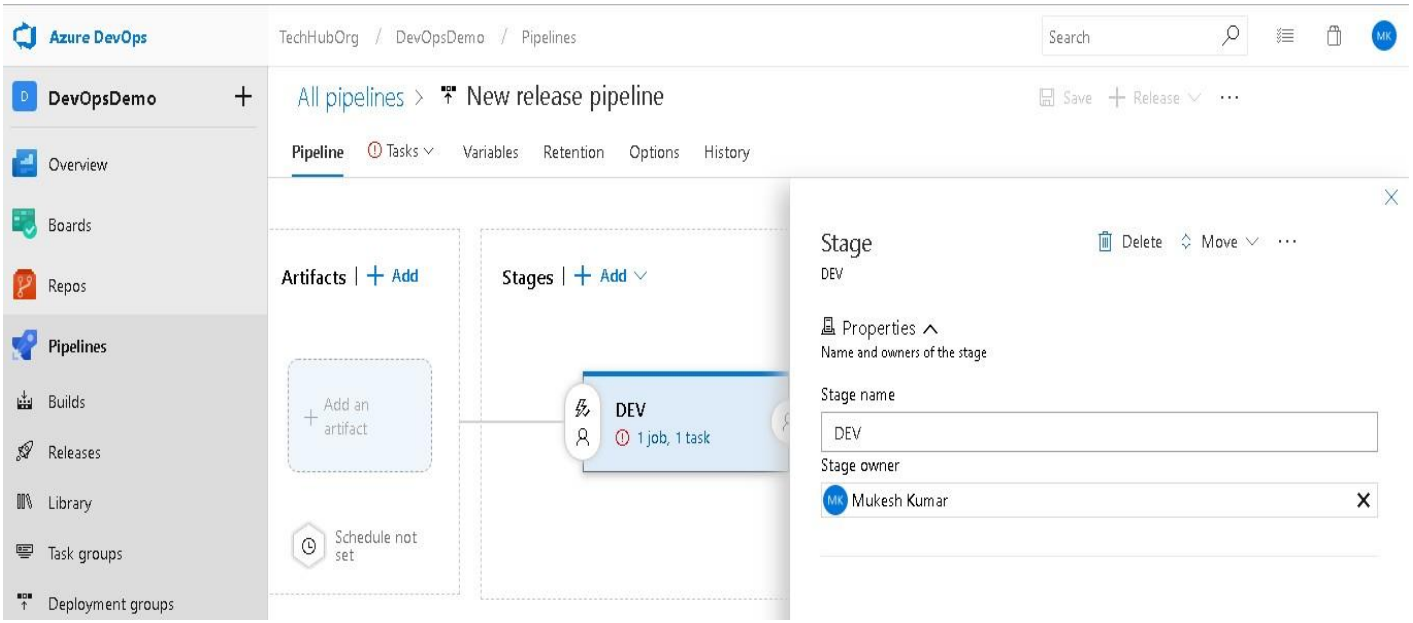


The next screen will ask to configure the **Release Pipeline**, such as what service we will use, which Artifact we will use for deployment, what template we will use for deployment etc. So, select a template **as Azure App Service Deployment** and click on **Apply**. As we have already created 3 app services above for deployment for different environments like DEV, QA and PRDO, we will choose TestDEV-100 for this DEV stage.

By default the name of the Release Pipeline is **'New Release Pipeline'**. We will change it later.

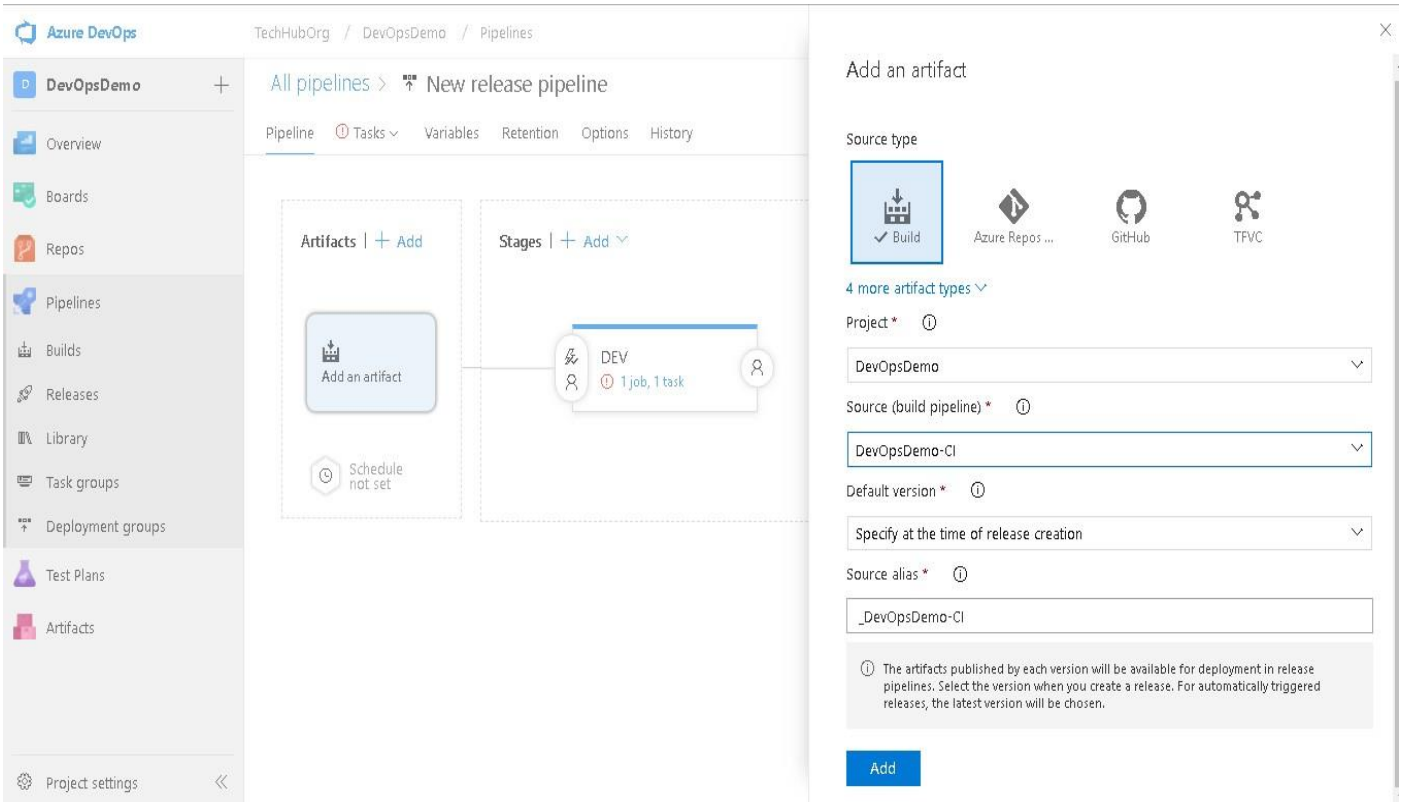


After selecting the template as **Azure App Service Deployment**, it will ask to provide the **name** for this stage. This is our first stage, so provide the Stage Name as '**DEV'** and click on the close **(X)** button in the right top corner **(Only for closing this dialog, don't close the page itself.).**

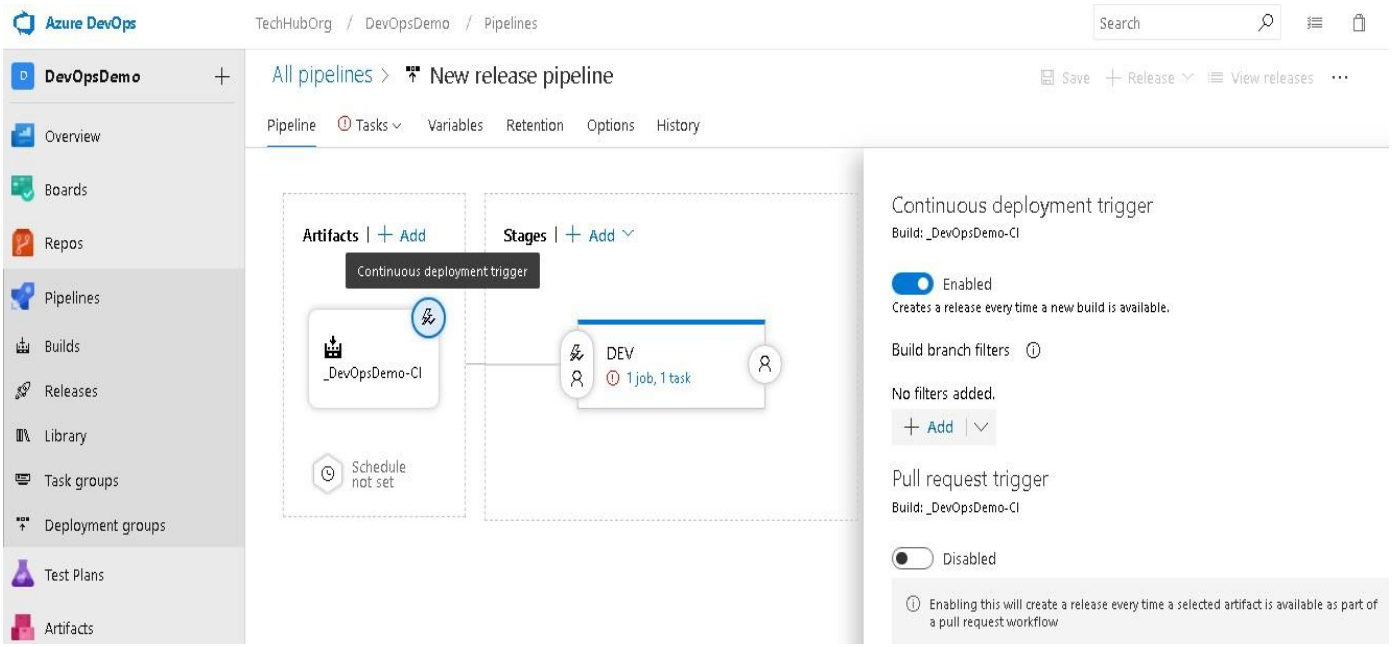


Once we close the stage dialog then we will get the screen something like below. Here, we have two sections for now. The first one is the **Artifact** section. Here we will **add the Artifact** which is created in **Azure DevOps Build Pipeline**. Another section is the **Stage** section. We could have multiple stages like DEV, QA, PRDO etc or many more as per the requirement.

Click on **Add an Artifact**. It will open a popup a dialog window in the right side as follows. Here we will configure the Artifact. So, first select **the Source Type as Build**, because we will take the Artifact for deployment from the Build Pipeline. Next select the **name of the project as DevOpsDemo**. Next we have to select the **Source (Build Pipeline).** As we have Build Pipeline as **'DevOpsDemo-CI'**, select it. For the rest of the options keep the default and click on **Add**.



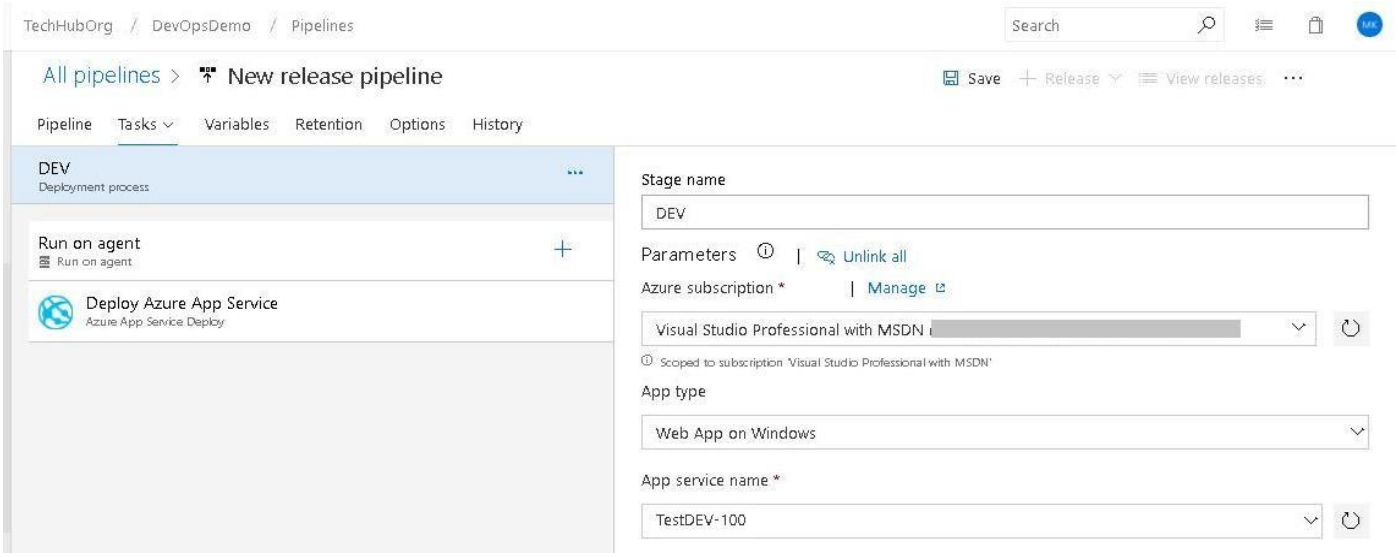
Once we click on **Add**, it will add the **'_DevOpsDemo-CI'** artifact. Now, we have available the **Artifact** which can be deployed to any environment. So, enable the **Continuous Deployment**. Click on the **Continuous Deployment Trigger** as we can see in the below image. It will open the Continuous Deployment Trigger dialog window in the right. **Enable the Continuous Deployment Trigger** which will create a new release every time a new build is available. Keep all other options as default.



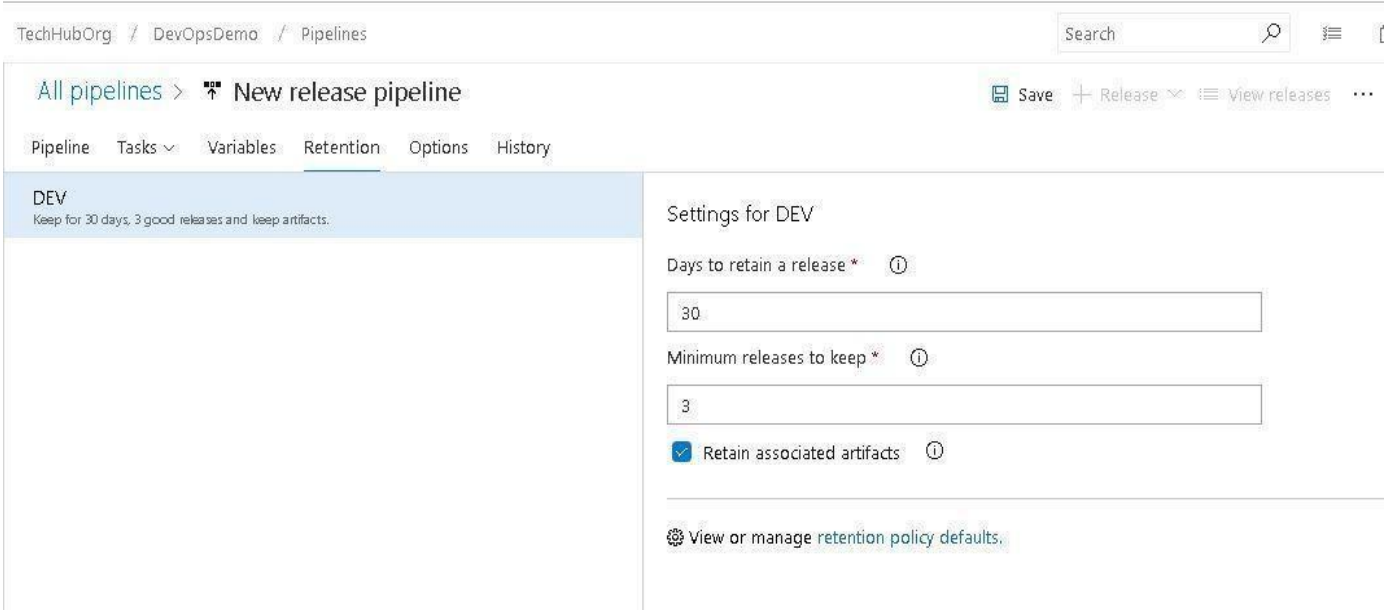
Close the **Continuous Deployment Trigger** dialog window. So we have finished the setup of the Continuous Deployment Trigger, if a new build is there then it will deploy to its respective environment.

Now let's move to the Stages section and click to **'1 Job, 1 Task'** in **DEV** stage for configuring the **DEV** environment so that if a new build is there it auto deploys on the DEV environment.

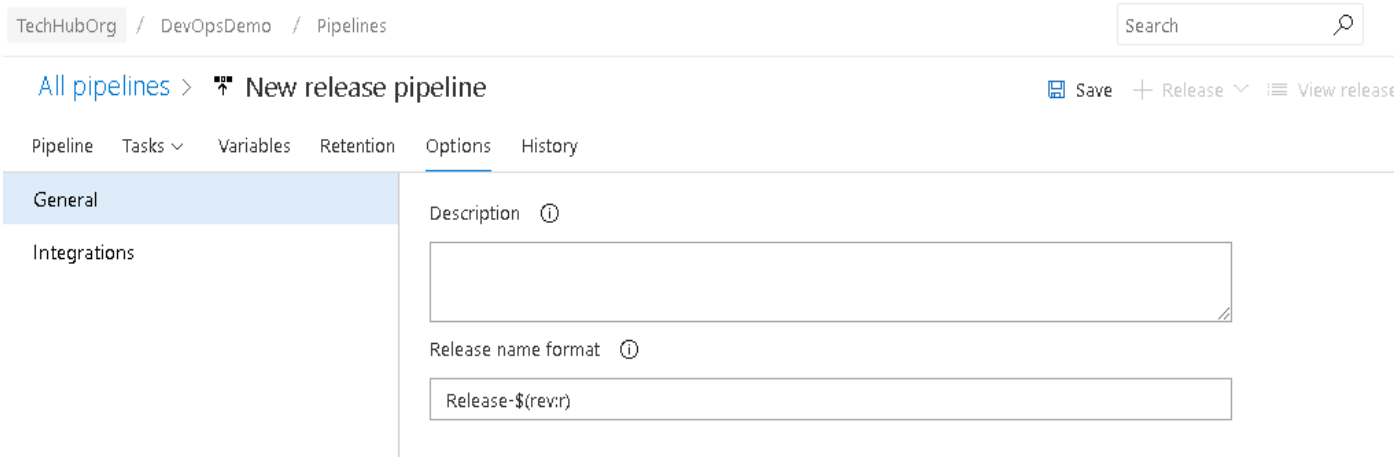
Each stage in the Release pipeline has its own configuration. We have to move to the **Tasks** tab, here we can provide or modify information like **stage name, Azure Subscription** etc. The most important configuration is **App Service name**. We will define the **TestDEV-100 web app in App Service Name** so that after building the application, it can deploy on DEV server first.



We don't have to do it in the **Variables** tab section, let it be the default value and move to the **Retention** tab. Here we can do the setting for the **DEV** environment like how many days we would like to retain the release information and how much release information we want to keep.

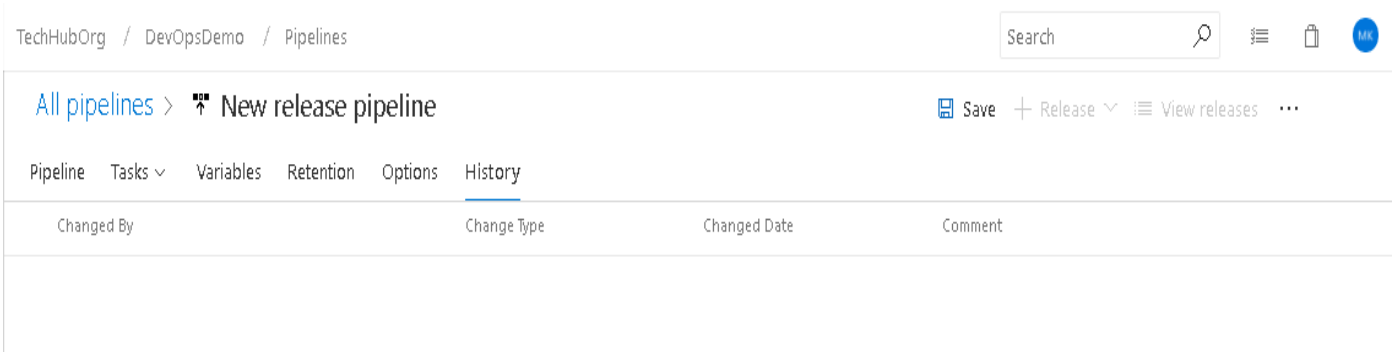


Now let's move to the **Options** tab, here we can provide the information about this Release, such as the **description of the Release** and what will be **format of the Release Name.** By default, it will create a default format for the release name but it can be modified.



The next tab is the **History** tab. Here for now, we will not find anything. It will be empty because we haven't run any release cycle yet.

After configuring all tabs as per the requirements, it's time to **save** the information. So, click on **Save**.

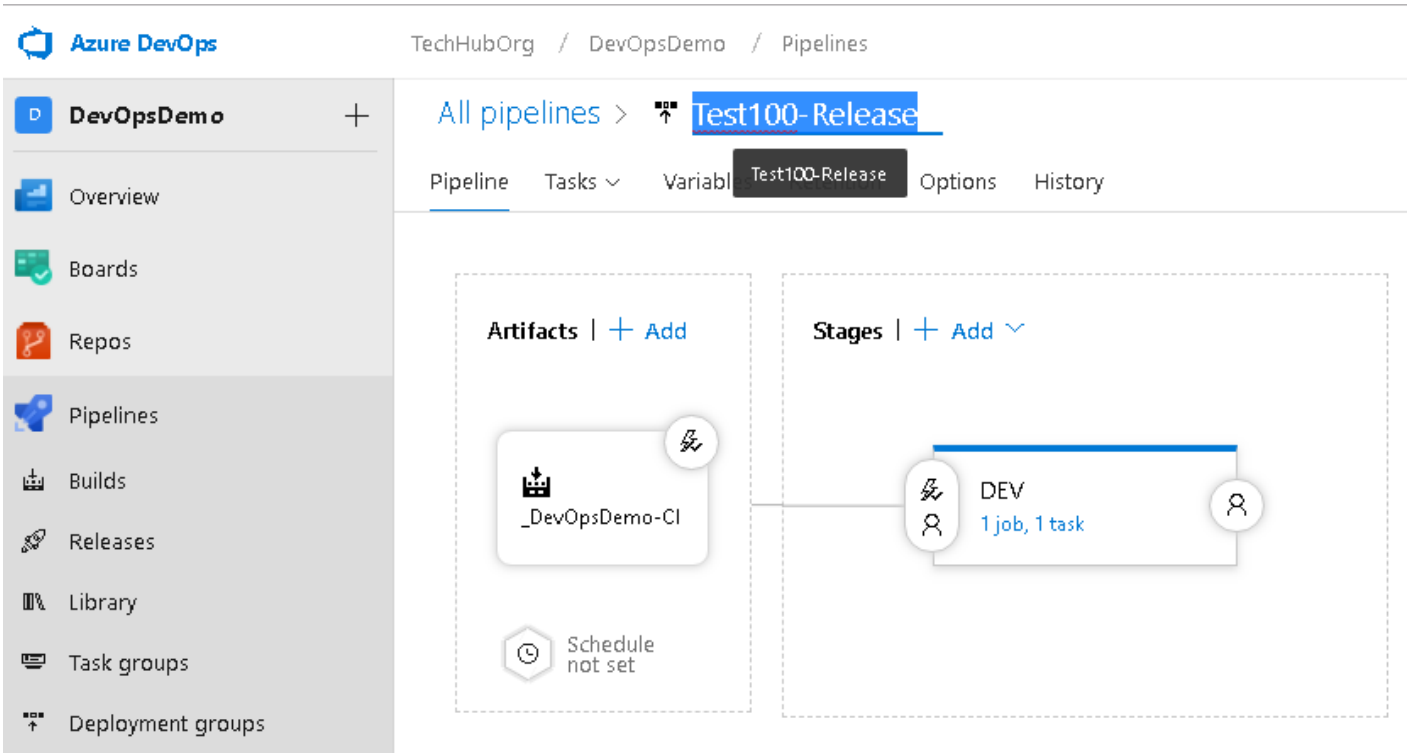


After creating and saving the release pipeline, we can see our configured Azure DevOps Release pipeline as follows.

We can see the Release pipeline anytime from **Pipeline>Releases**. Here we can see all the available Release pipelines. If we would like to **Edit** and then we can **edit** also.

Click on the **Edit** button **(as shown in above image)** for **editing the Release Pipeline**. Once we click to edit the release pipeline, it will open the same page from where we can define the artifact and other stage information.

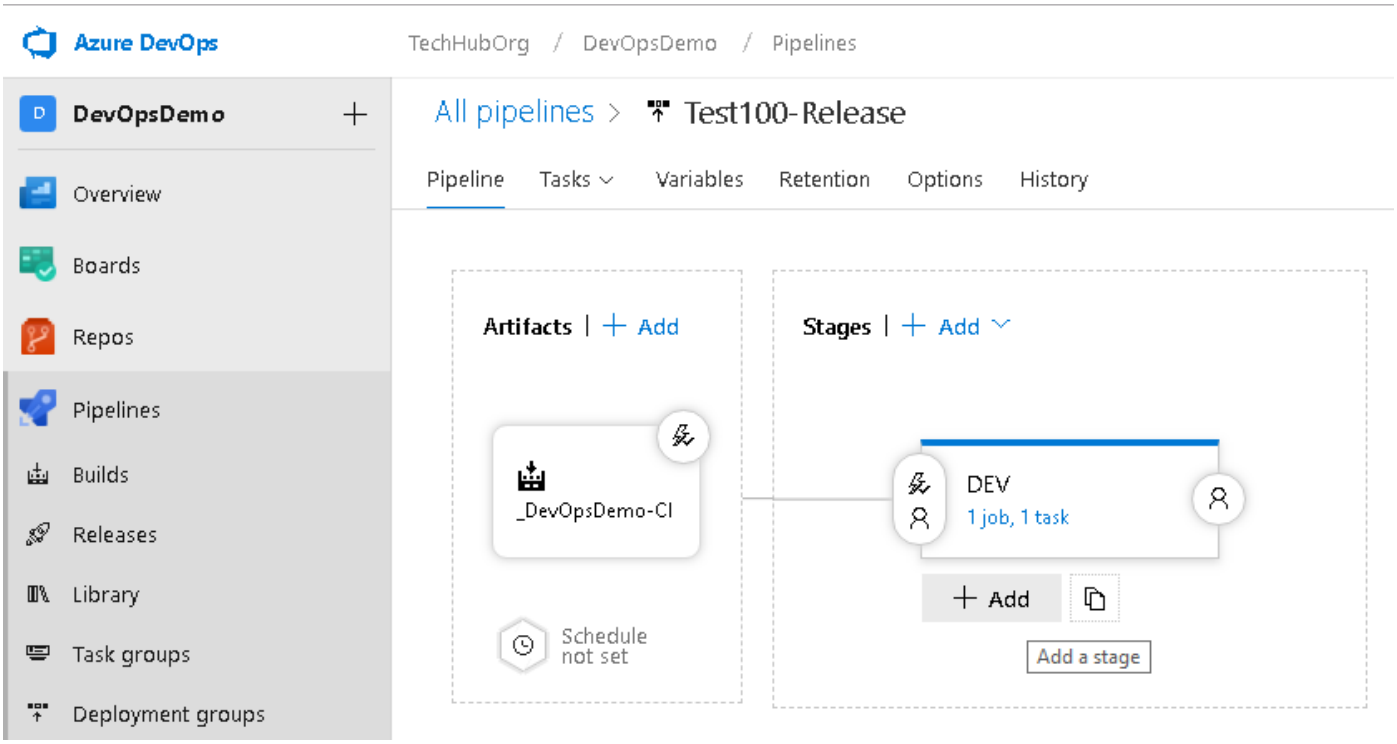
So, first modify the name of the Release Pipeline. Click on the **'New release pipeline'** text just after All pipelines and rename it with **'Test100-Release'**. So, we have changed the name of the **Azure DevOps Release Pipeline** name.



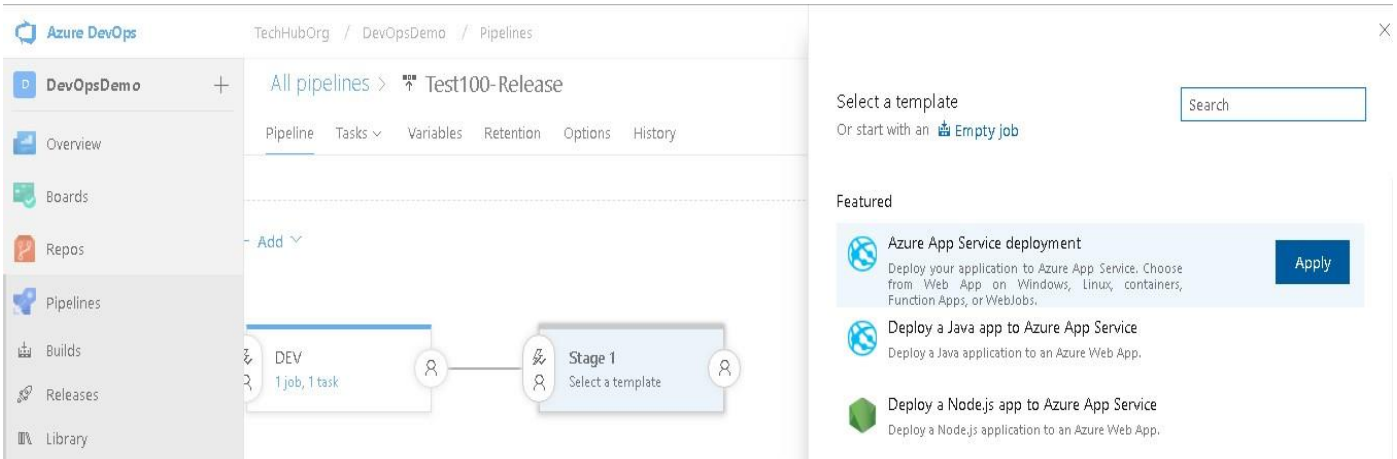
8.2 Create QA Stage

We have already set up the **DEV** environment. If a new build is available it will deploy to the **Dev (TestDEV-100 Web App)** environment. But let's add one more stage for **QA (TestQA-100).**

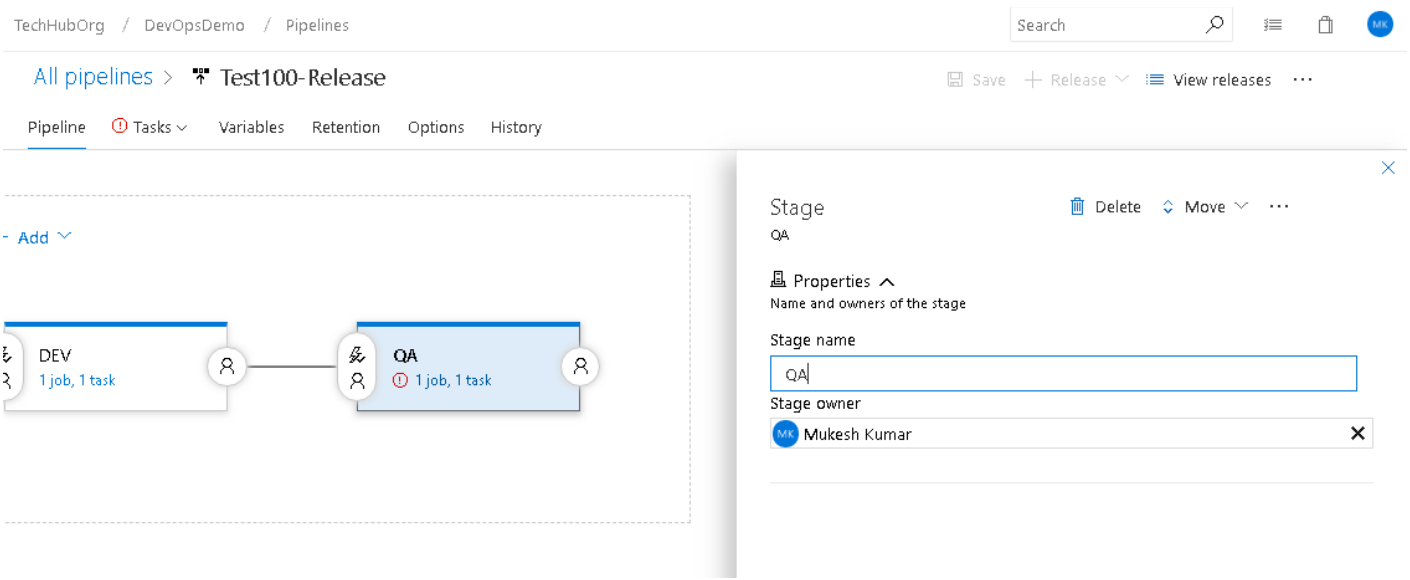
For **adding a new stage**, we can directly click to the **(+ Add)** icon just after Stages or mouse hover just below to the DEV stage, here we will get the icon (+ Add). Just click on it.



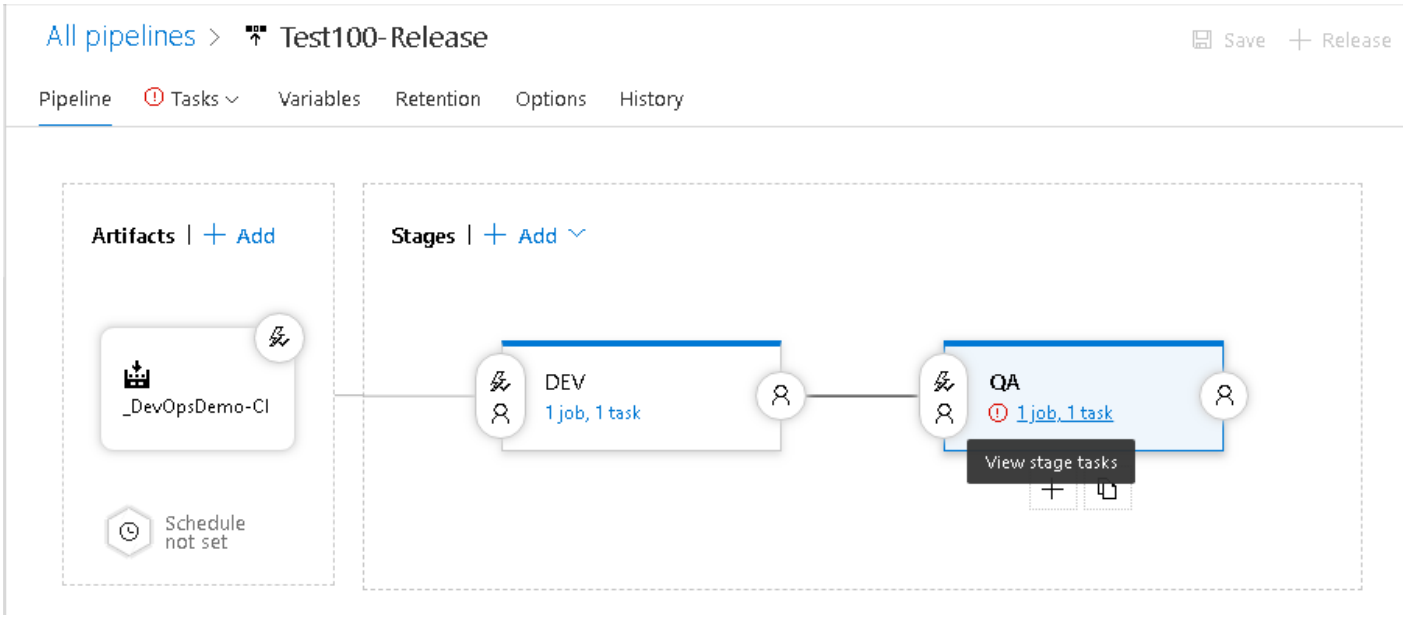
So, it has added one more stage in **Azure DevOps Release Pipeline.** Now, it is asking us to select the template. We have to follow the same process which we have already followed for **DEV (TestDEV-100)** environment setup. So, select the template as **'Azure App Service Deployment' and click to Apply button.**



Close the **'Select Template'** dialog window and we will see that one more stage (**Stage 1 – in above image)** has been added just after DEV stage. Now, click on **Stage 1** and change the stage name to **QA** as shown in the following image.

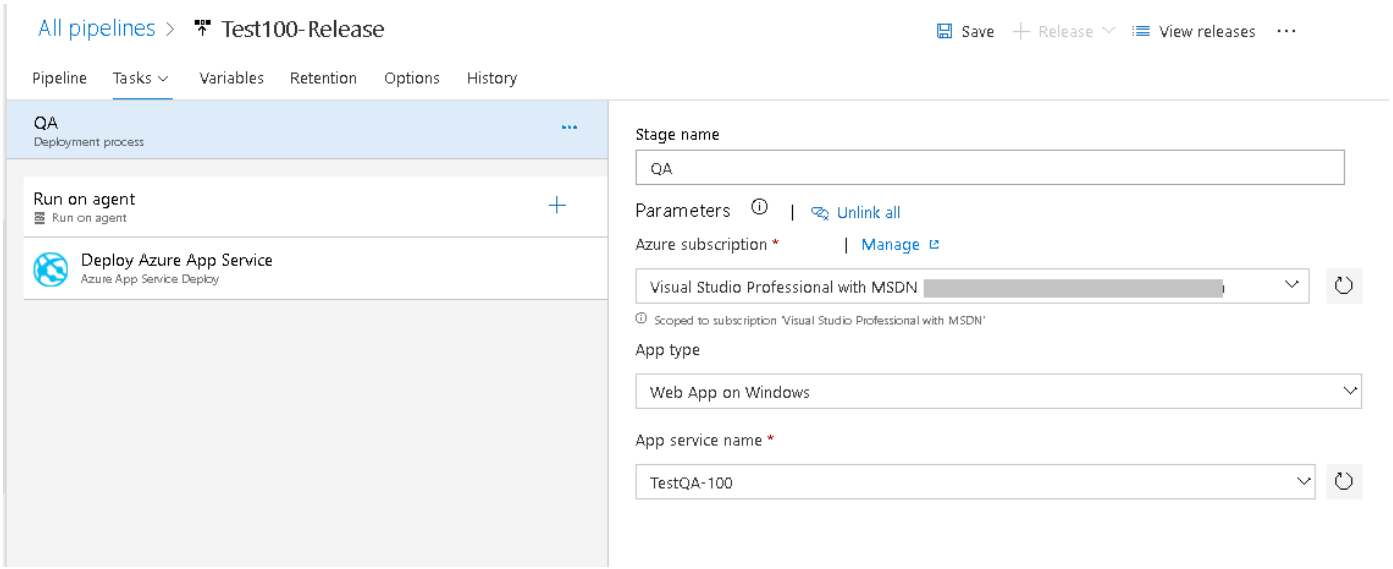


Now close the Stage name dialog using the close **(X)** icon in the right top corner. It's time to configure the **QA (TestQA-100)** environment. So, click on the **'1 Job, 1 Task' in QA stage.**



Here, we have to configure the **QA stage**. The first tab is Tasks tab. Here we will do the configuration as we have already done for DEV stage. So, provide the Azure Subscription and most important the App Service Name as **'TestQA-100'**.

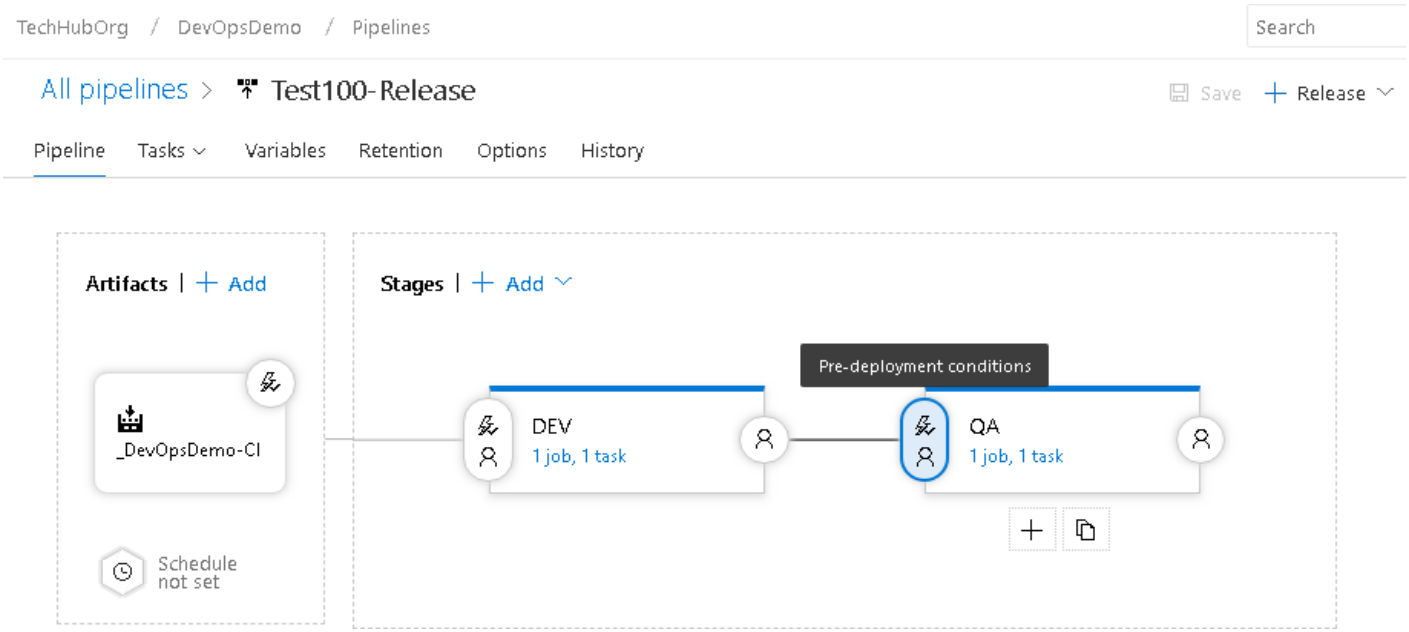
For the rest of the tabs like **Variables**, **Retention**, **Options** and **History**, we are not going to do anything. We will keep the default setting for these tabs. But we can modify it as per our requirements. Now click to **Save**.



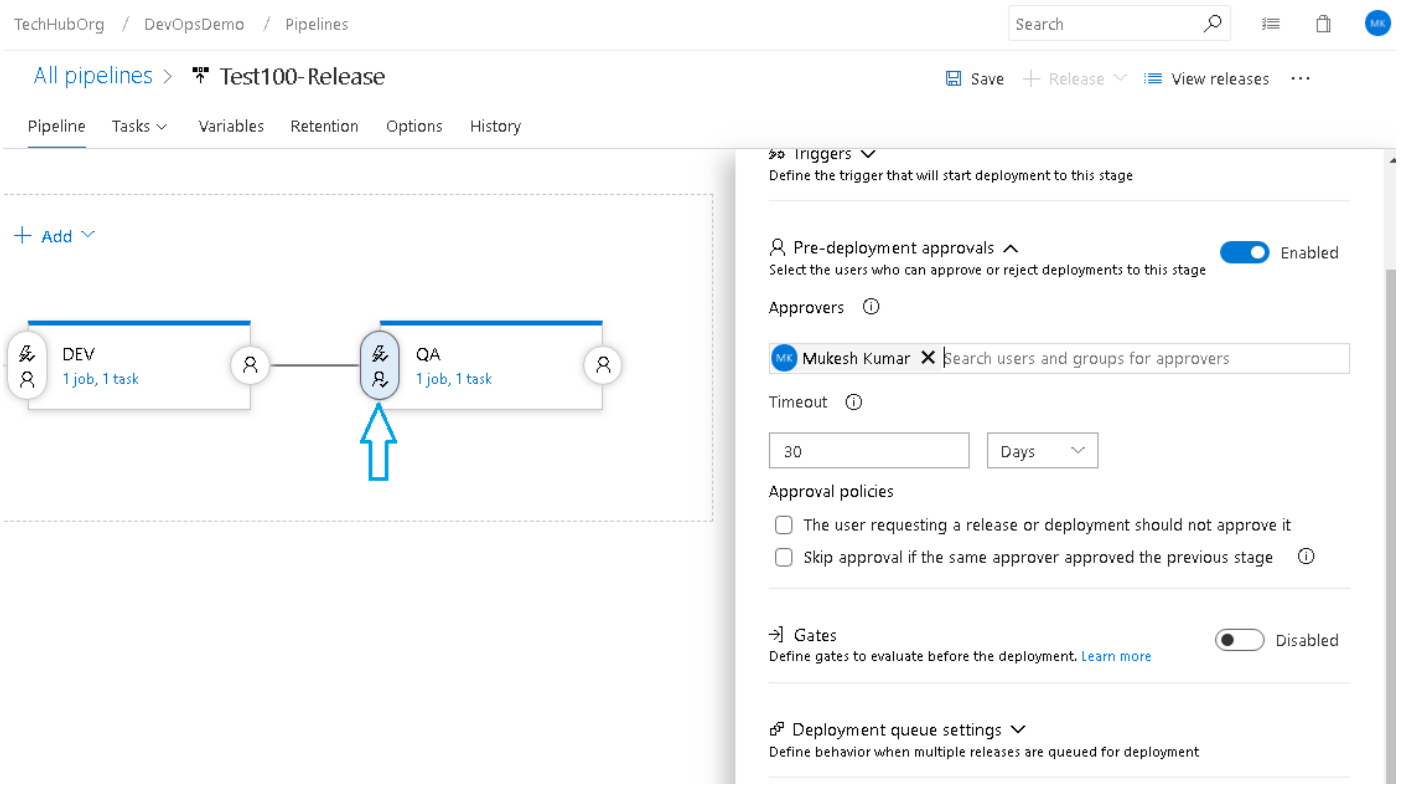
This time, we have two stages, **DEV and QA**. What we have configured so far will deploy automatically on the DEV stage if a new build becomes available. But what about QA? How we will deploy on QA?

Actually for build deployment, it will deploy automatically to all stages one by one once the build is available. But here we would like some confirmation before deploying on the QA stage.

So, configure the **'Pre-Deployment Conditions'**. Click on the **'Pre-Deployment Conditions'** from the **QA stage** as shown in the below image.



Enable the Pre-Deployment Approvals and provide the **name of the approver** in the Approvers section. For this demonstration, we are providing the current user name, **'Mukesh Kumar'**. So, now after DEV deployment, the Artifact will not auto deploy to QA. It will first wait for approval by Mukesh Kumar.

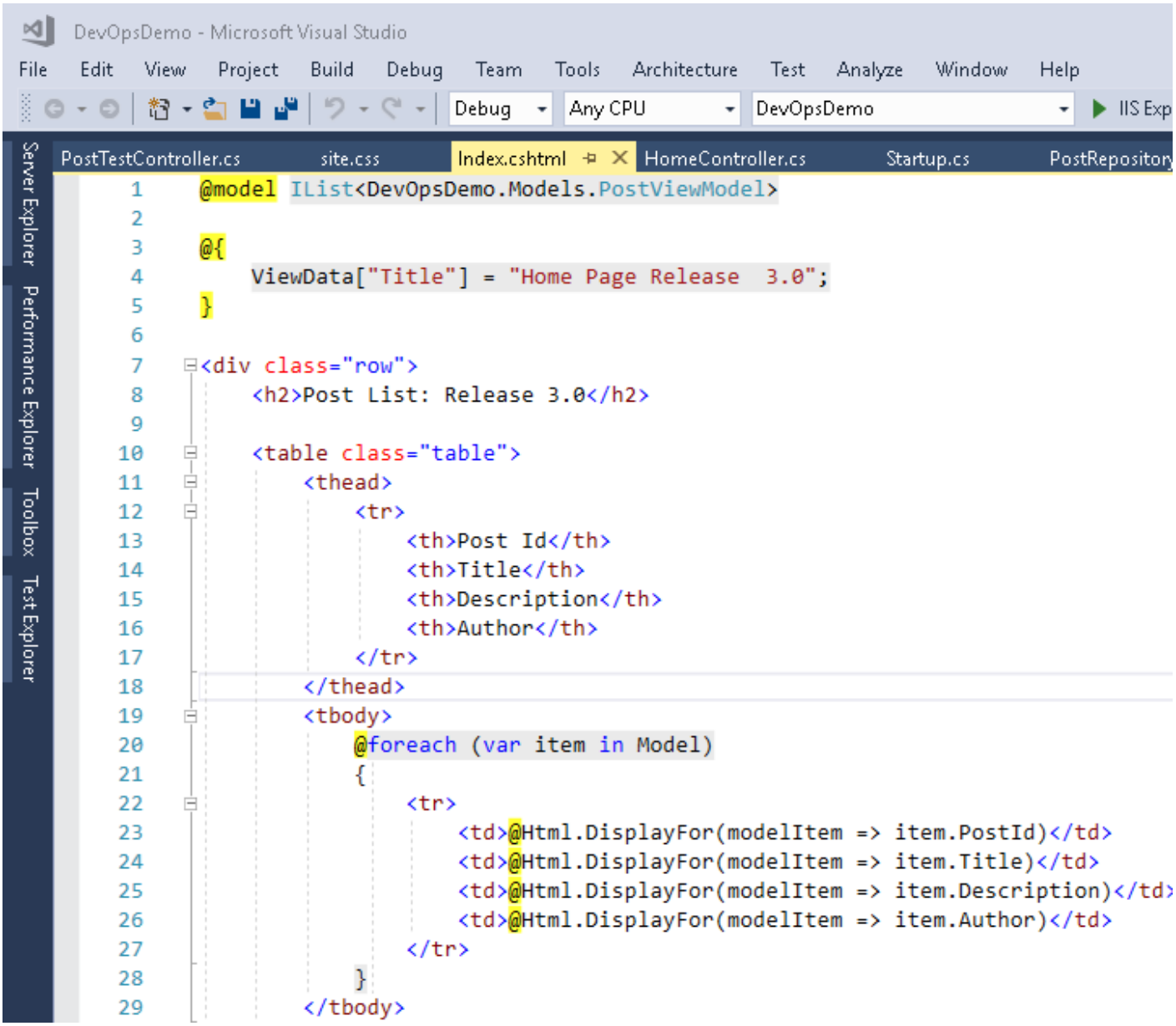


We have finished with the **QA environment setup**. Click to **SAVE** for saving the **Azure DevOps Release Pipeline with new stage as QA.**

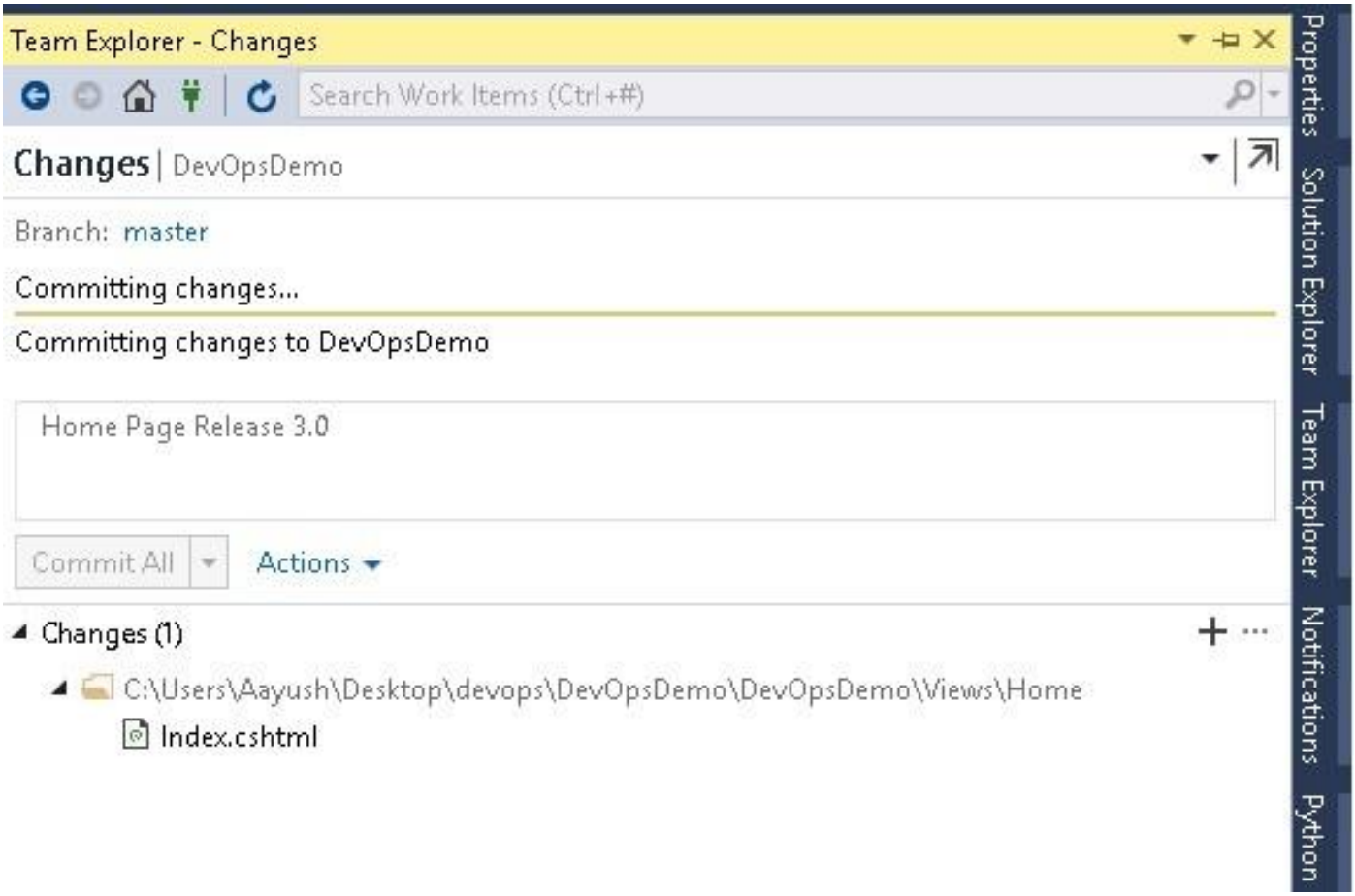
So, what we have achieved so far: We have configured the Azure DevOps Build Pipeline as well as Release pipeline. In Release pipeline, we have created two stages for two different environments like DEV and QA. So, if a new build (Artifact) is available then it first will deploy to DEV (TestDEV-100) environment and will ask for approval before deploying it to QA (TestQA-100).

Now, it's time to see the real time deployment and how the build deploys on different stages. So, open the Visual Studio 2017 or higher version. And open the project '**DevOpsDemo'**, which we have created in the previous section.

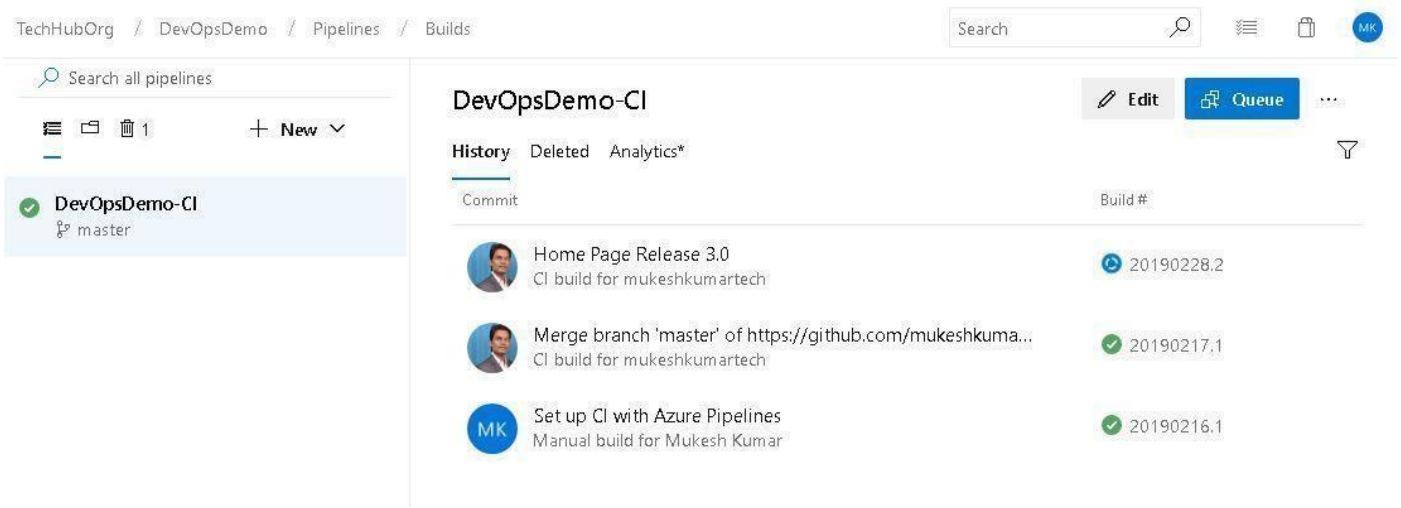
Open the **Index.cshtml** file in main project and replace the text for title from **'Home Page'** to **'Home Page Release 3.0'**. And similarly change the text **'Post List'** with **'Post List Release 3.0'**. Here we will only change the version number and see the output after auto deployment.



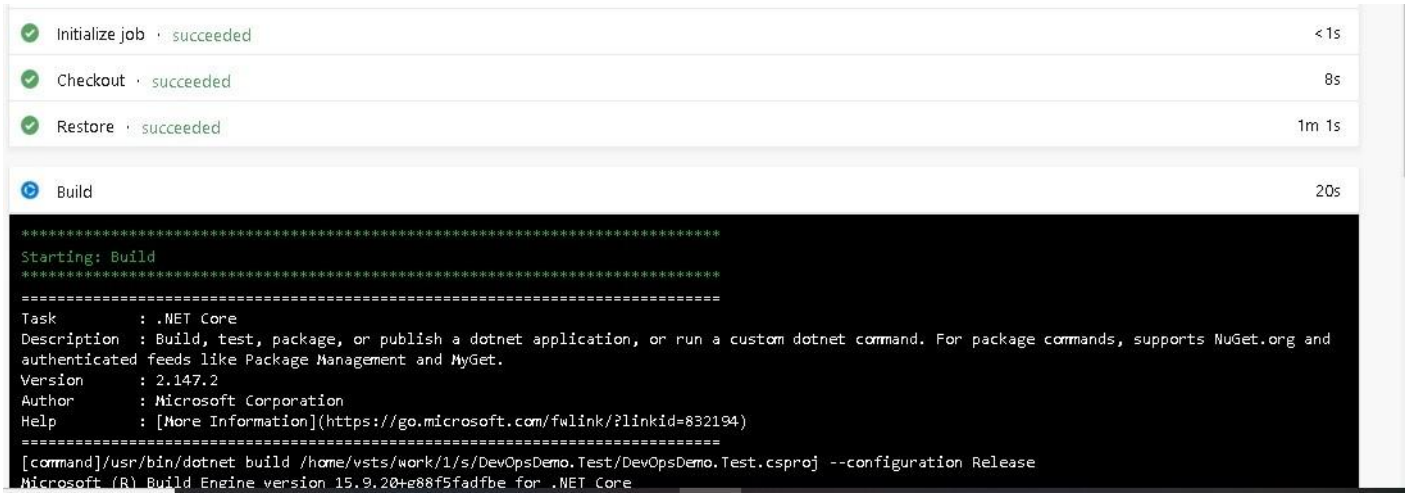
It's time to check the code in the **GitHub** repository. So, go to **Team Explorer > Changes > provide the comment as 'Home Page Release 3.0'** and click to **'commit all and sync'.**



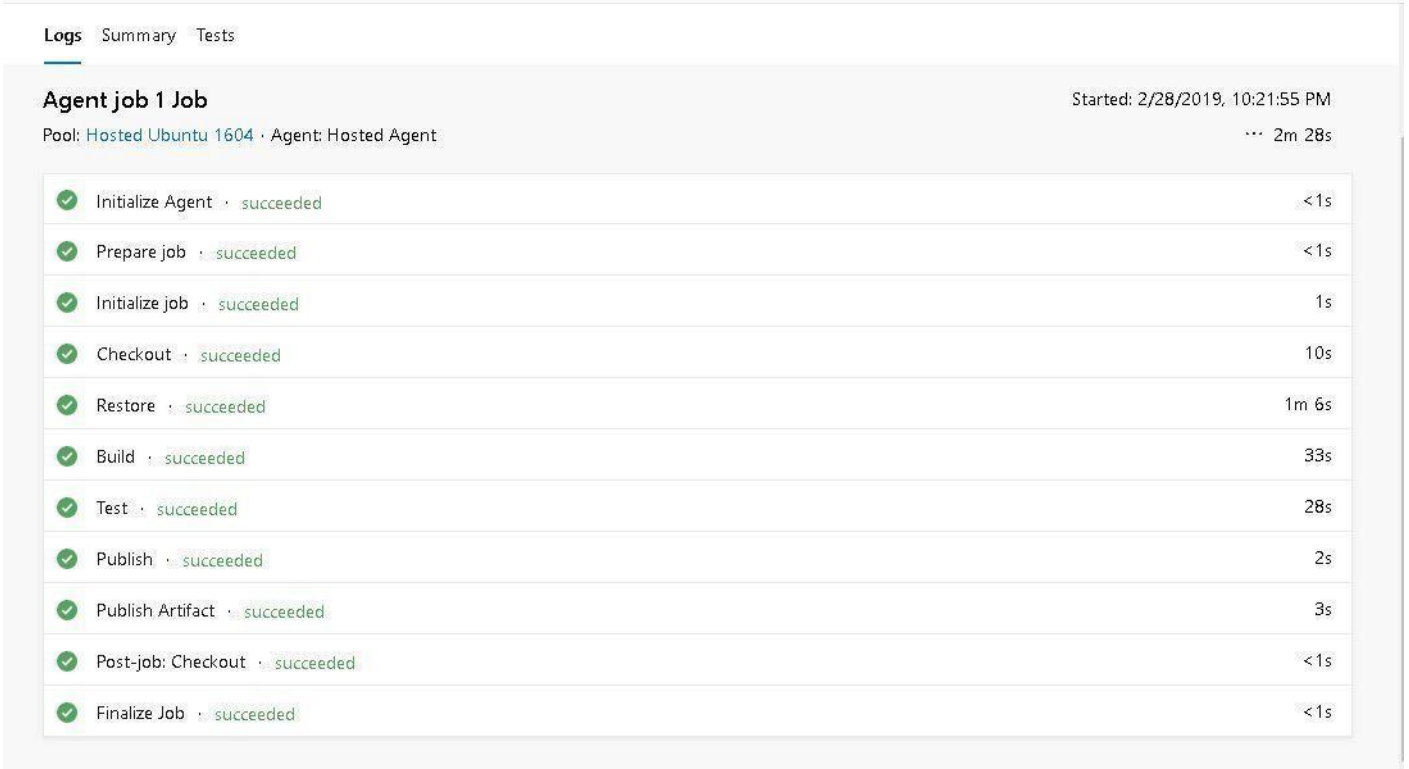
When we open the **Build Pipeline**, in the **history** tab, we can see that a new **Continuous Integration build is started** with the same comment which we have provided at the time of checking the code as **'Home Page Release 3.0'**.



Click on the newly-begun build and we will see that all the Jobs are executing one by one as follows.

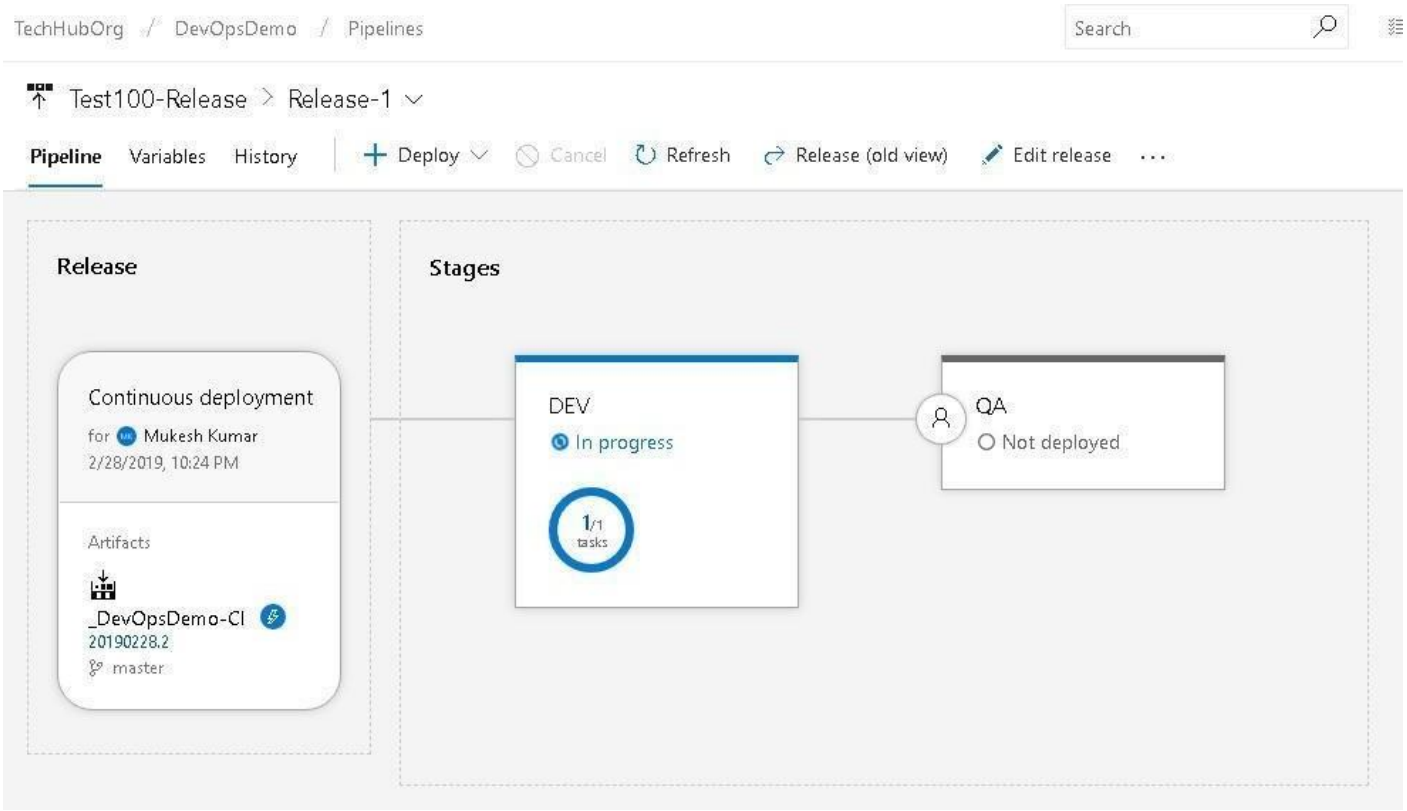


Once all jobs execute successfully, then we can see the **'succeeded'** message in **green for each Job**.

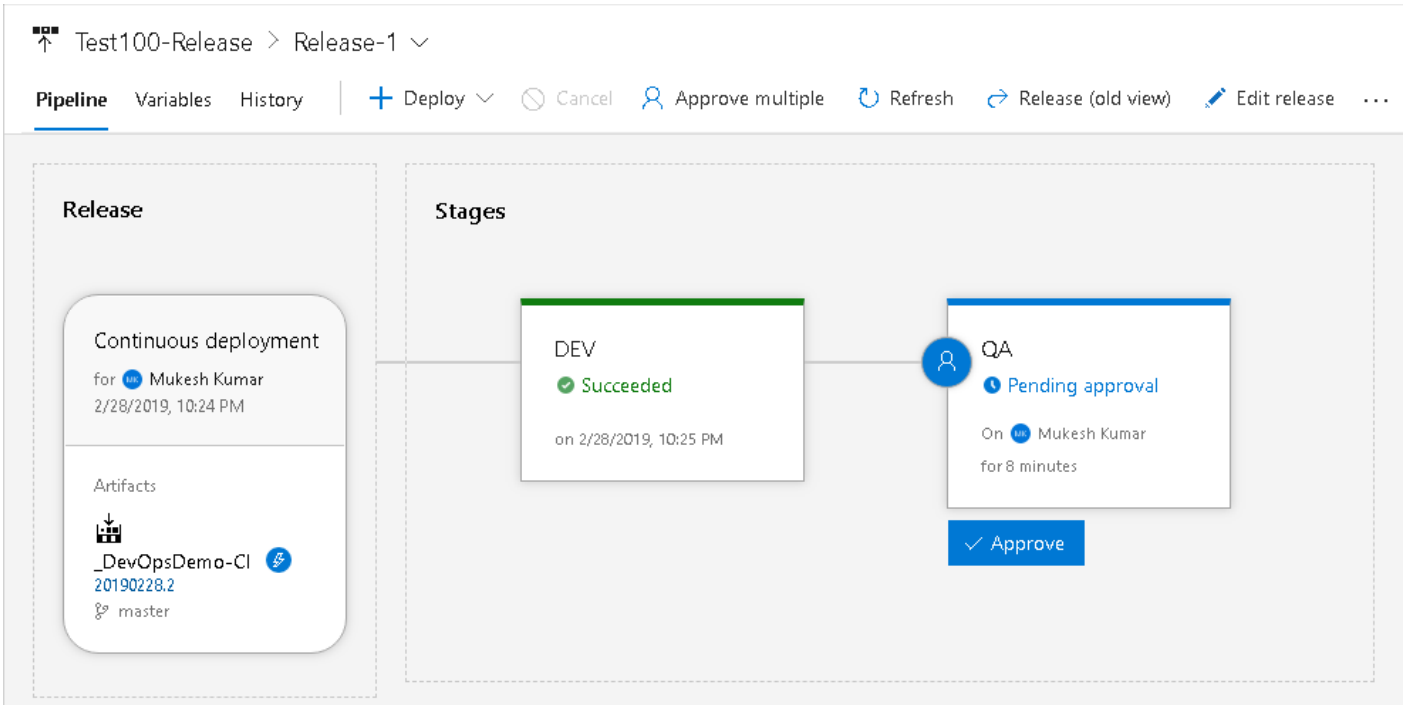


Let's now move **to Azure DevOps Release Pipeline** and open **Test100-Release**. Here we can see that one release is initiated as **'Release-1'** and **DEV** is processing this.

Now, click on the **Release-1** for detailed information. As per the following image, we can see that an artifact **(build)** is available and this is performing the **Continuous Deployment on DEV (TestDEV-100)** environment. It is in progress.

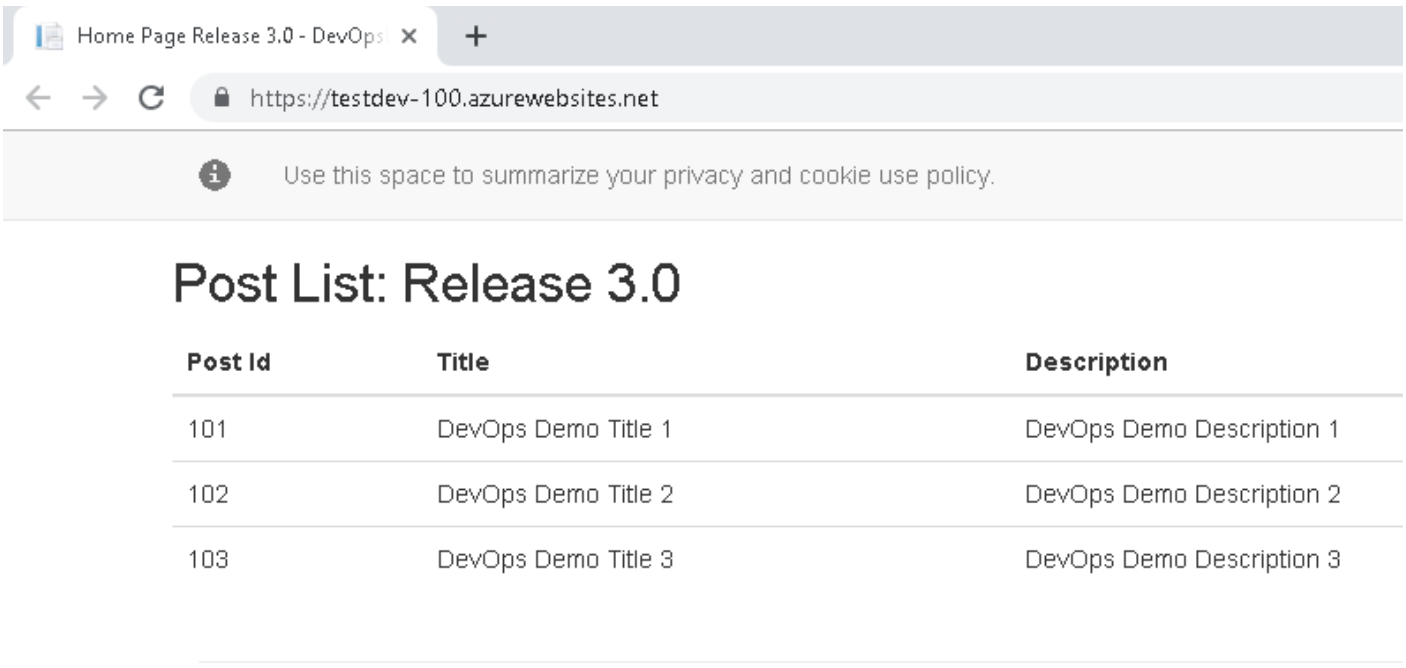


After a few minutes, we will see that **Continuous Deployment** has begun on **DEV**. But for **QA**, it is **pending for approval**. Once a current user (**Mukesh Kumar**), as we have defined earlier, approves it, then Continuous Deployment will start on QA.

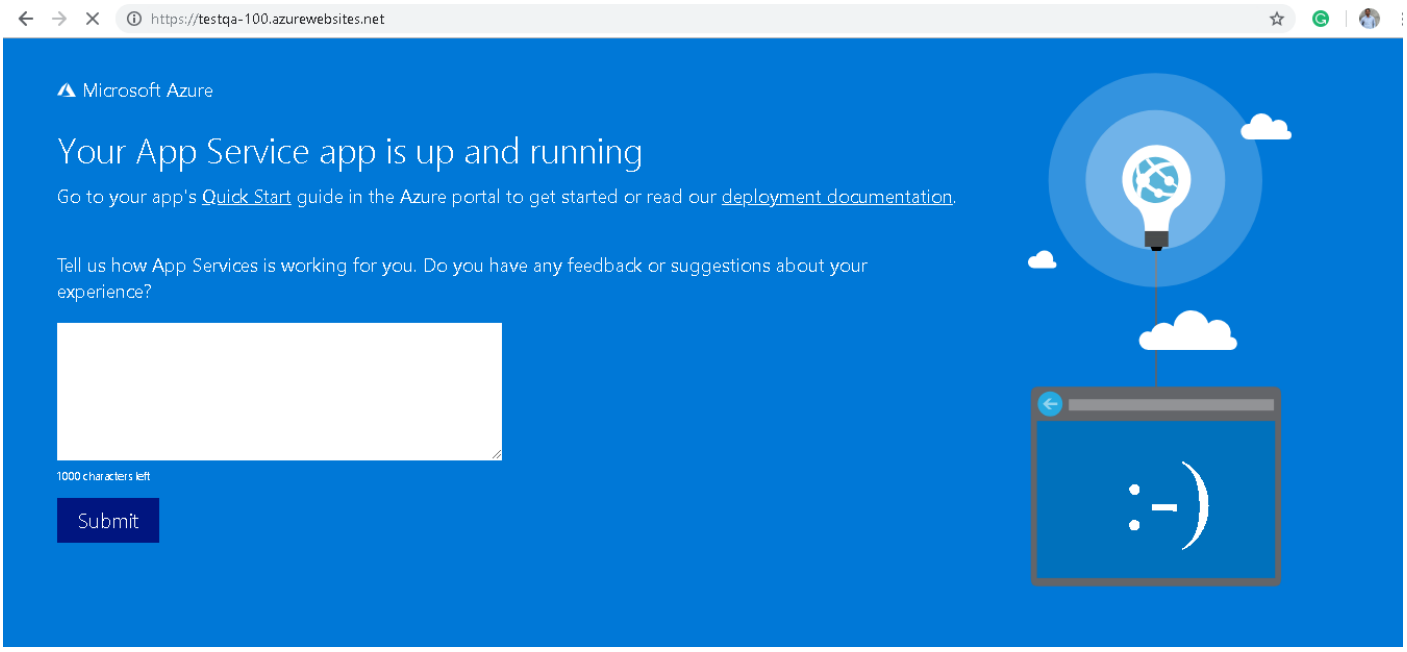


So, let's first check what has been deployed on the **DEV** environment. So, open the **TestDEV-100 App service**. Here we can find the URL for accessing the TestDEV-100 app service and it is **https://testdev-100.azurewebsites.net**. Open this in **browser,** and here we go.

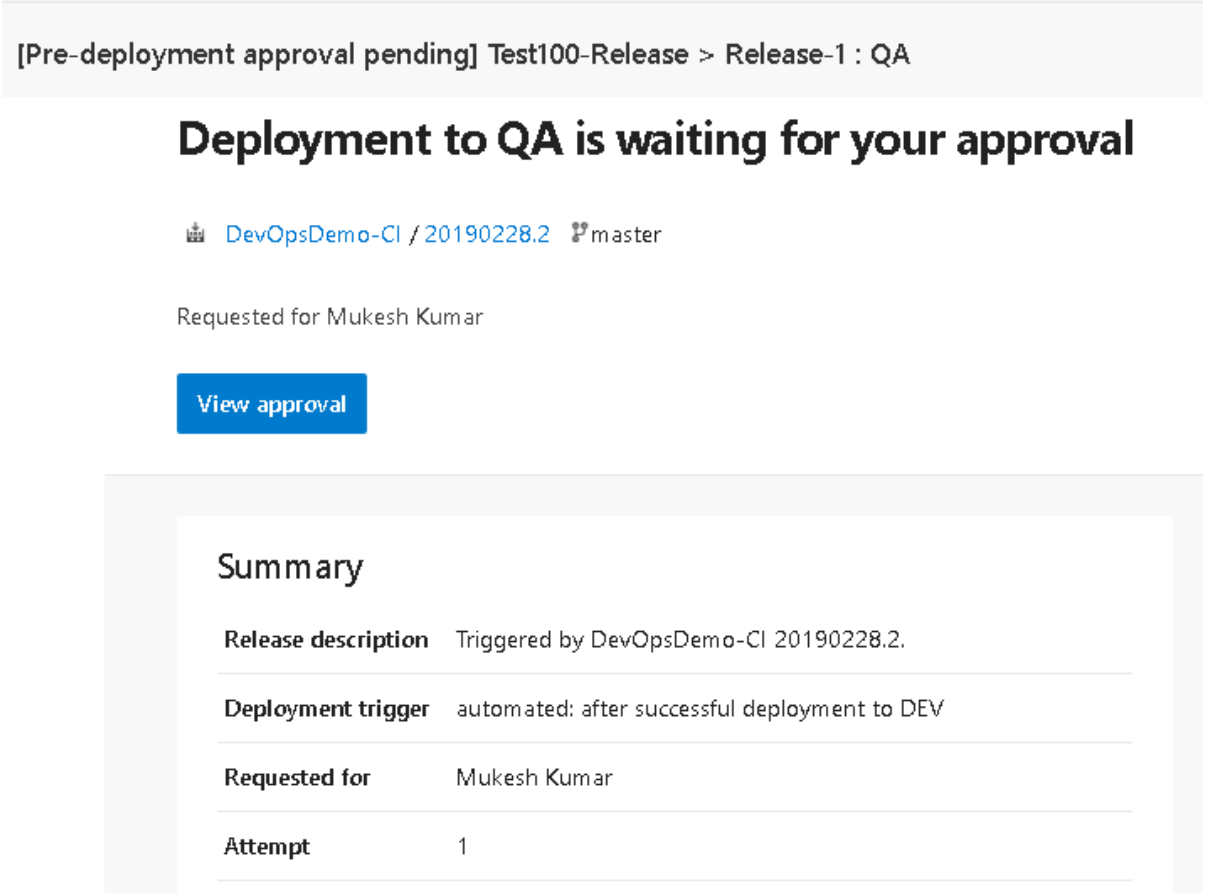
Great, as we can see with the below image, the **Release 3.0 on DEV (TestDEV-100)** environment has been successfully deployed. We can see both the title and post list text are showing **Release 3.0**.



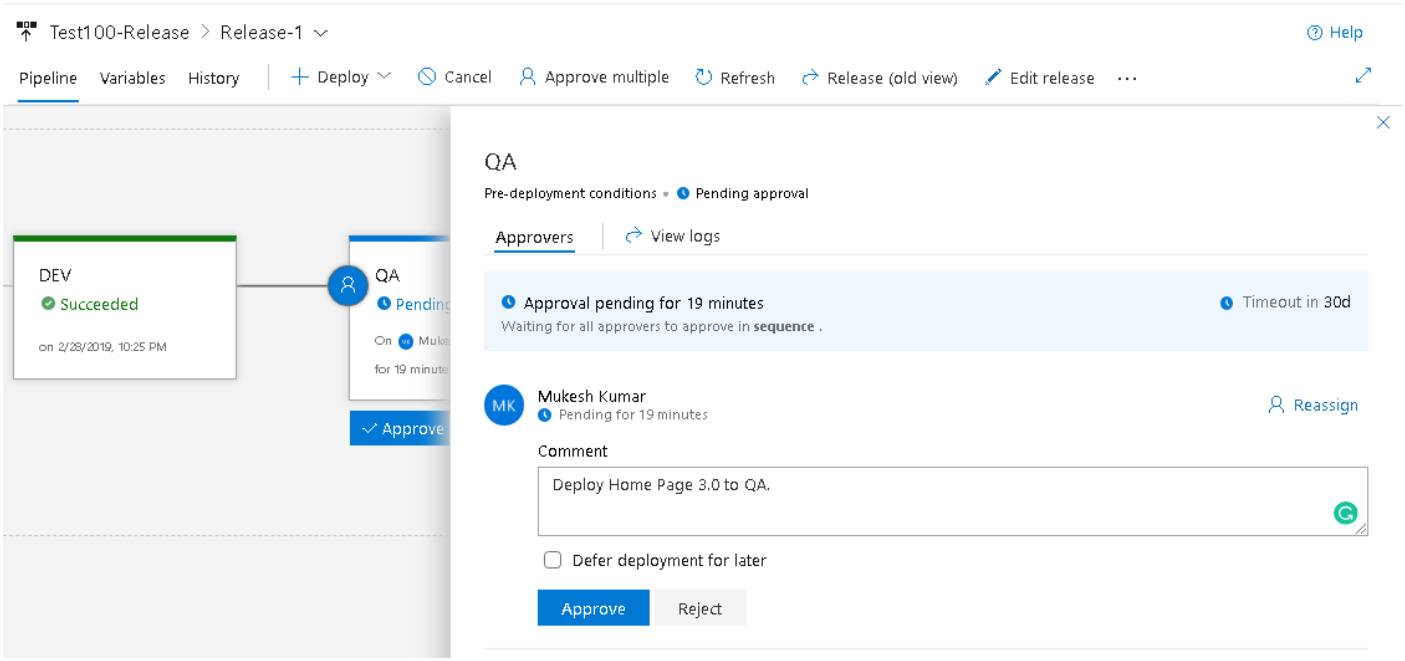
Now, let's move on to the **QA** environment and open **TestQA-100** app service using the URL **https://testqa-100.azurewebsites.net**. And we will see that nothing has deployed yet on QA environment. And it's showing the default page for Azure App Service.



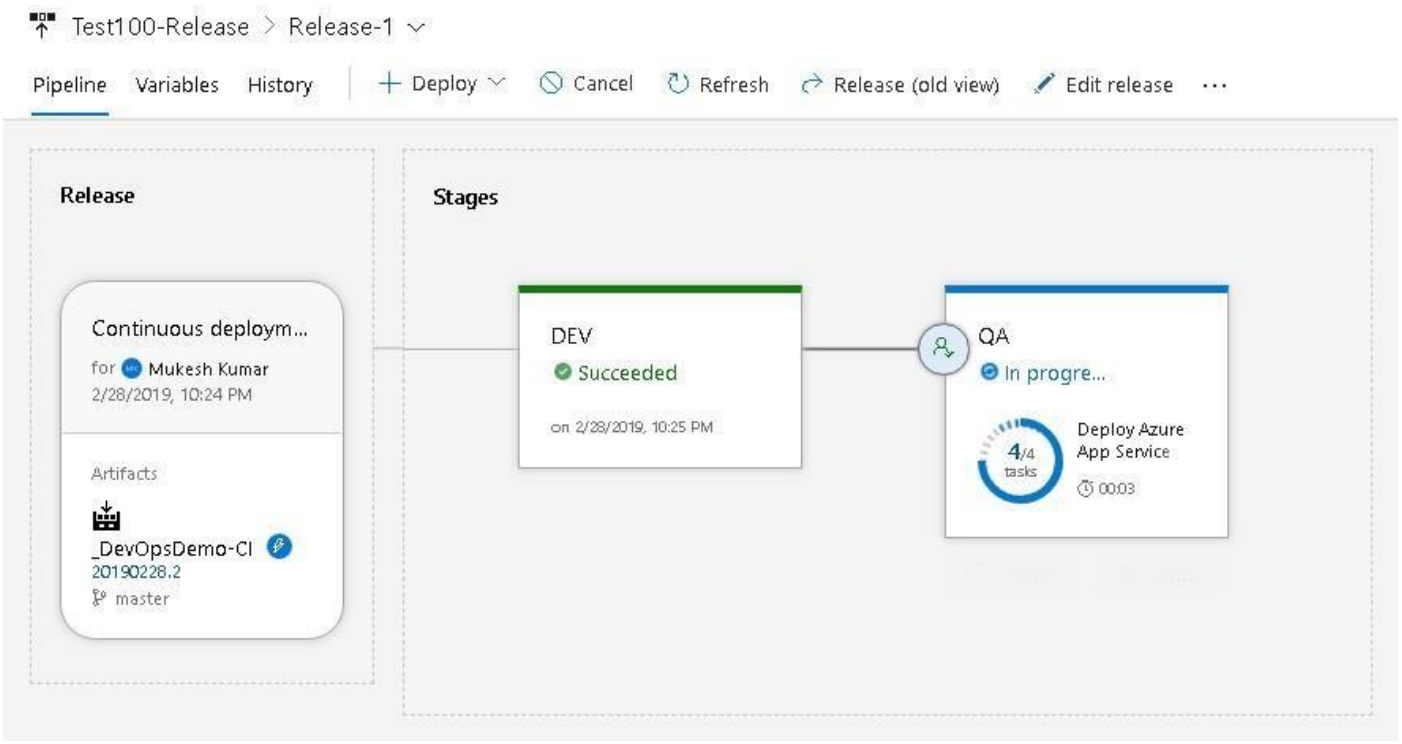
We will also be informed via email that approval is pending for QA deployment.



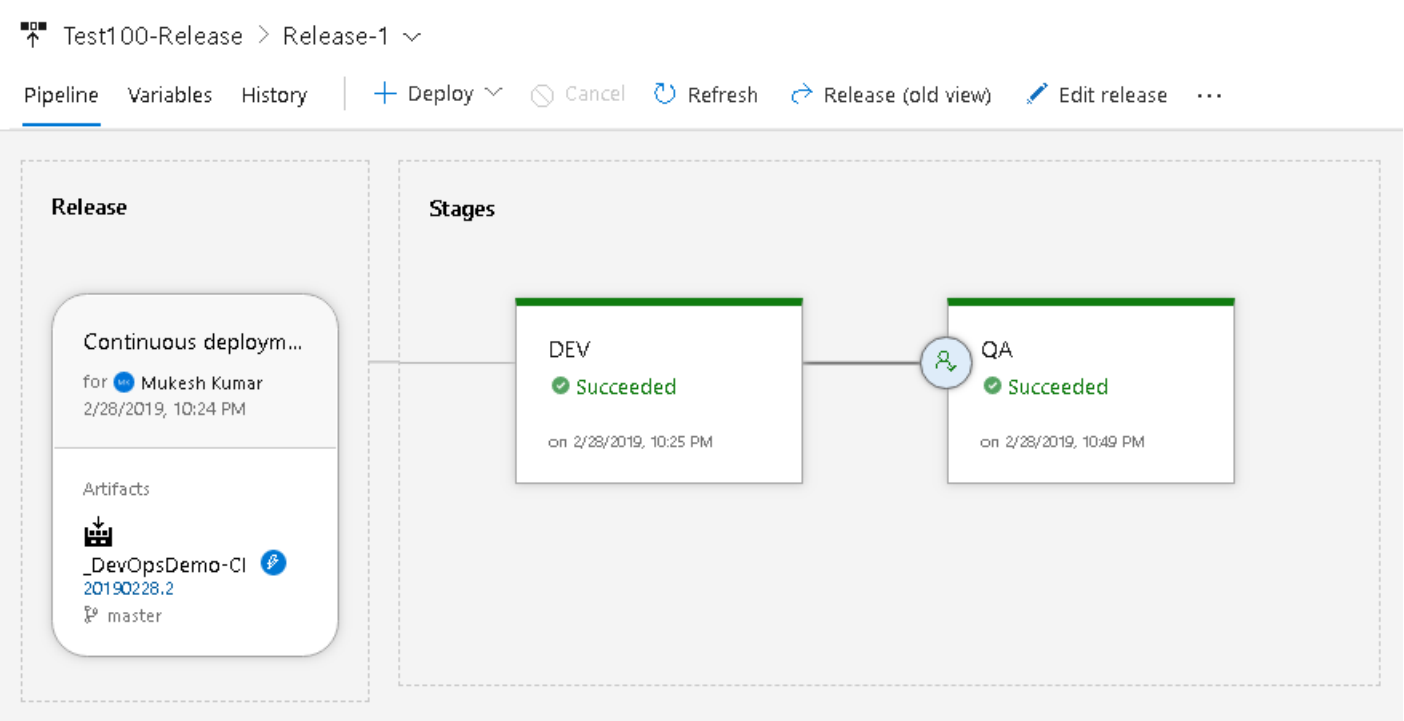
Now, the current user **(Mukesh Kumar)** needs to approve the pending approval for QA environment. When we click on the **Approve button** from the **QA** stage, it will open the dialog where we can put some comment before approving it. So, put the comment as **'Deploy Home Page 3.0 to QA'** and click the **Approve** button.



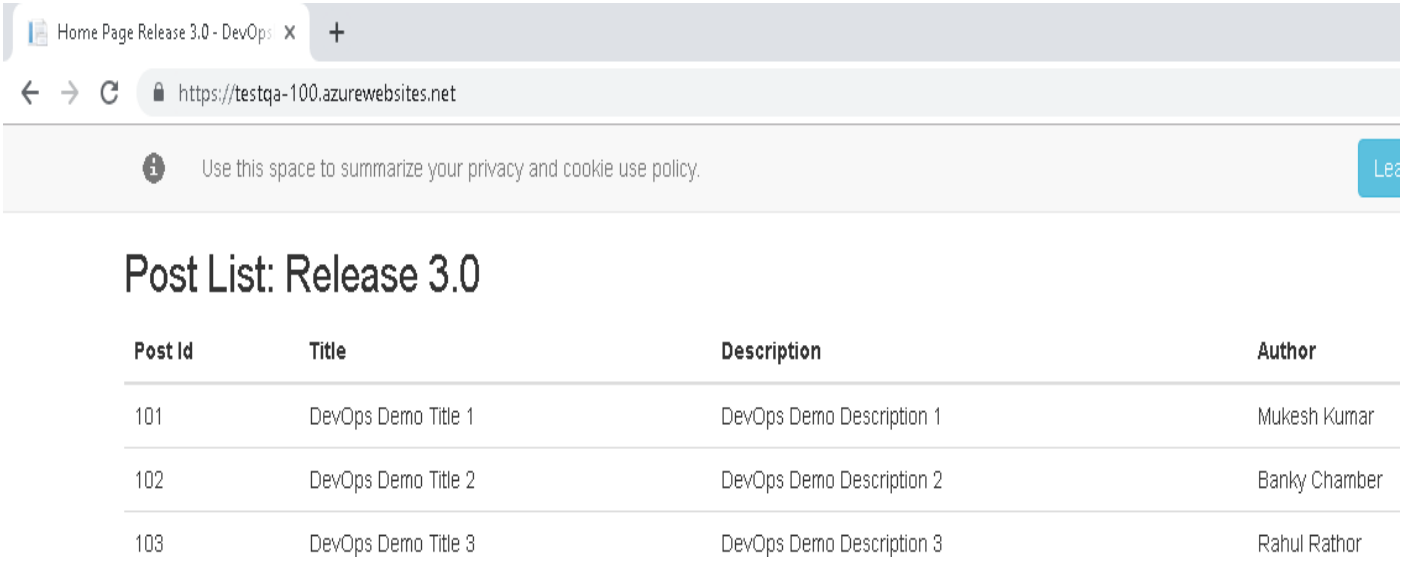
Continous Deployment has initiated and will start deploying the Artifact (Build) on the **QA (TestQA-100)** environment.



After a few minutes, the deployment will be finished on QA and we can see the **succeeded** message on the **QA stage.**



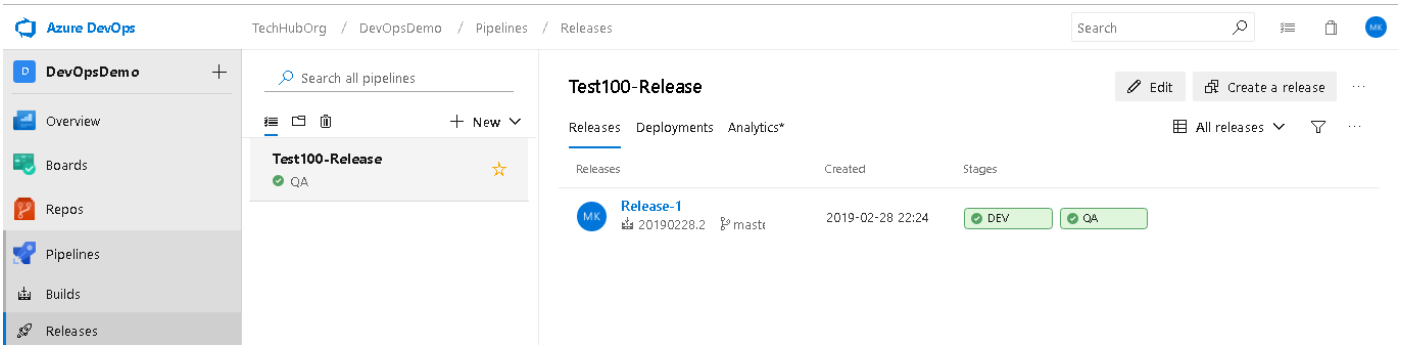
Now, once more let's open the **QA (TestQA-100 App Service)** URL as **https://testqa- 100.azurewebsites.net**. And here we go, we have **successfully deployed the QA of release 3.0.**



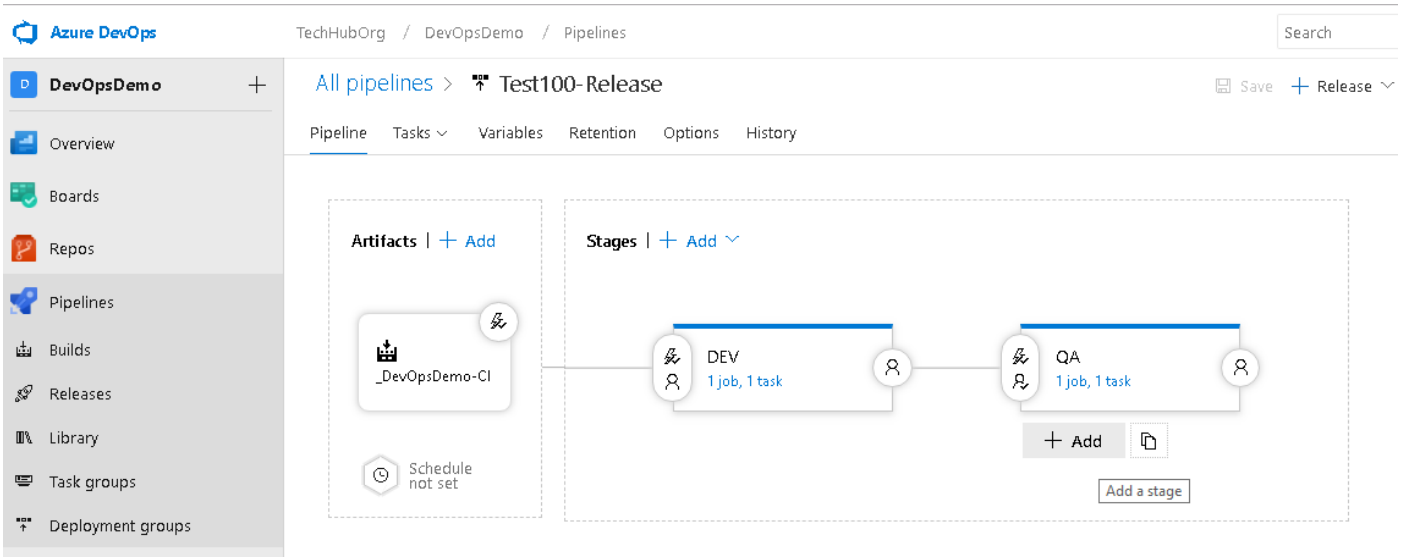
So far, so good. We have seen how to commit the code into the GitHub repository, build the code in Build pipeline, execute the unit test cases in Build pipeline, and deploy on the DEV and QA environment.

8.3 Create Prod Stage

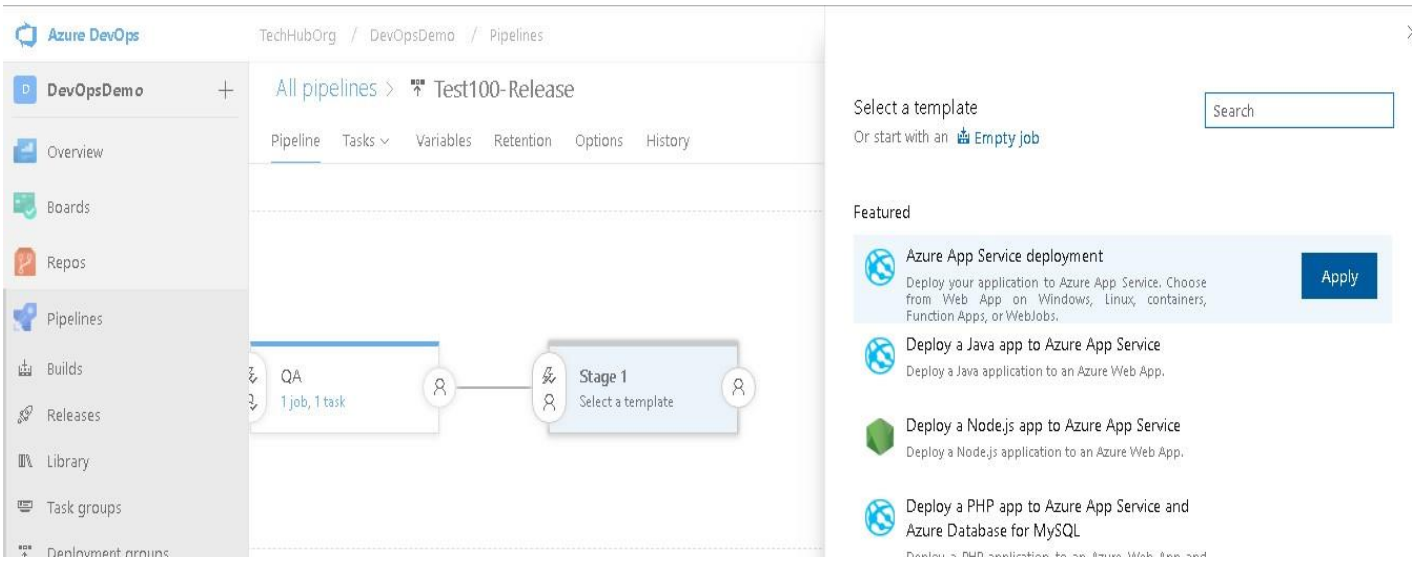
Now, it's time to add one more stage as **PROD**. So, open the **Pipeline > Release > Test100-Release**. From here just click on **Edit** button for editing the **Azure DevOps Release Pipeline**.



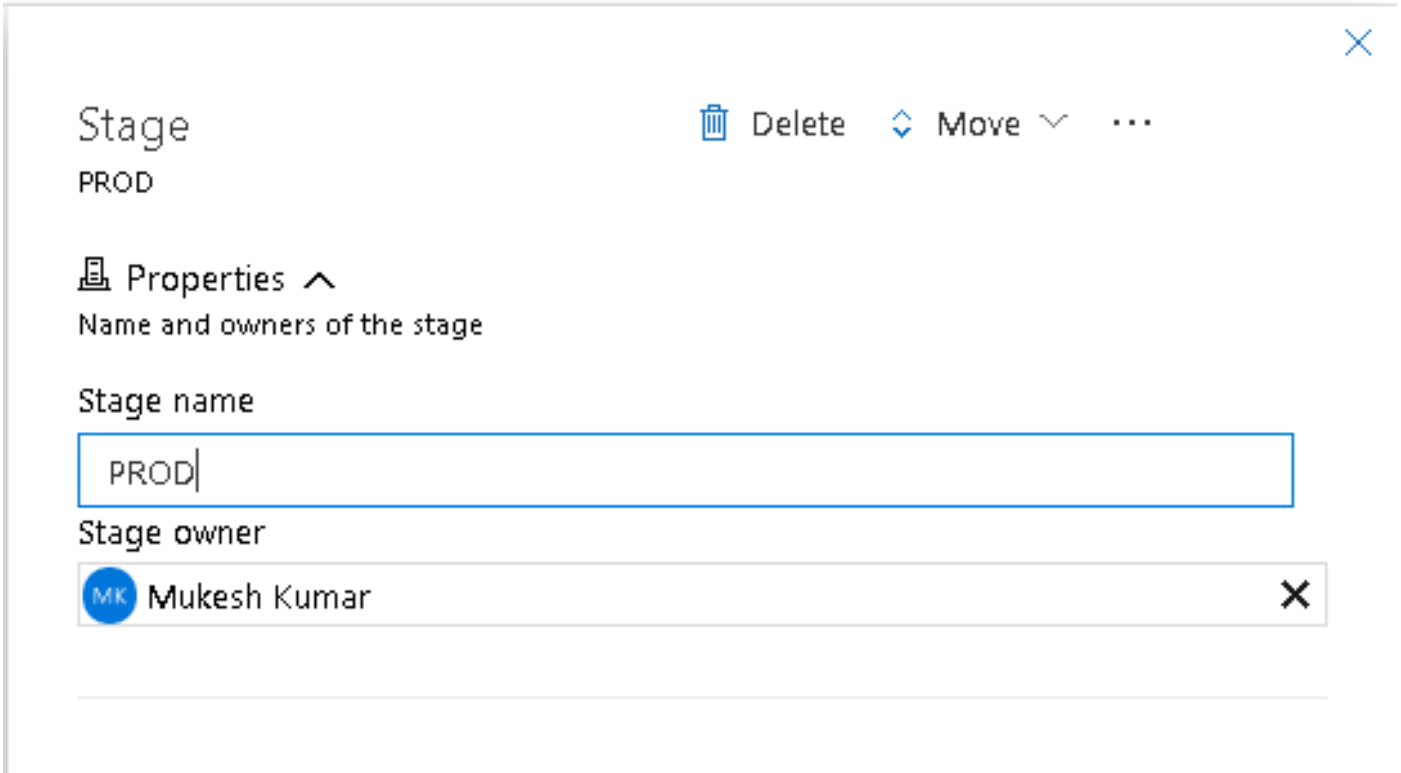
Follow the same steps which we have already followed while adding the **QA stage**. So, just mouse hover as below over QA stage and click on **(+ Add)**icon for adding the new stage.



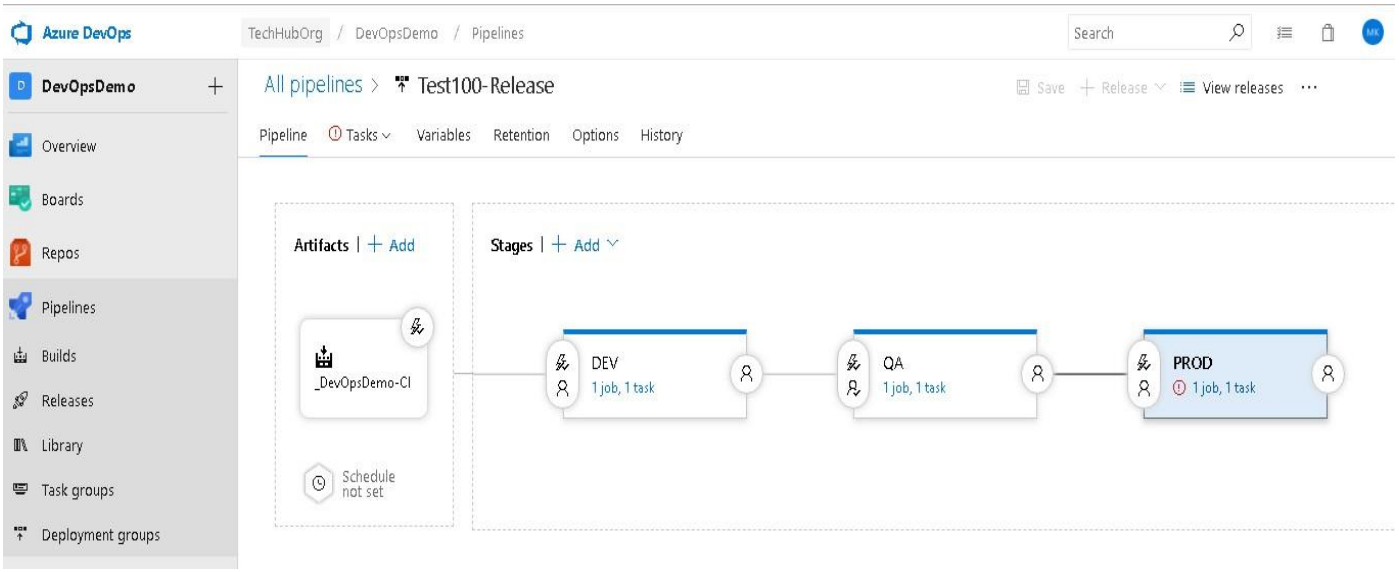
As always, it will ask you to select the template. So, select the template as **'Azure App Service Deployment'** and click on **Apply**. After applying the template just close this dialog window.



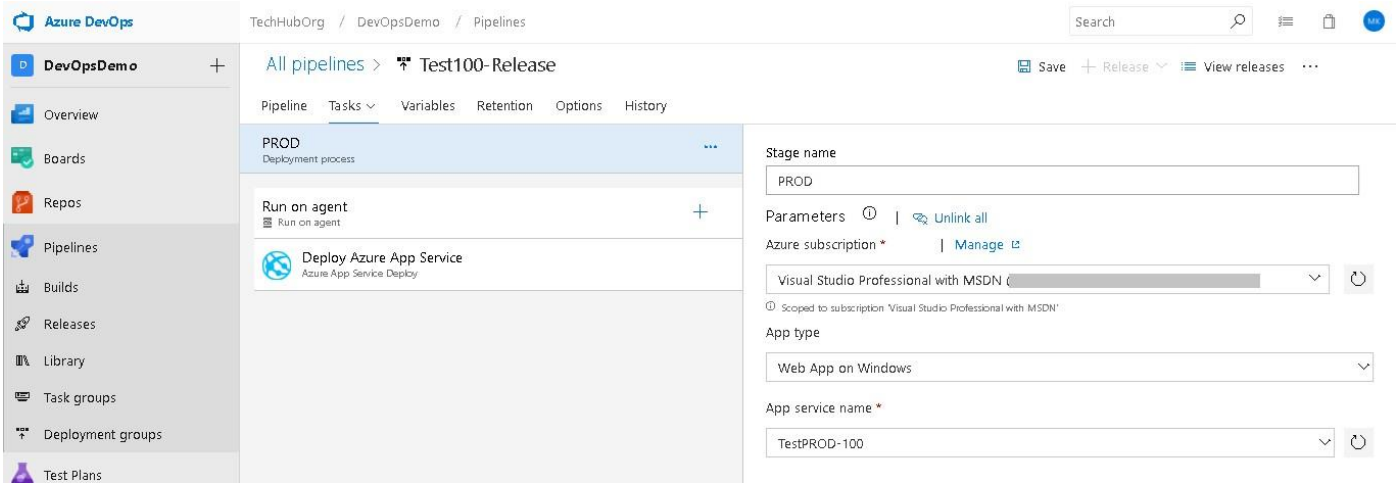
Now, we can see that just after the **QA** stage, one more stage has been added as **'Stage 1'**. Just click on it and it will open the stage name dialog popup. From here we can change the name of the stage. So, just add the name as '**PROD'**. And close the Stage dialog popup using the close icon **(X)** at the right top corner.



Here with the following image, we can clearly see that we now have 3 different environments for **DEV, QA and PROD** respectively. But configuration is pending for the **PROD (TestPROD-100)** environment. So, let's complete it first. Click on the **(1 Job, 1 Task)**.

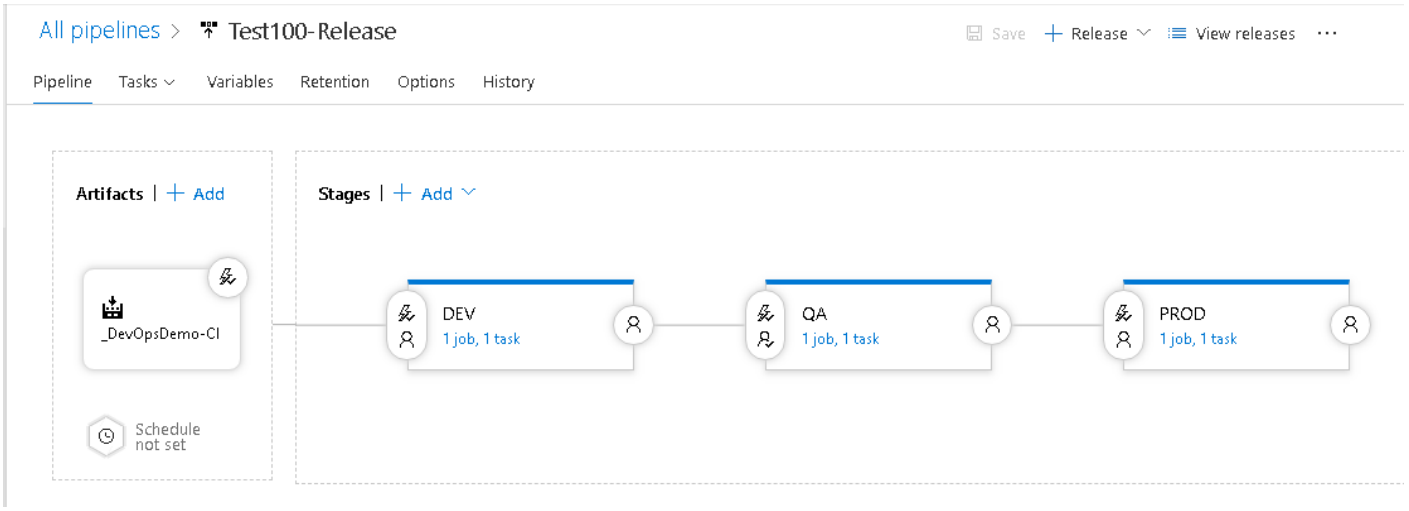


Go to the **Tasks** tab, Provide the **Azure Subscription** and app type as we have done previously. But be careful while selecting the App Service Name. This time, we will choose the **TestPRDO-100** as an App Service name.

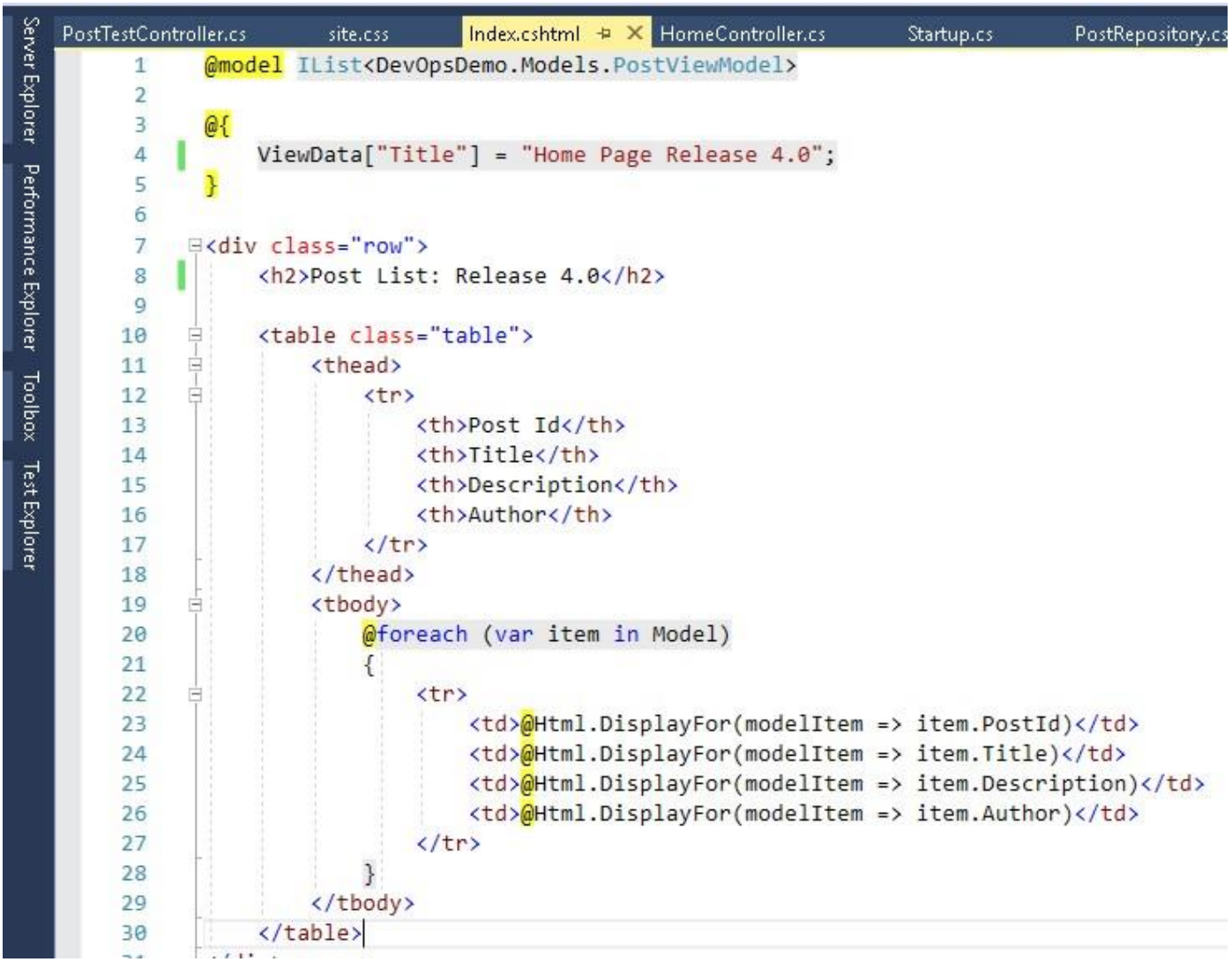


We don't have to do anything for **variable, retention, options and history**. Keep all tabs with default values and click on **SAVE**.

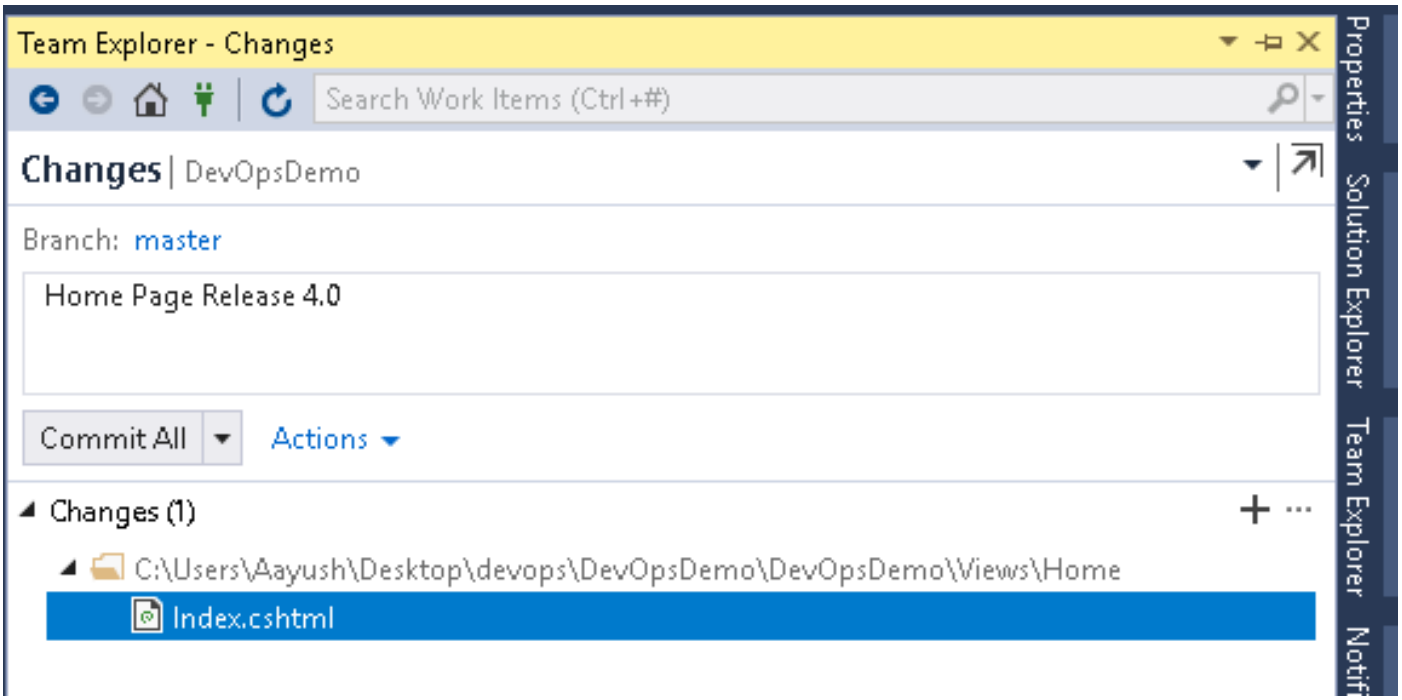
So, we have all 3 environments ready as follows. We are configuring any approval for **PROD** deployment. After QA deployment, PROD will start deploying.



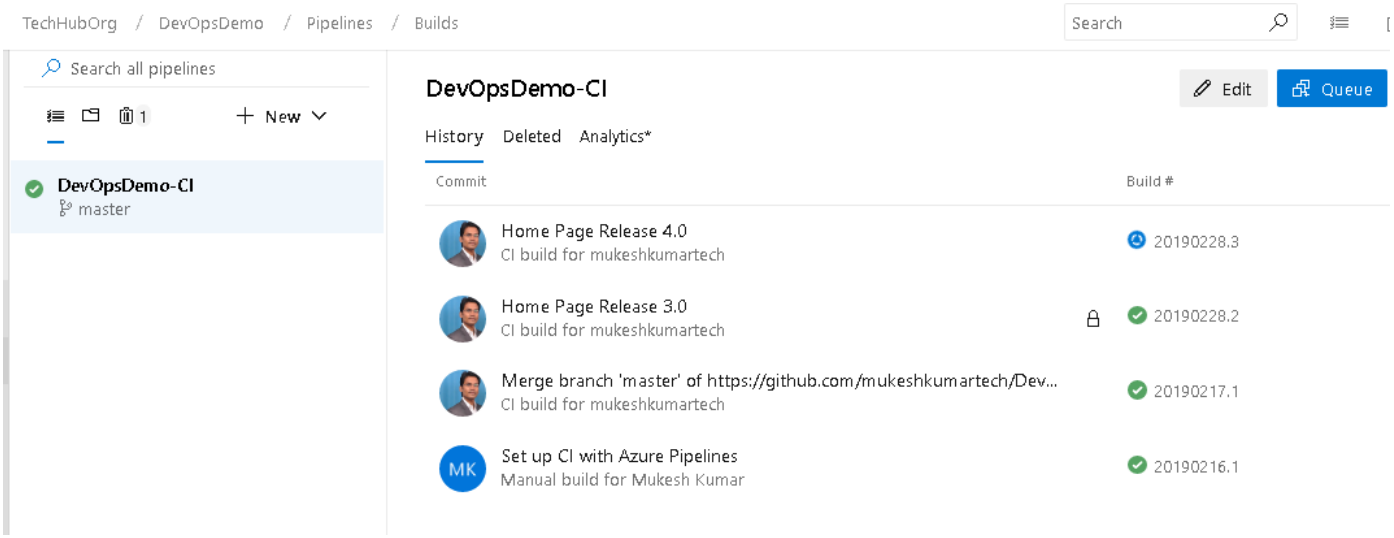
Let's move to **Visual Studio 2017 or higher version** and make again changes in **Index.cshtml** file. This time we will only change the version number from **'3.0' to '4.0'**. The rest of the code will be the same.



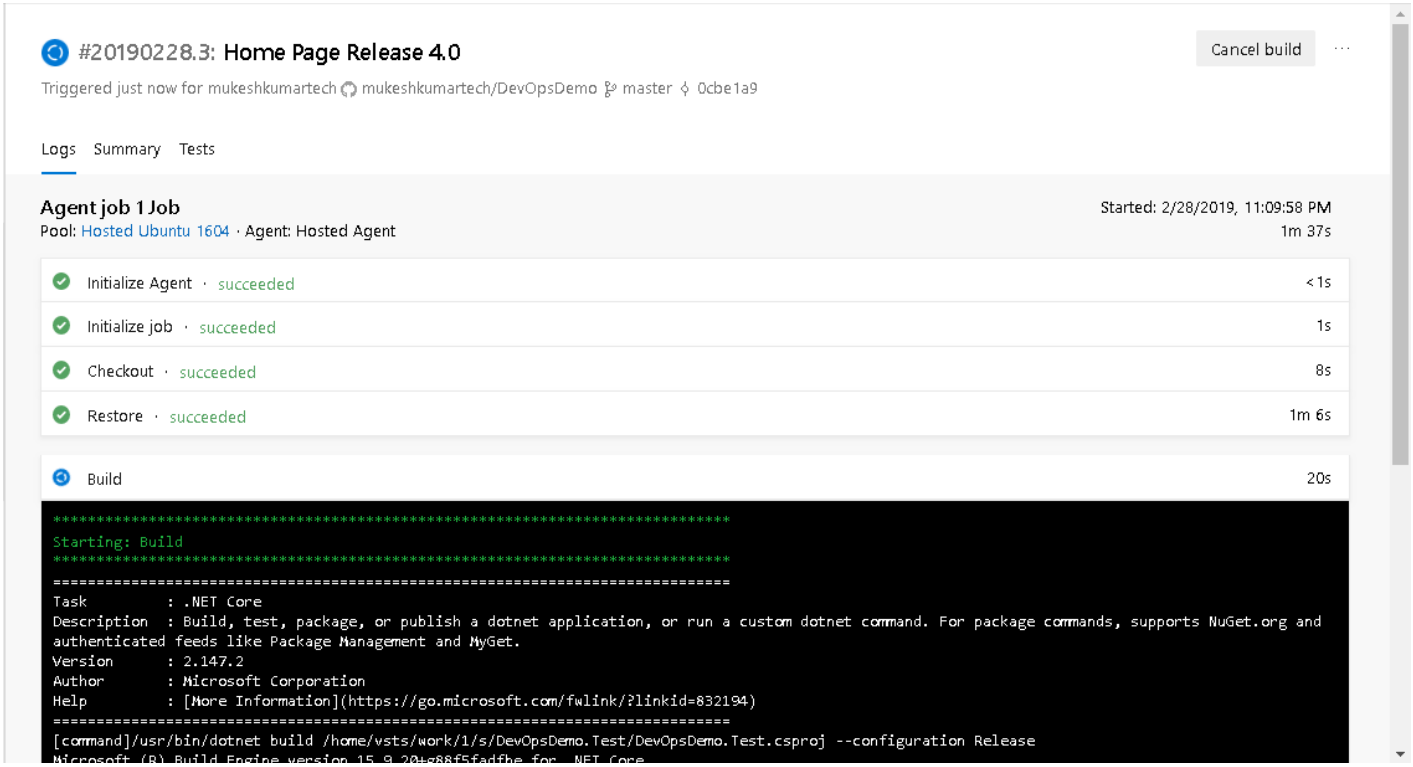
After making the changes in **Index.cshtm** as above, check in the code. First provide a valuable comment so that it can be tracked and **Commit All and Sync**.



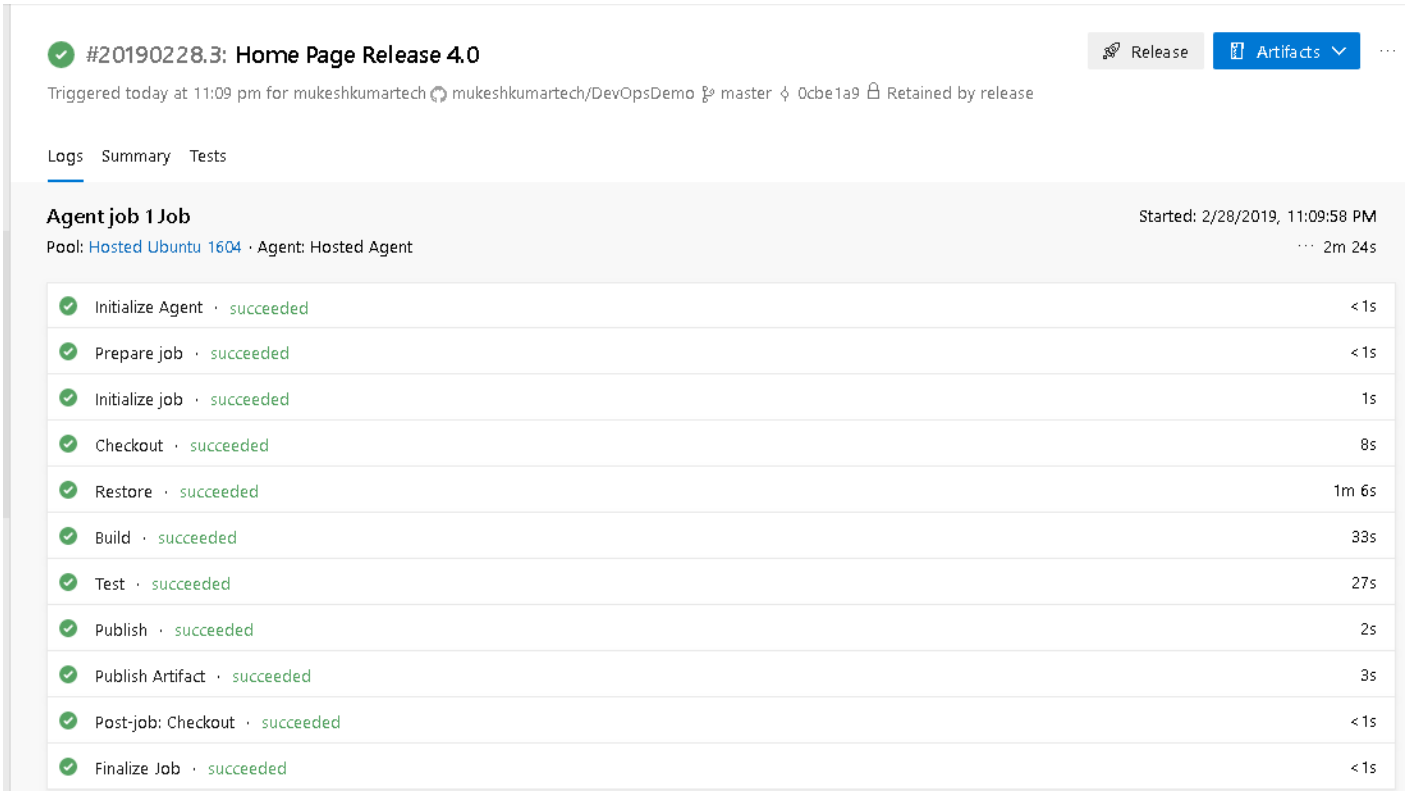
After successfully checking in the code, move to build pipeline **(TechHubOrg > DevOpsDemo > Pipeline > Builds)**. Here we can see that a new build has initiated as **'Home Page Release 4.0'**.



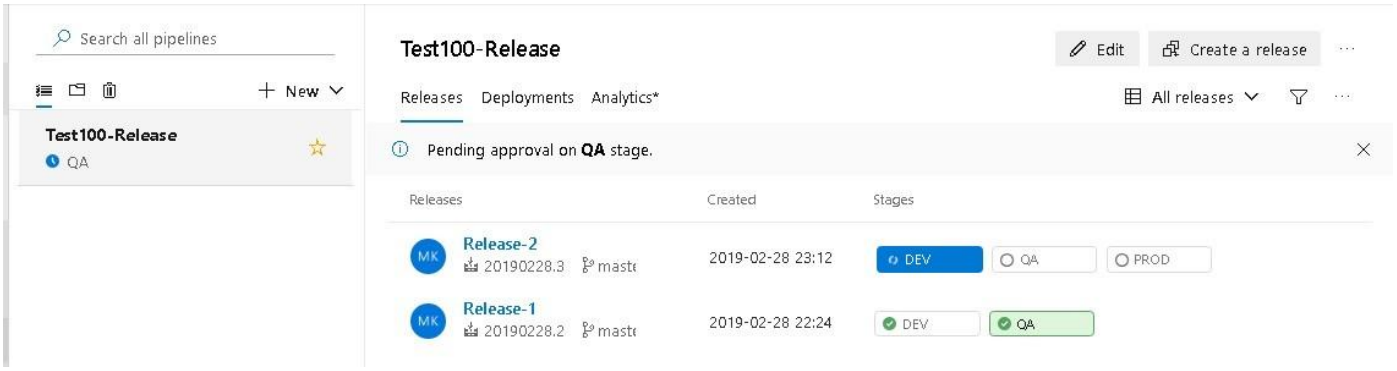
Open the new build **'Home Page Release 4.0'** as follows. Here it will perform all jobs before completing the Build.



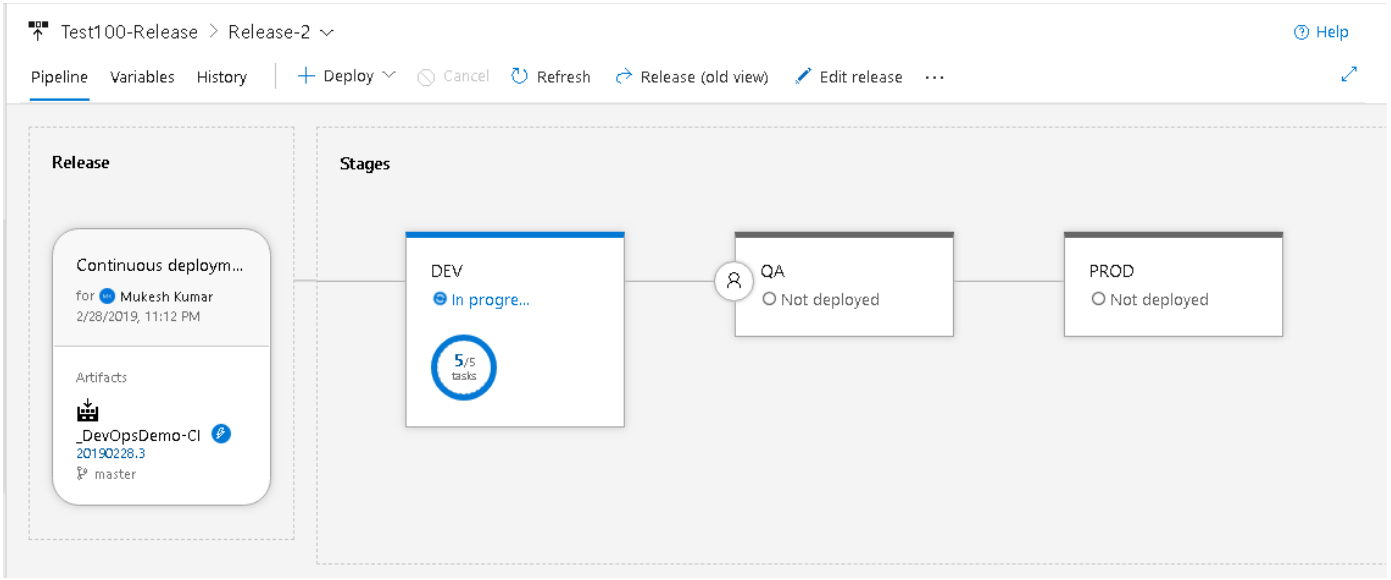
Wait for completing the build and within a few minutes we will see that it has completed as follows.



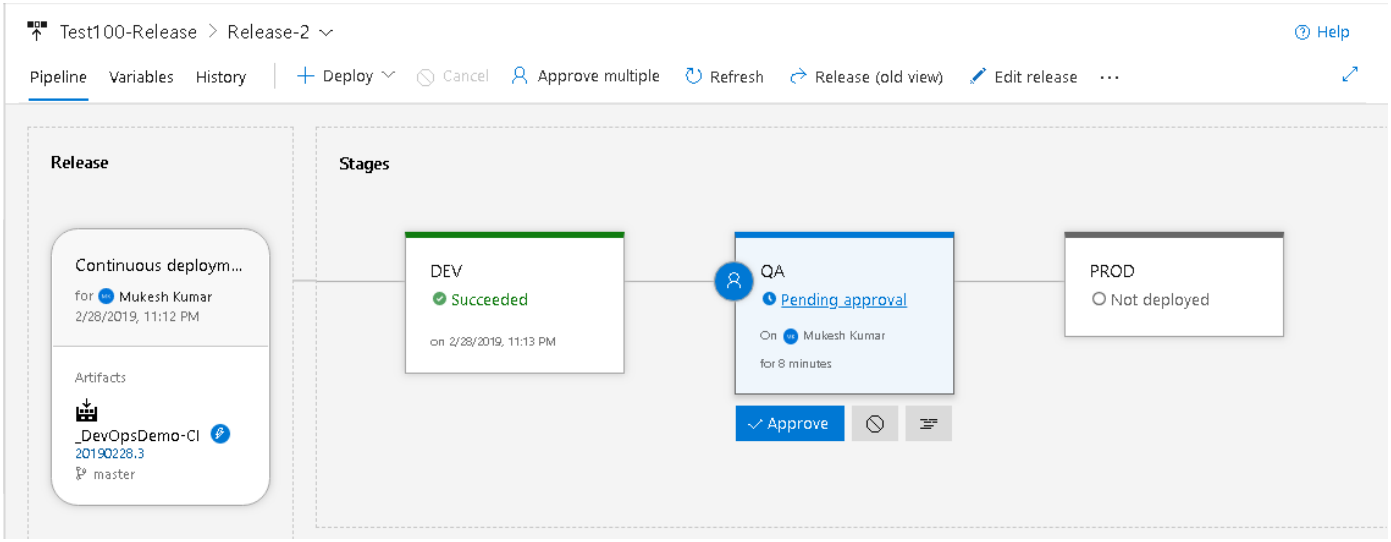
Let's move to **Release Pipeline** for this **(TechHubOrg > DevOpsDemo > Pipeline > Releases > Test100- Release)**. Here we will find a new release has initiated as **Release 2** and **Continuous Deployment** has started on **DEV**.



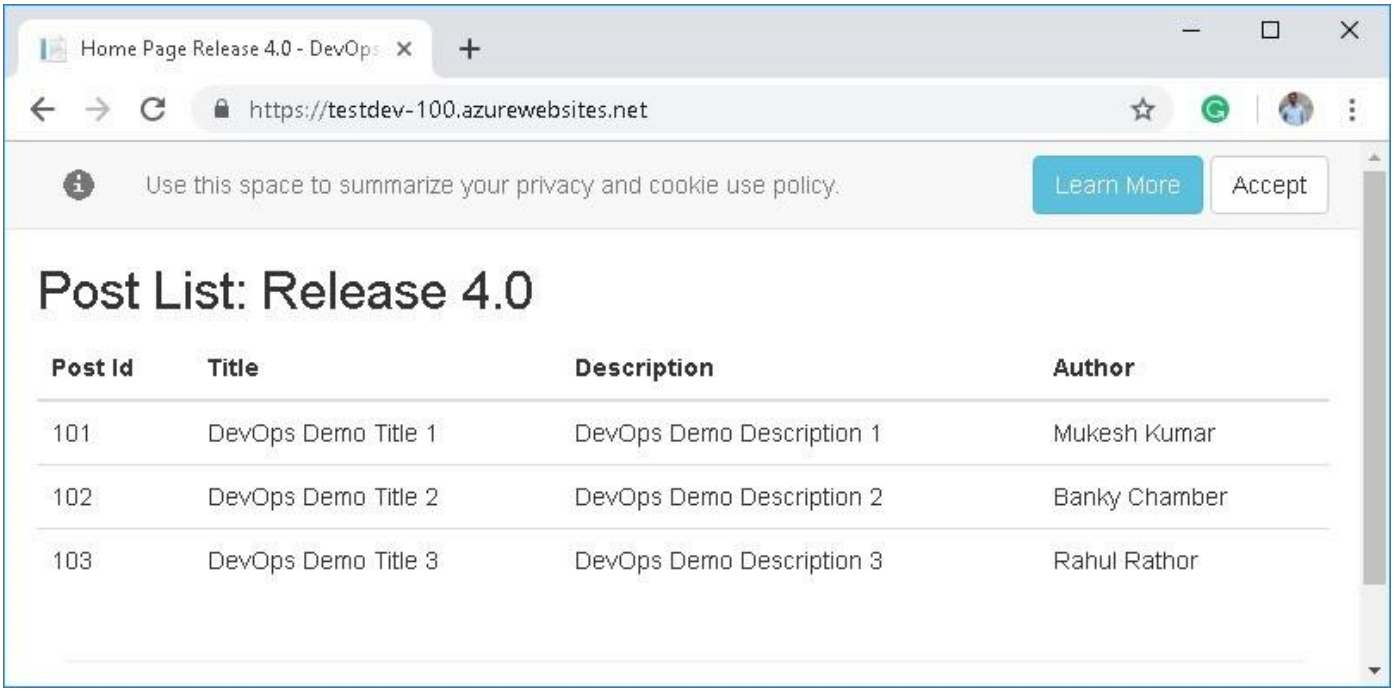
Open the **Release-2** and we will find that **Continuous Deployment** is in progress on the **DEV** environment. Wait for it to complete.



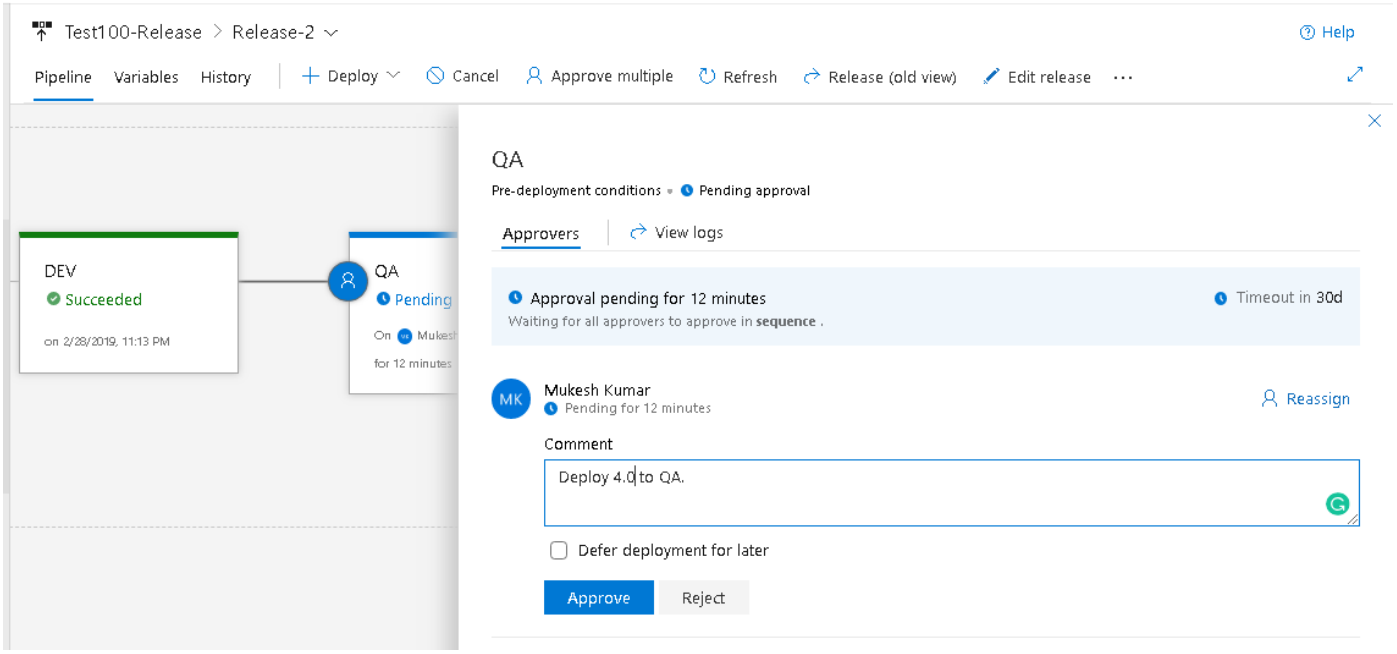
After a few minutes, it will complete the deployment on **DEV** and move to **QA**. But as we have configured that, approval is required before deployment on QA. It will ask for **Approval**.



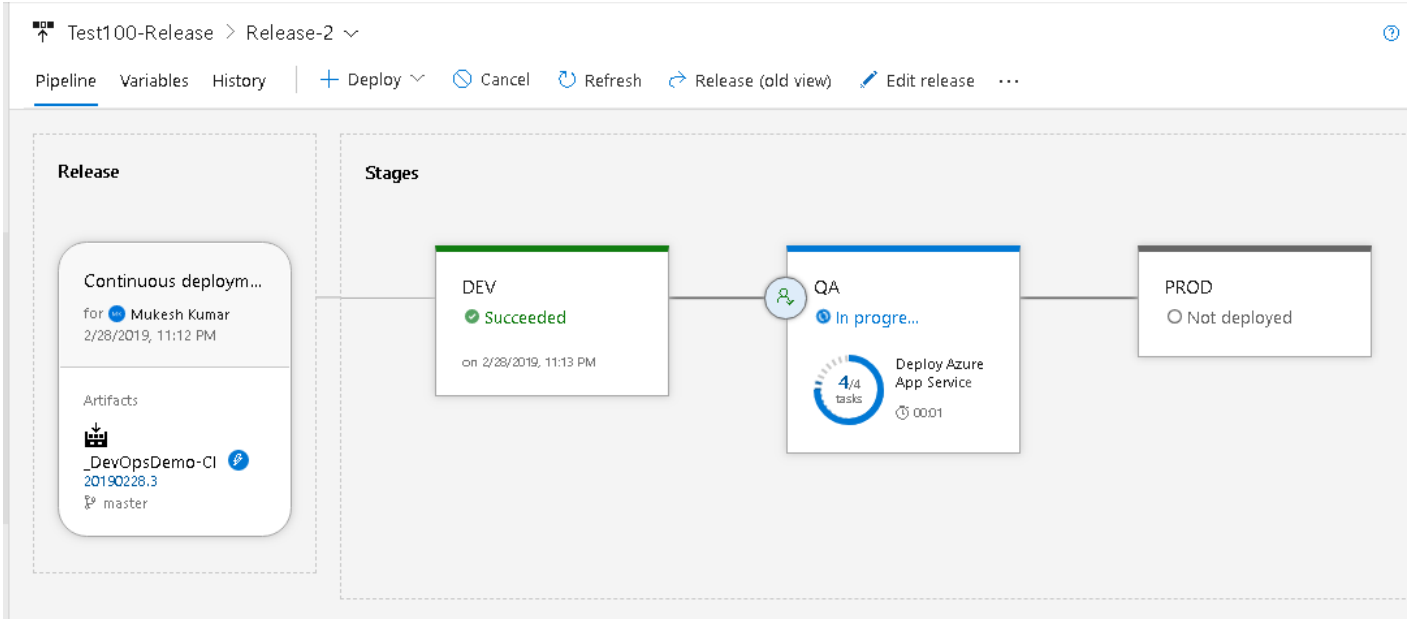
Before approving for QA, check what changes has deployed on **DEV (TestDEV-100 App Service)**. So, open the URL for DEV server as **https://testdev-100.azurewebsites.net** and here we go. Good, we have deployed **Release 4.0** to the **DEV** server successfully.



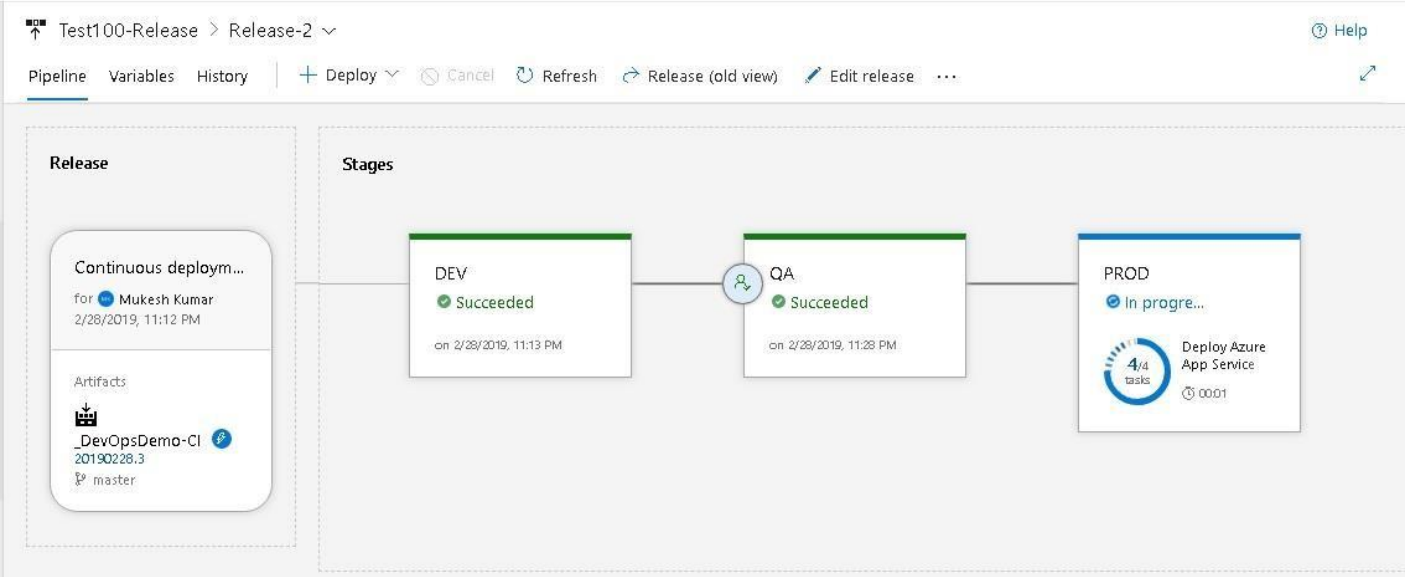
Now, it's time to approve the pending approval for **QA**. So, click on Approve. It will ask for comment. Provide a comment like **'Deploy 4.0 to QA'** and click the **Approve** button.



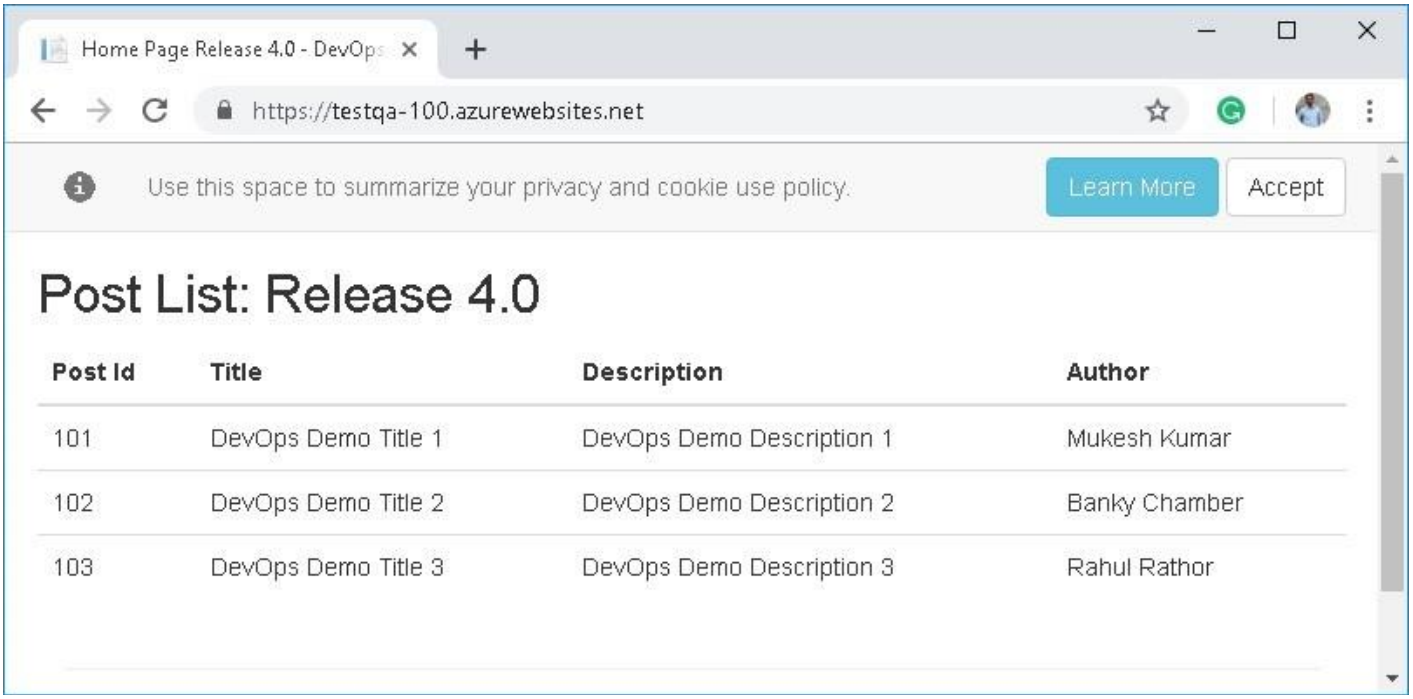
As we have approved it, it will start the deployment on the QA server, as we can see with following image. Continuous Deployment on QA is in progress.



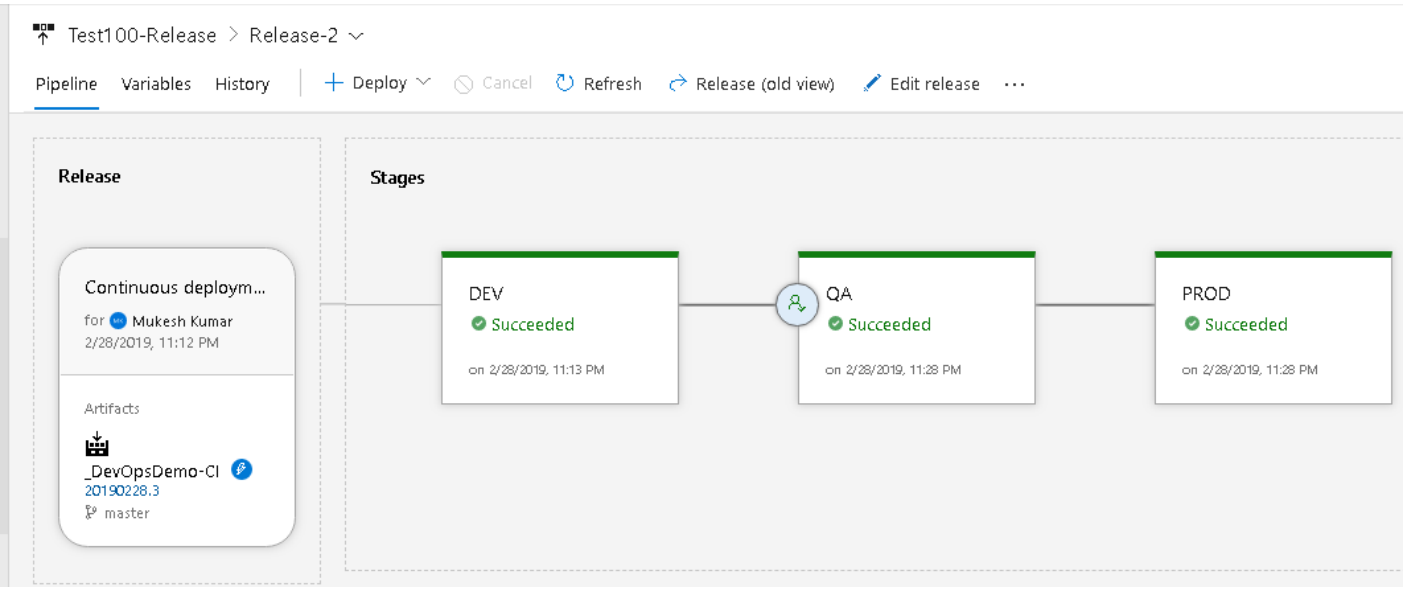
After a few minutes, it will complete the **Continuous Deployment on QA**.



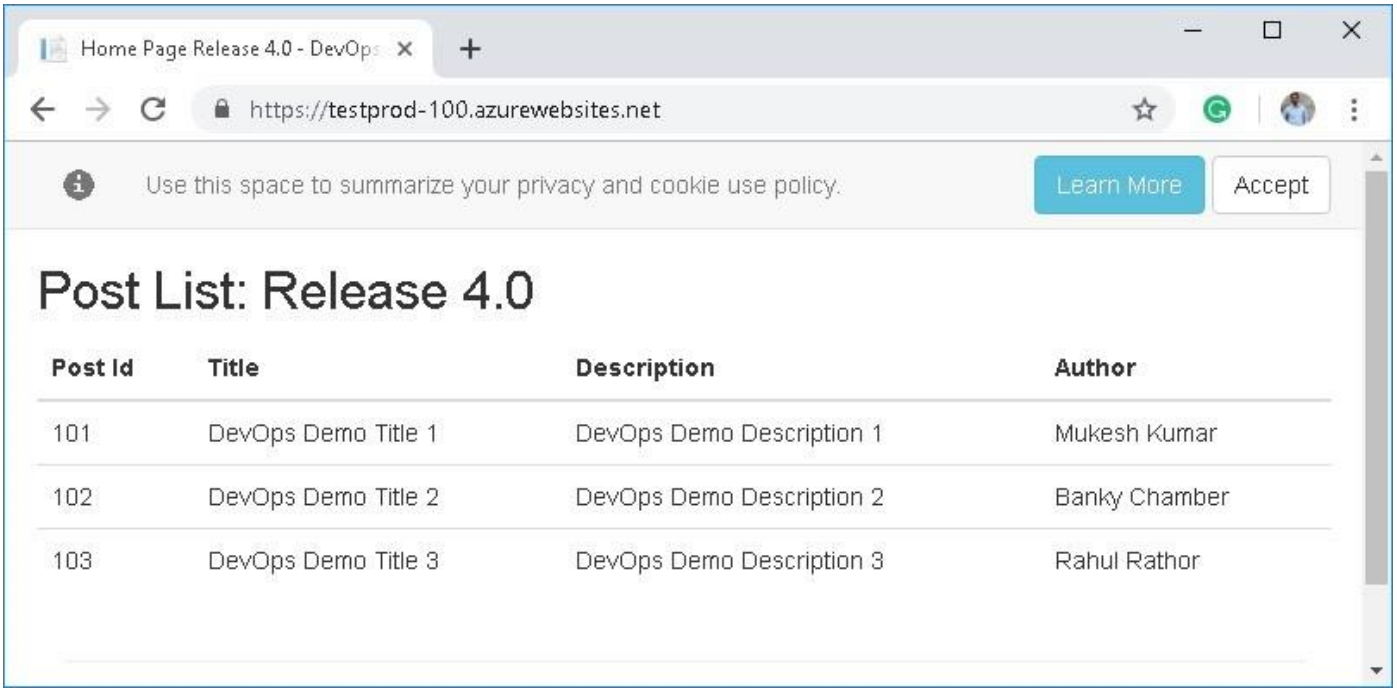
Open the QA environment **(TestQA-100 App Service)** URL **https://testqa-100.azurewebsites.net** in the browser. Great, **Release 4.0** is also deployed on **QA**.



As we didn't configure any approval before **PROD** deployment, after deployment on QA, it will auto deploy on PROD. So, **Release 4.0 has deployed to DEV, QA and PROD as well**.



Open the PROD server and see whether release 4.0 has deployed or not. Open the **PROD** environment URL as **https://testprod-100.azurewebsites.net**. Great, we have deployed **release 4.0 on PROD** as well.



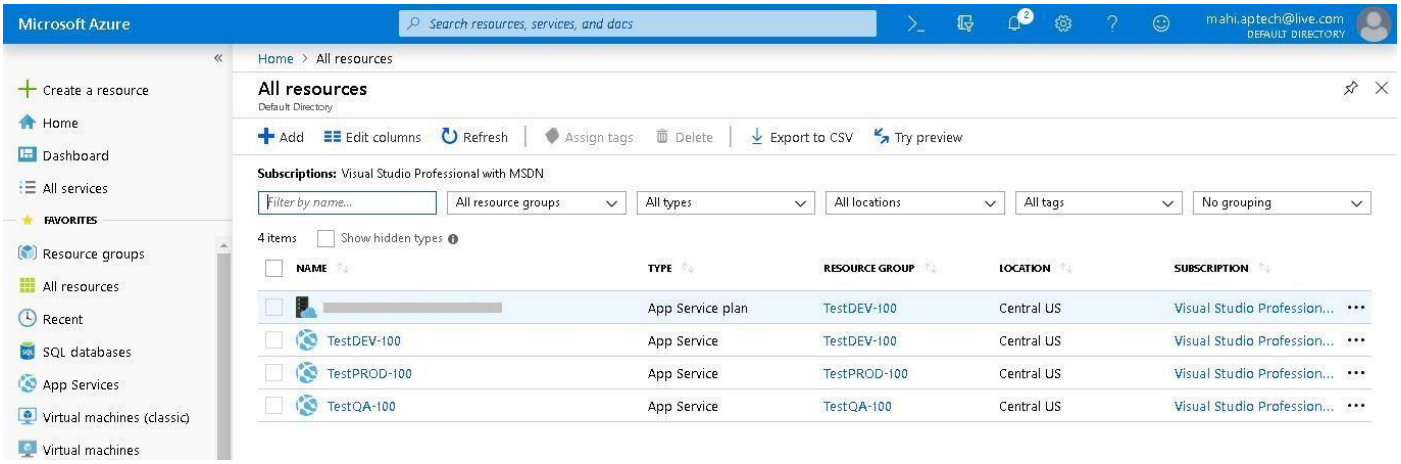
Chapter 9.Add Slot

Go to https://portal.azure.com and All Resources page. Here we can see listed all the available resources which we had created so far. Here we would like to add one more stage between QA and PROD. Actually, the reason for using the **Staging** environment is that we don't want to directly publish the changes from QA to PROD. First we will verify the changes and functionality and if everything is working as expected then we will move the changes on PROD using **SWAP**.

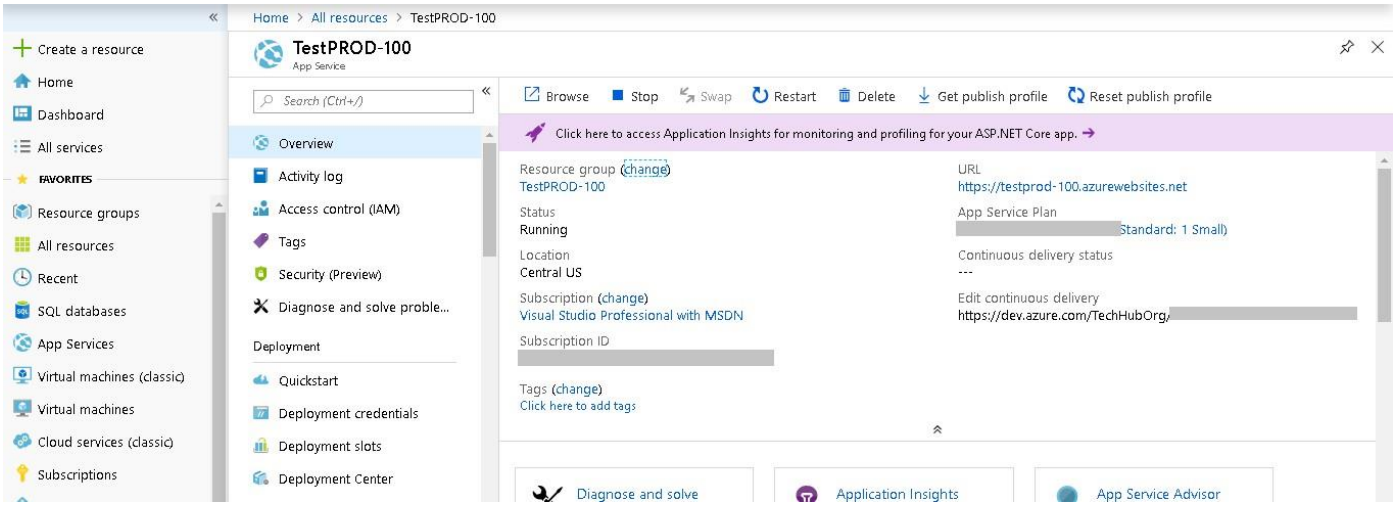
Slots allow you to deploy your application to a separate live app service, warm it up and make sure it's ready for use in production, and then swap the slots to provide seamless traffic redirection. You can slot swap manually (in the portal or command line) or you can automate the slot swap with Auto-swap or in a script.

SWAP is basically a feature of Azure DevOps where we use two different environments for deployment, where one is slot. Using SWAP, we can swap the production environment with some staging (slot) environment with 0 downtimes. So, it is a very great feature for PROD deployment.

As we are going to create one more staging environment just before PROD, open the PROD app service from the **All Resources page**.

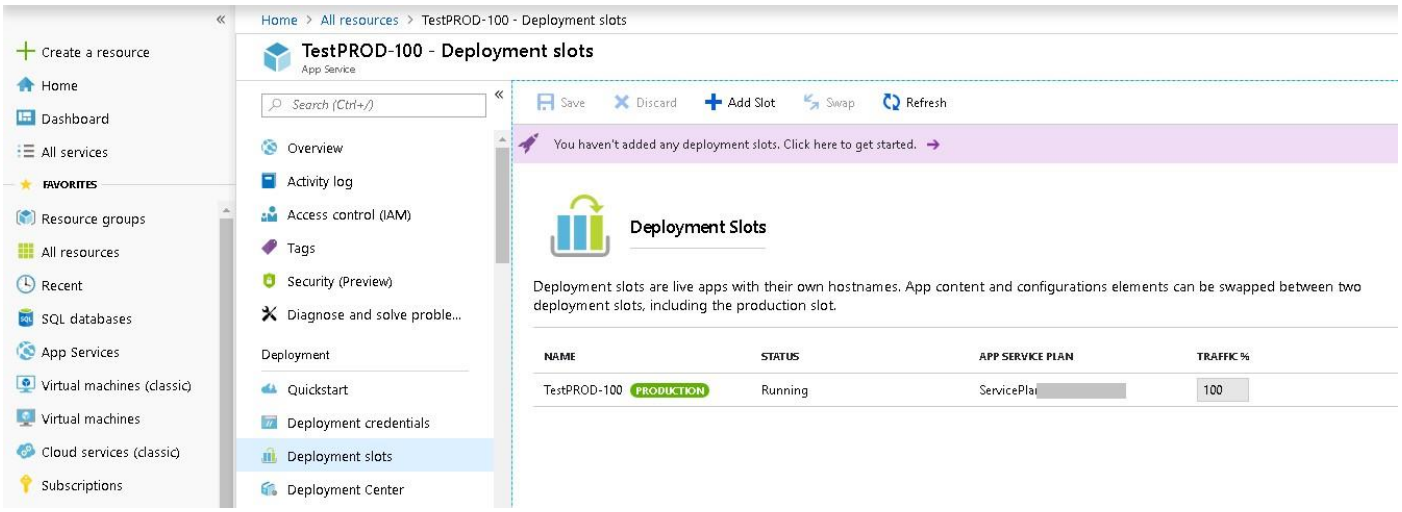


Once we open the **PROD (TestPROD-100)** environment, here we can see all configuration related to this App Service. Apart from this, we have an option to create **Slot** in the Deployment section as **'Deployment Slots'**. Using the Deployment Slots option, we can create a new slot **(staging environment)**. So, click on the **Deployment Slots**.

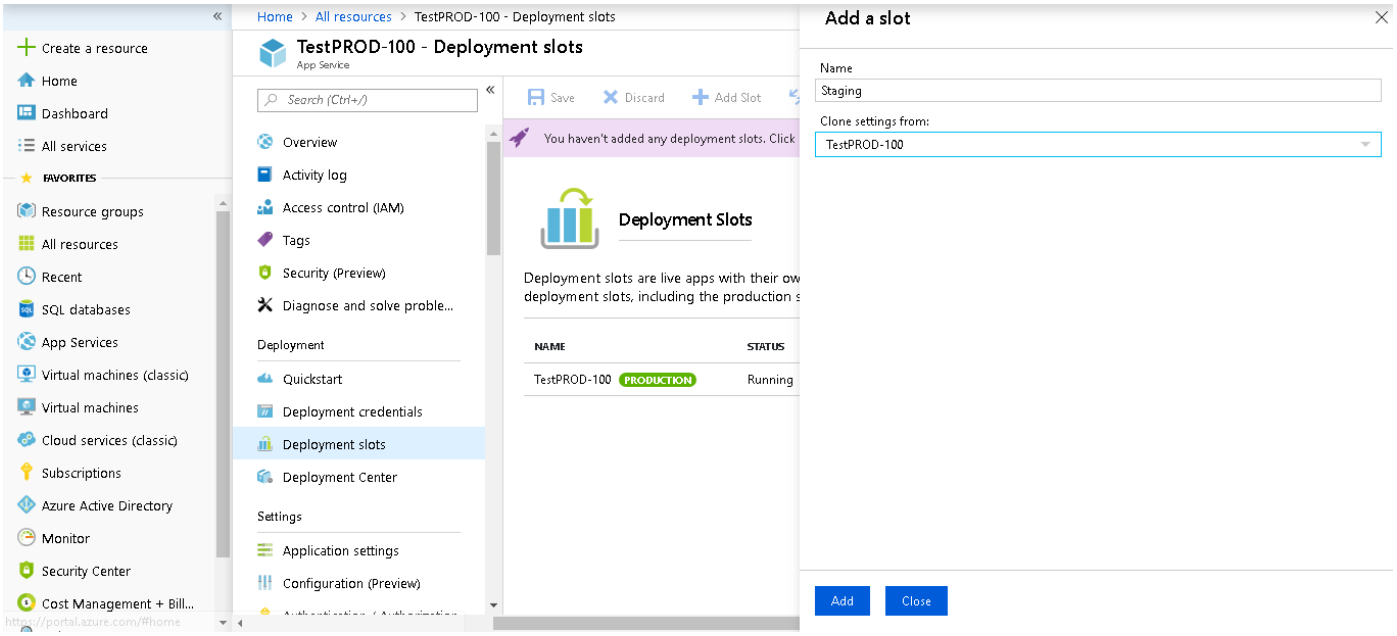


The next page is the deployment slots page. Here we can see that the **PROD** environment is running with **100% traffic**. It means all the traffics which is going to access the PROD environment will hit the **TestPROD-100 App Service**.

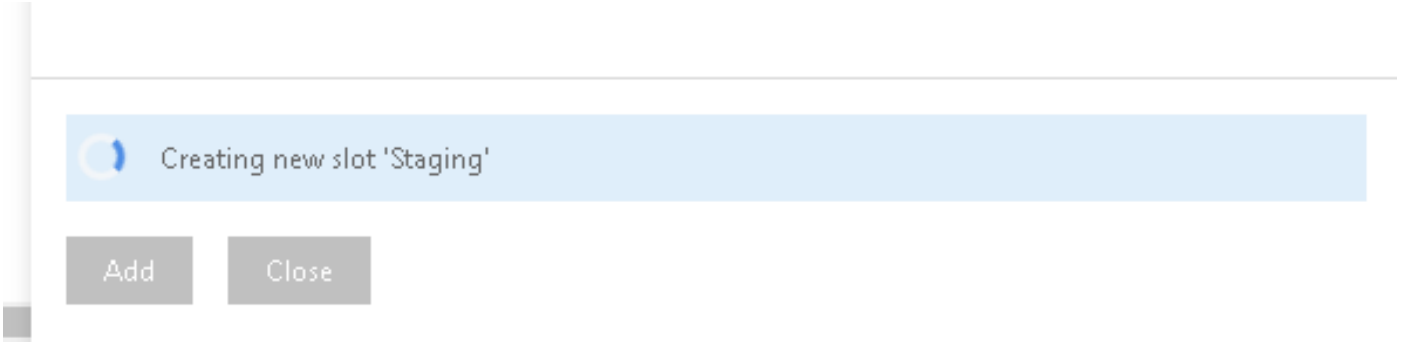
At the top, we have one button as **'Add Slot'**. For adding a new slot, just click on **'Add Slot'**.



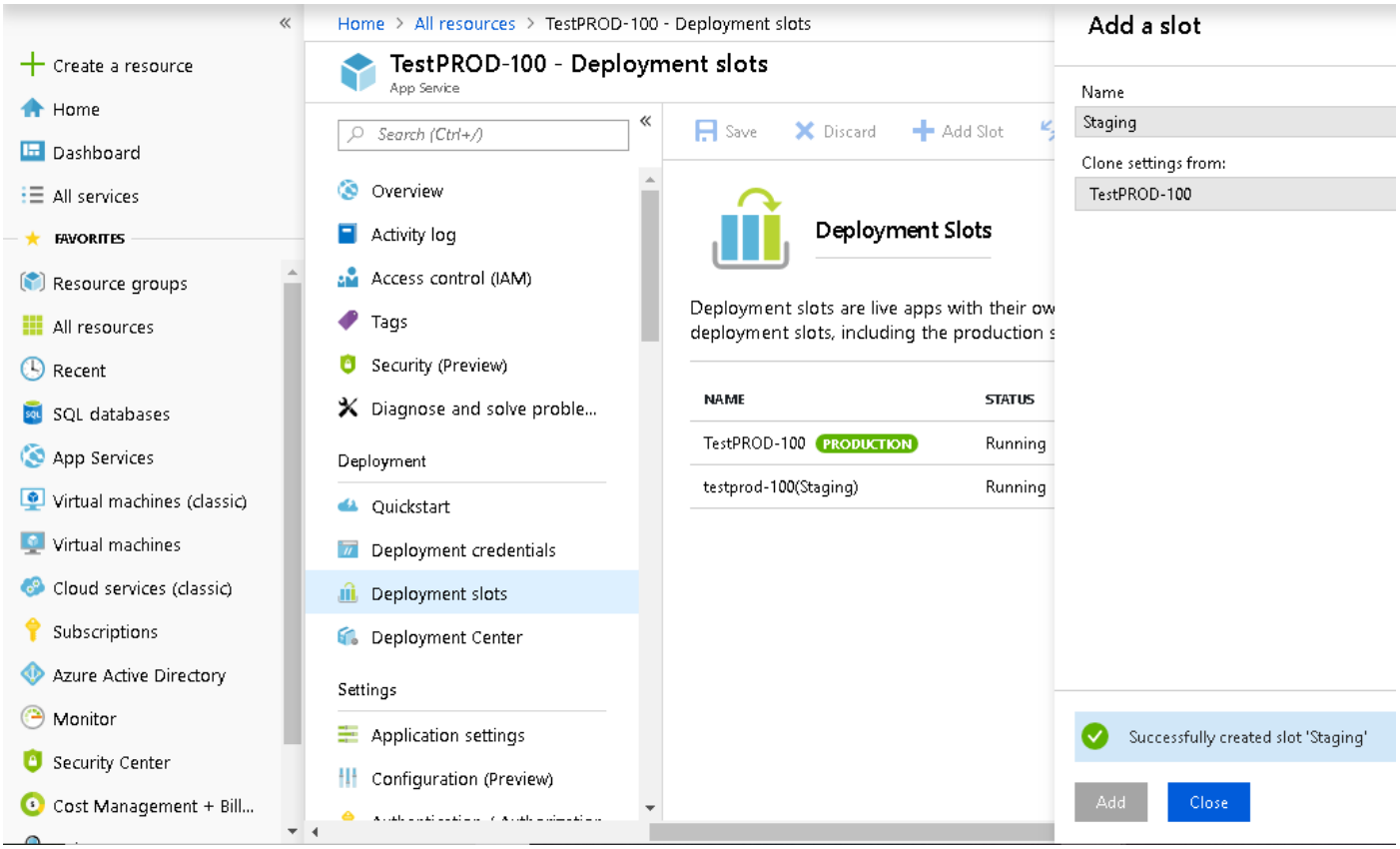
A dialog window will open in the right side and ask for the name of the new slot. Here we will give the name as **'Staging'** and use the **TestPROD-100** for the **clone** setting. It means, a new **Staging** environment will be created to clone the **PROD** environment. Click the **Add** button for adding a **new Slot**.



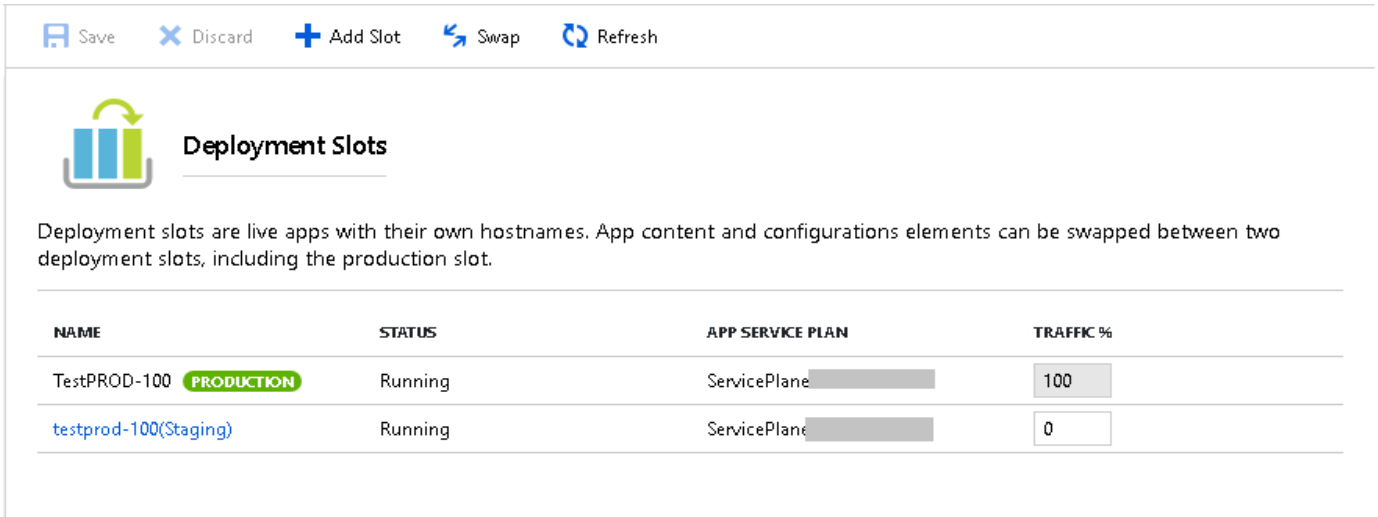
It will take a few minutes to create the new **Staging environment**.



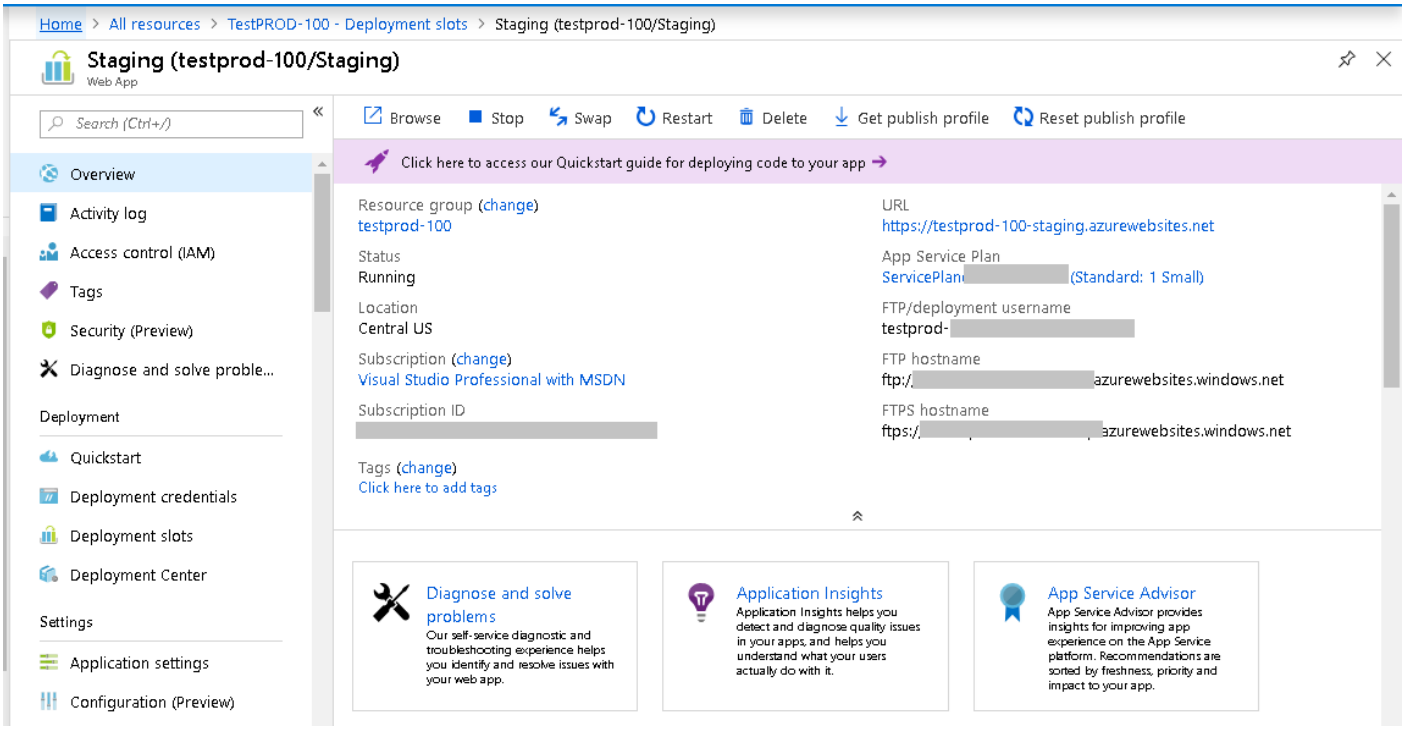
Once it is done, it will show a **Success** message as follows. Just close this dialog window.



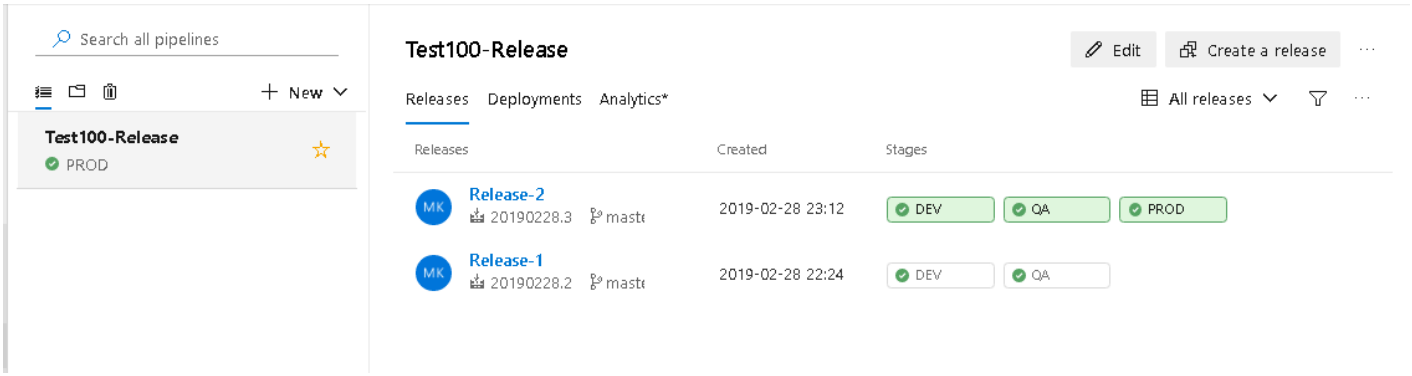
Now, we have one Staging environment for **TestPROD-100 as TestPROD-100(Staging)**. It is also running but right now, there is no any traffic on this.



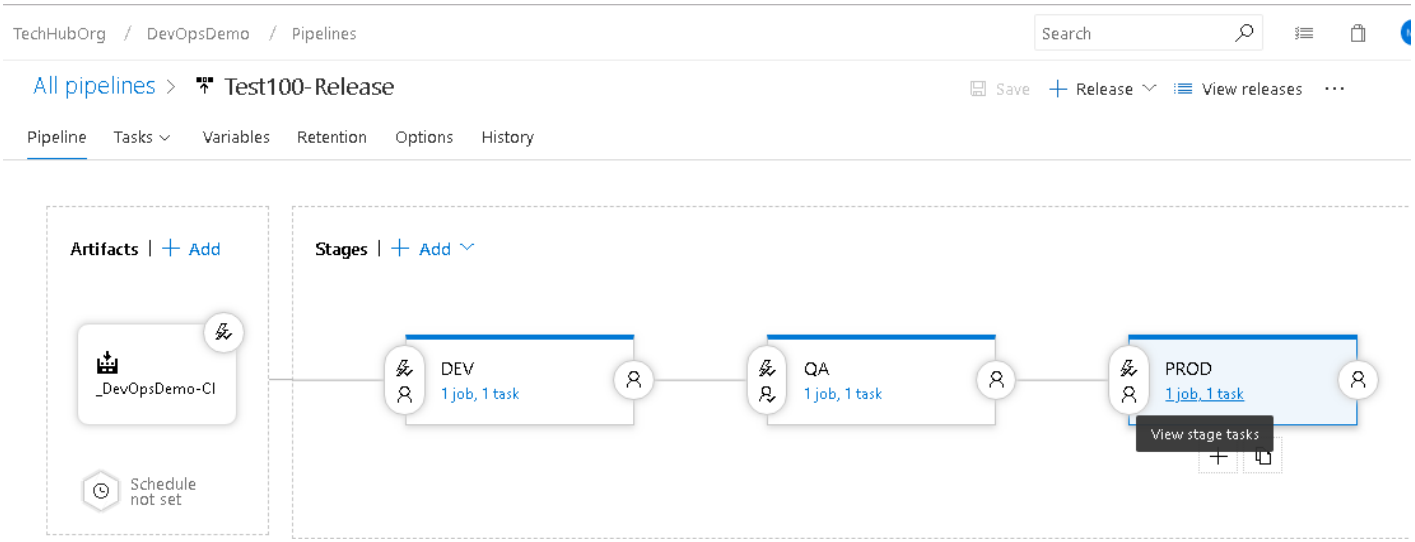
Click on the name of the staging environment, **TestPROD-100(Staging).** It will open the configuration page for this Staging environment. Here we can see the URL for the staging environment which will be used to access it. We have several options for this Staging environment like **STOP, SWAP, RESTART, DELETE** etc.



The staging environment is available now. Let's configure it in the **Release** pipeline. So, open https://dev.azure.com/TechHubOrg/DevOpsDemo and release pipeline as **Test100-Release** and **Edit** it.

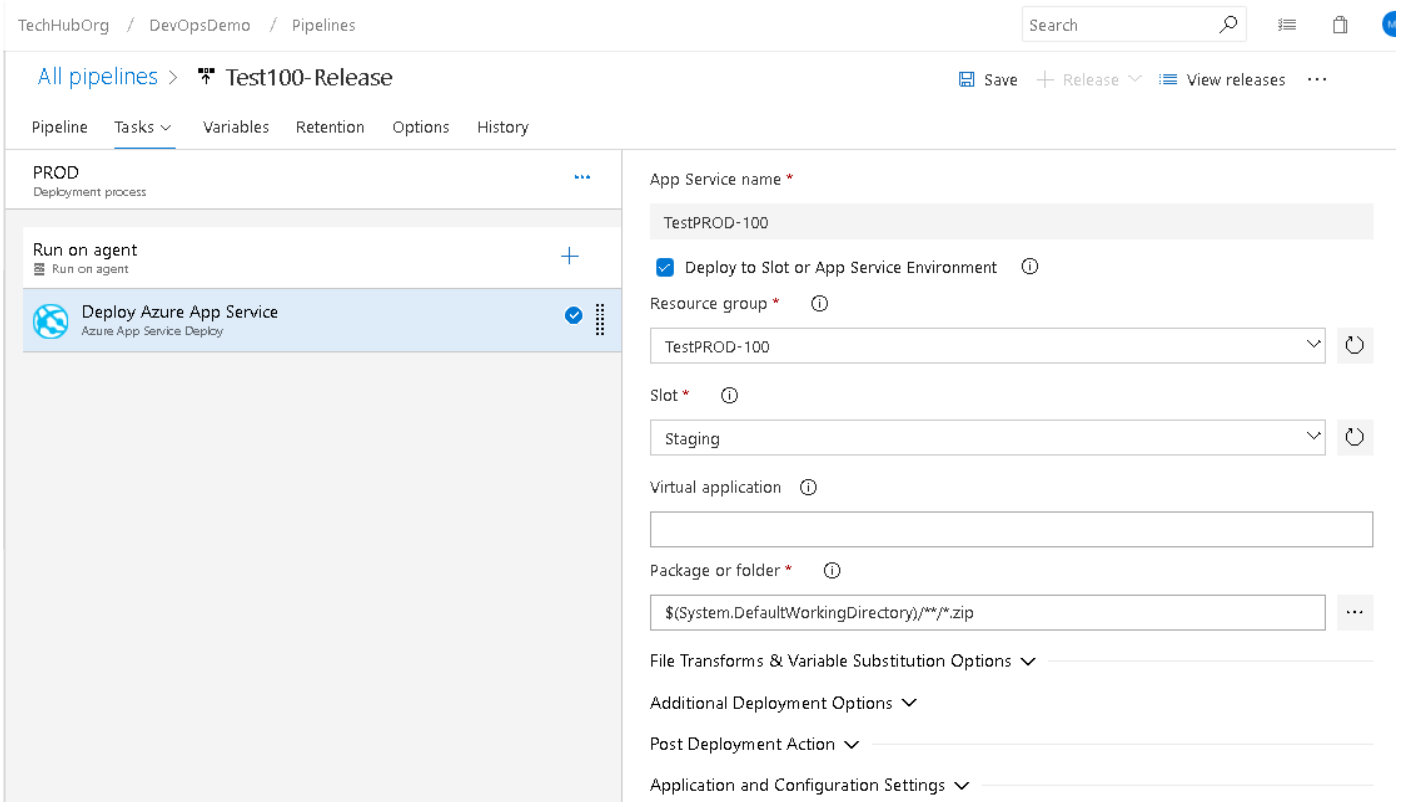


Once **Test100-Release** opens in **Edit** mode, just click on **PROD stage**. Actually we will configure the Staging environment with PROD. If any new **build artifact** will be available and is going to deploy on **PROD** after **QA** then it will first deploy on the Staging environment. Later the user can use the **SWAP** functionality to **SWAP** the production environment with the staging environment.



In the PROD stage, go to the **Tasks** tab and it will open a dialog window in the right. Here check the option for **'Deploy to Slot or App Service Environment'** and select the **Resource Group name as 'TestPROD-100'**. The next option is to **choose the Slot**. So, in the Slot dropdown, we have to select the **'Staging'**.

So far, so good, we have finished implementing the slotting in the PROD environment. We don't need to change any other options for now. So, now just click on the **SAVE** button to save the **Release pipeline**.

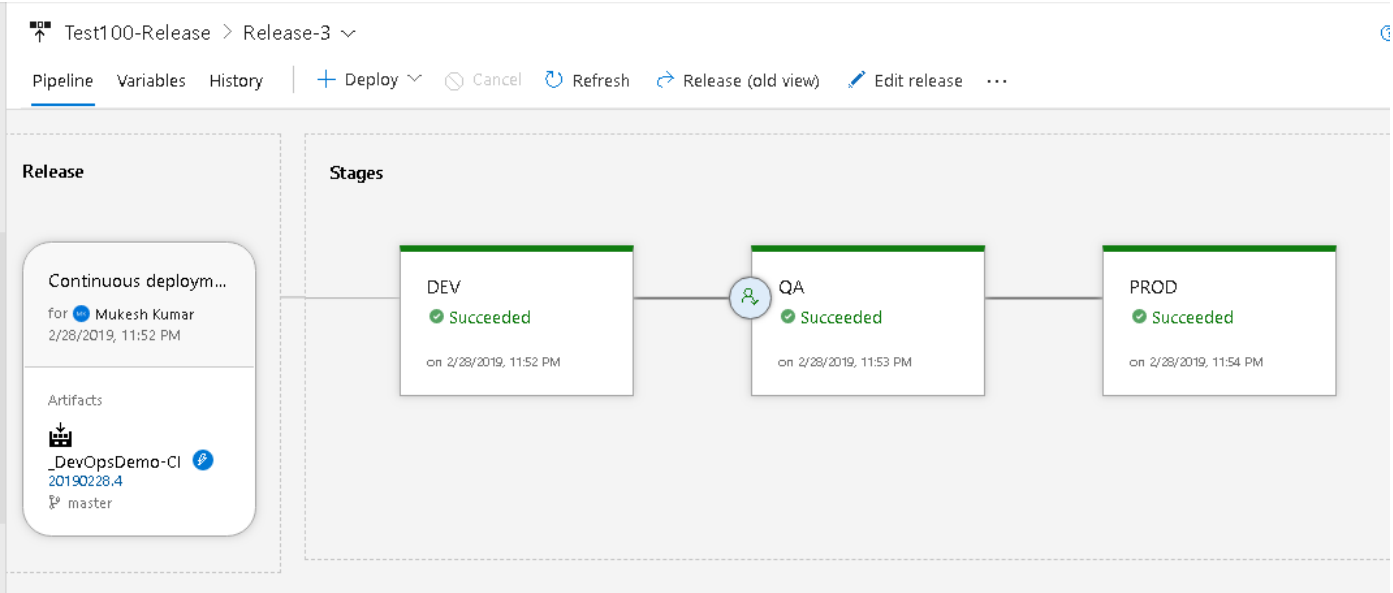


Now we are finished adding the **Slotting functionality with RPDO environment**. So, let's make the changes in the code and check in the code. When we check in the code, what should happen? First, it should make the build artifact in Build Cycle and deploy the artifact on the DEV server in the Release cycle and after approval, it should deploy on QA and then deploy to the Staging environment. It should not deploy anything on **PROD**. We will deploy on the PROD using the **SWAP** feature.

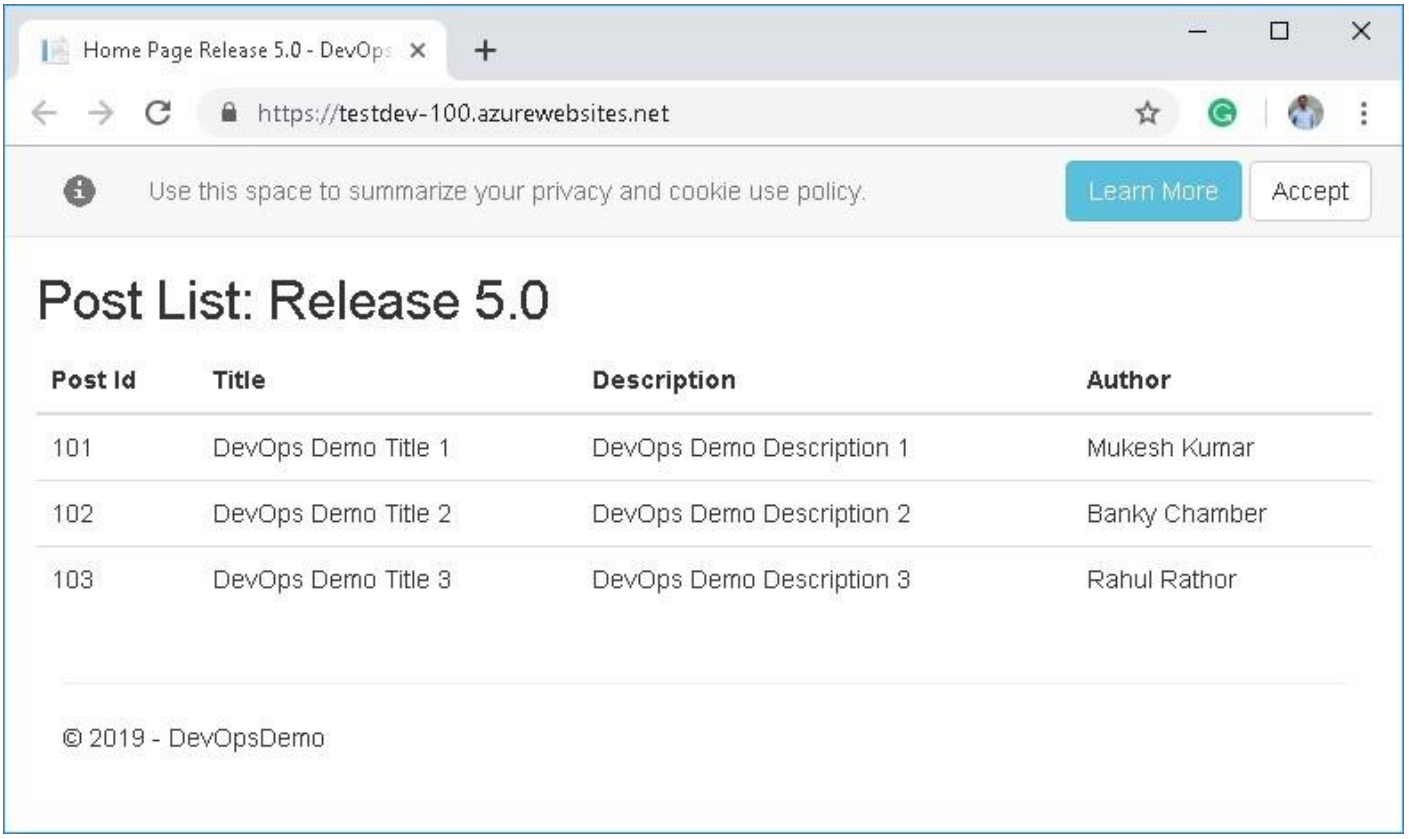
Chapter 10.Run and Test Azure DevOps Pipeline

So, open the **Visual Studio 2017 or higher version** one more time and open the **Index.cshtm**. Here we will only change the version number in the page title and post title. So, change the version number from **4.0 to 5.0**.

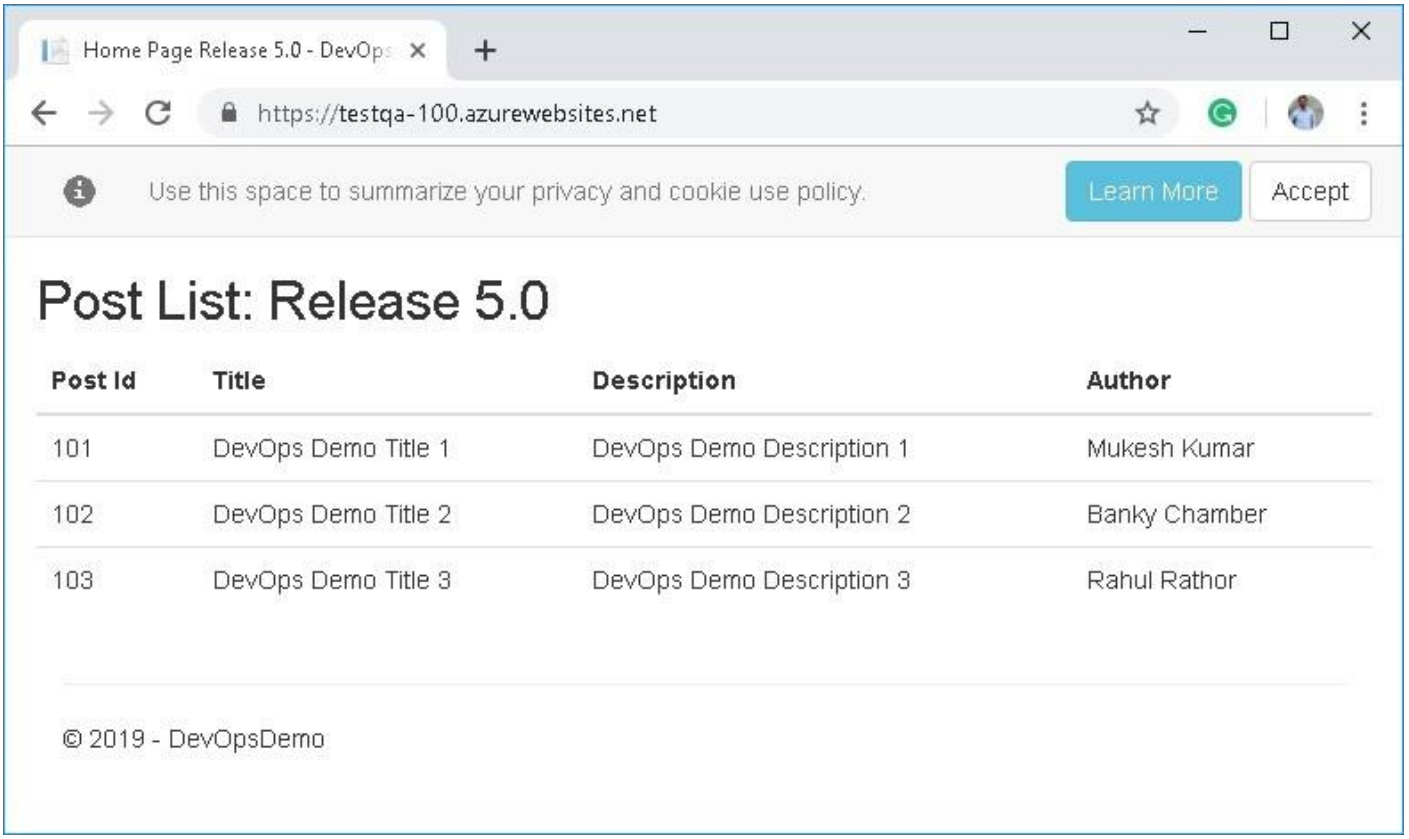
Save the changes and go to **Team Explorer** and **check in the code** as we have done previously. Once the code checks in then wait for the completion of **Azure DevOps Build pipeline**. After creating the **build artifact**, it will auto deploy on the **DEV** and ask for approval before deploying on the **QA**. Approve and it will deploy **QA and Staging (PROD)**.



Let check first how **DEV (TestDEV-100)** is working. Here we can see that our latest changes have deployed on the DEV environment as follows. **[Latest Release is 5.0].**

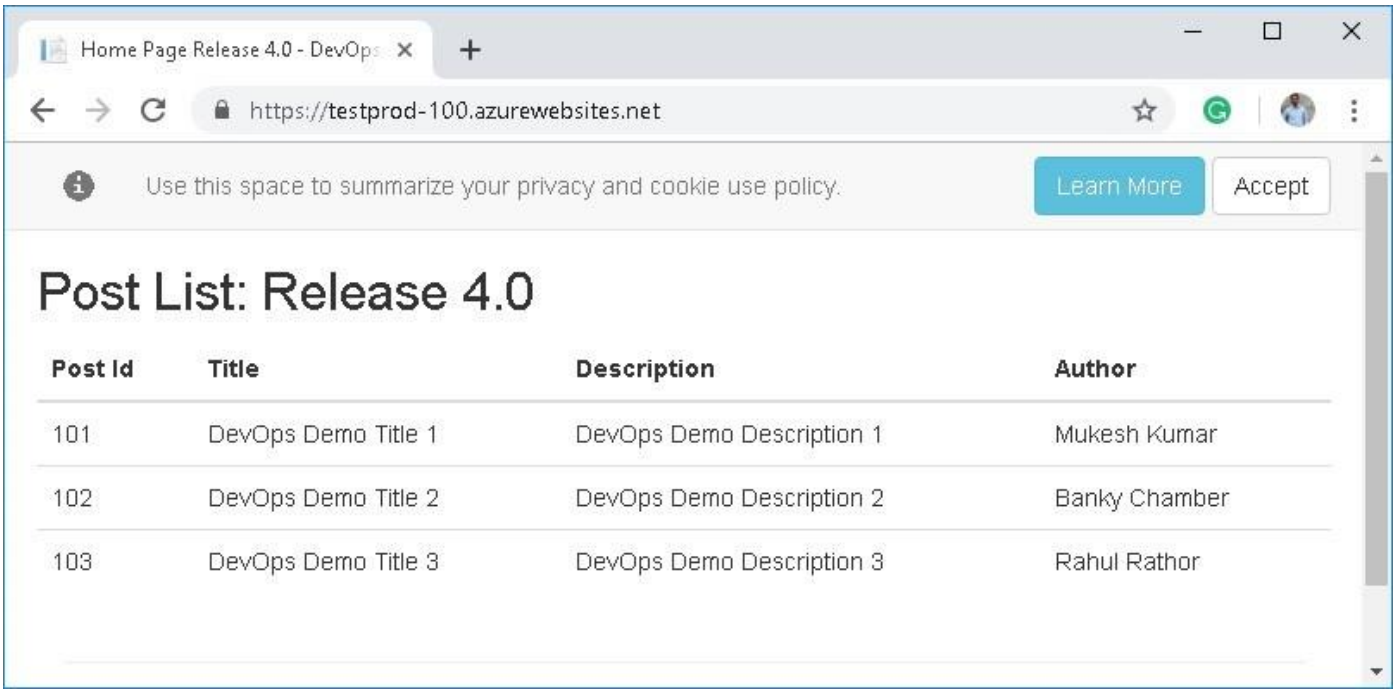


Open the **QA environment (TestQA-100)** using URL **https://testqa-100.azurewebsites.net** and here we can see that the latest build artifact has deployed successfully on the **QA environment** as well.



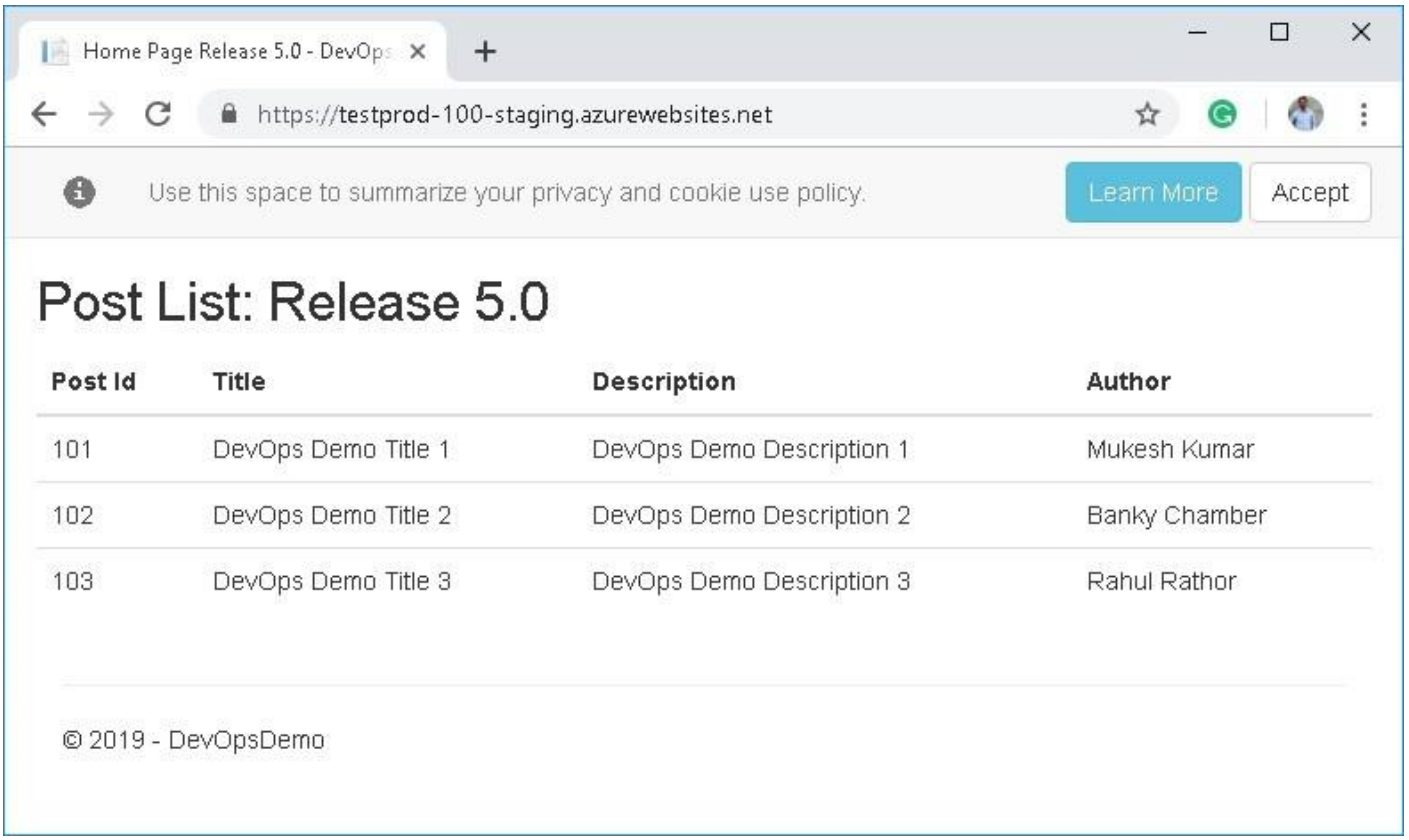
Now, it's time to check if the changes have deployed on PROD or not. It should not be deployed because we have already configured one stage between QA and PROD. The deployment should happen on the Staging environment after QA. So, open the **PROD (TestPROD-100)** environment using the URL **https://testprod-100.azurewebsites.net** and here we go.

Good, our **PROD** environment has **not deployed** the latest changes yet.



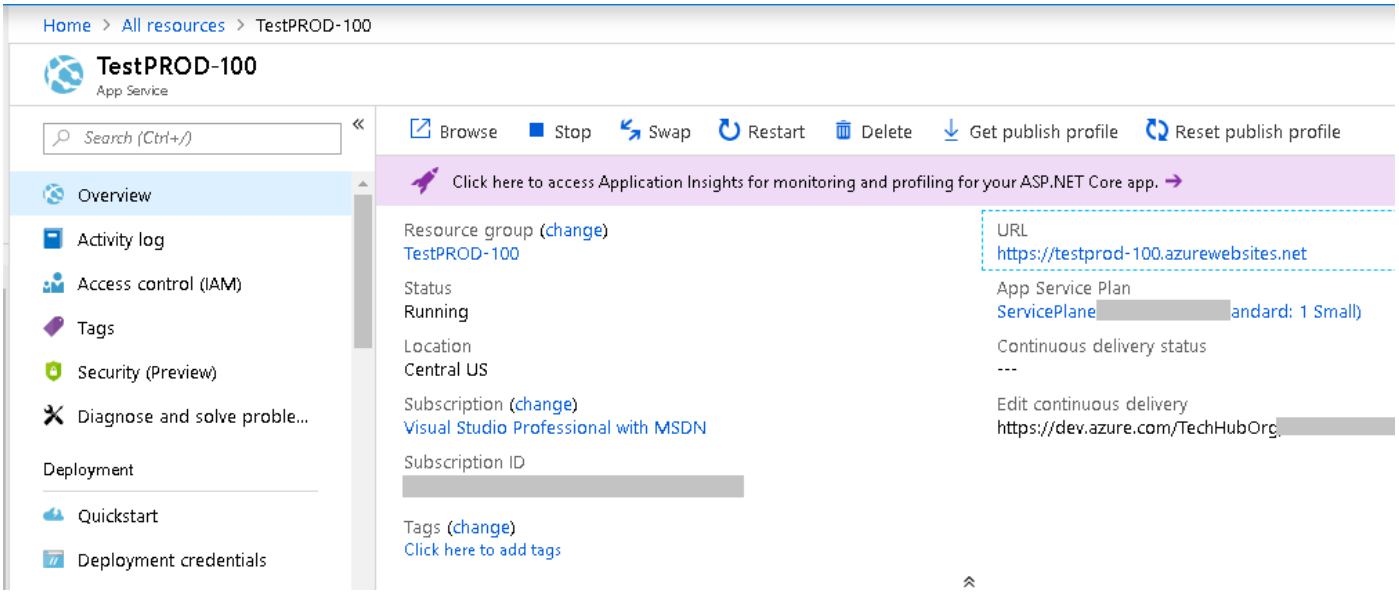
Check the Staging environment using the URL **https://testprod-100-staging.azurewebsites.net** in the **browser** and here we go. Yes, the latest changes have deployed on the Staging environment.

It means, our Release deployment has proceeded in this way: **DEV > QA (After Approval) > Staging > PROD (Not Yet Deployed).**



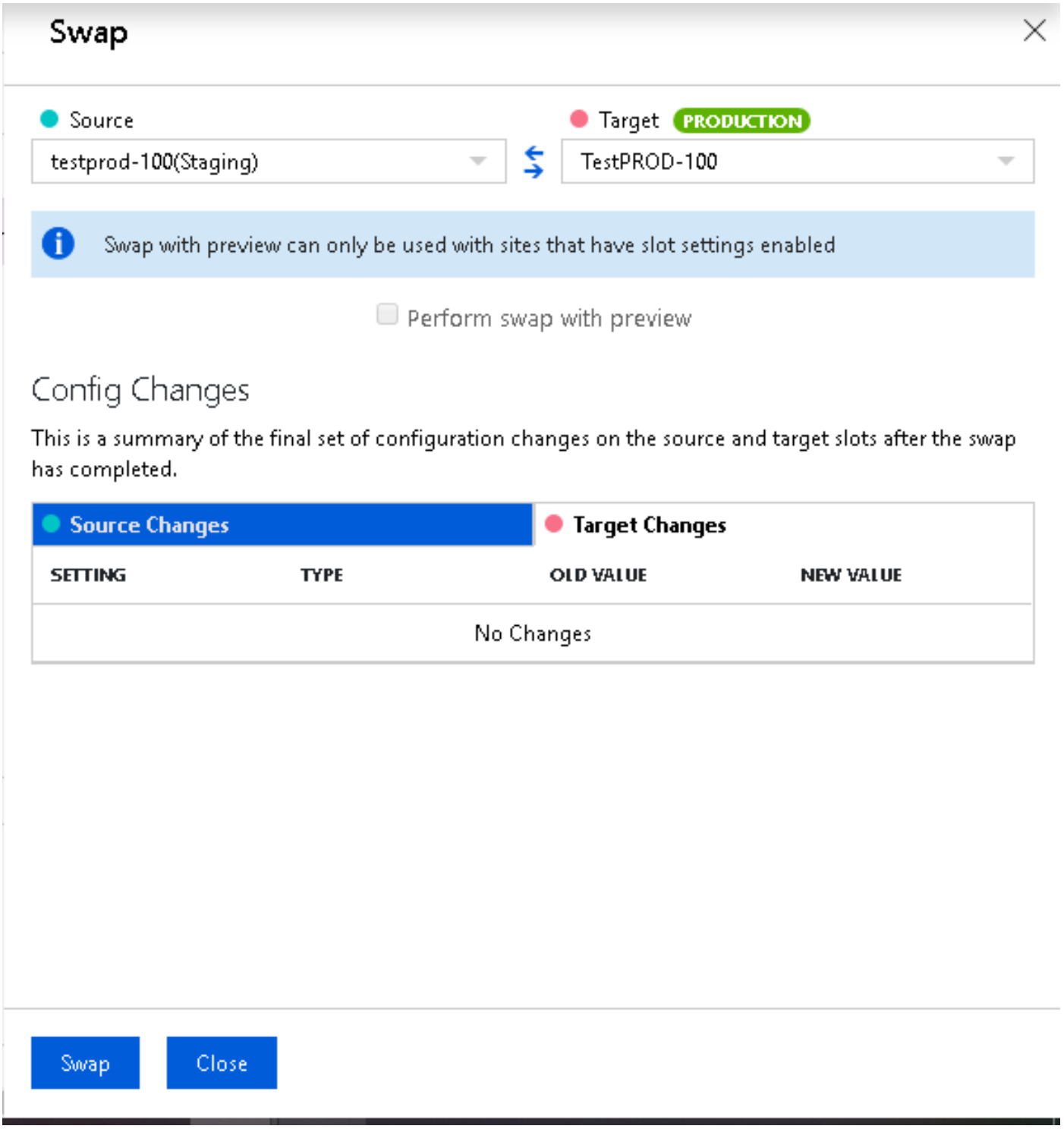
As we learned above, we will use the **SWAP** feature to swap the environment. Here we will **swap between Staging and PROD**. **After swapping, the Staging environment will become PROD and the PROD environment will become the Staging environment.**

Go to **https://portal.azure.com** and open the **PROD app service (TestPROD-100)** from the **All resources in Azure Portal**. Now click on the **SWAP** button at the top just after **STOP**.

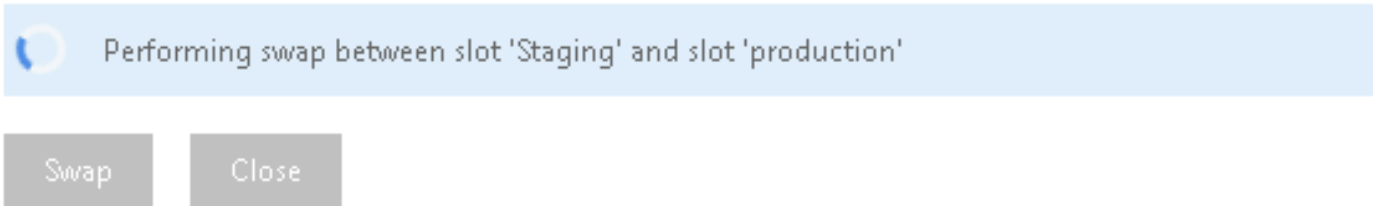


It will open the **SWAP** dialog window in the right. Here we have to provide the **Source environment** and **Target environment**. As our build artifact has already deployed on the Staging environment, so Staging will be part of Source and PROD and where we have to deploy will be part of Target.

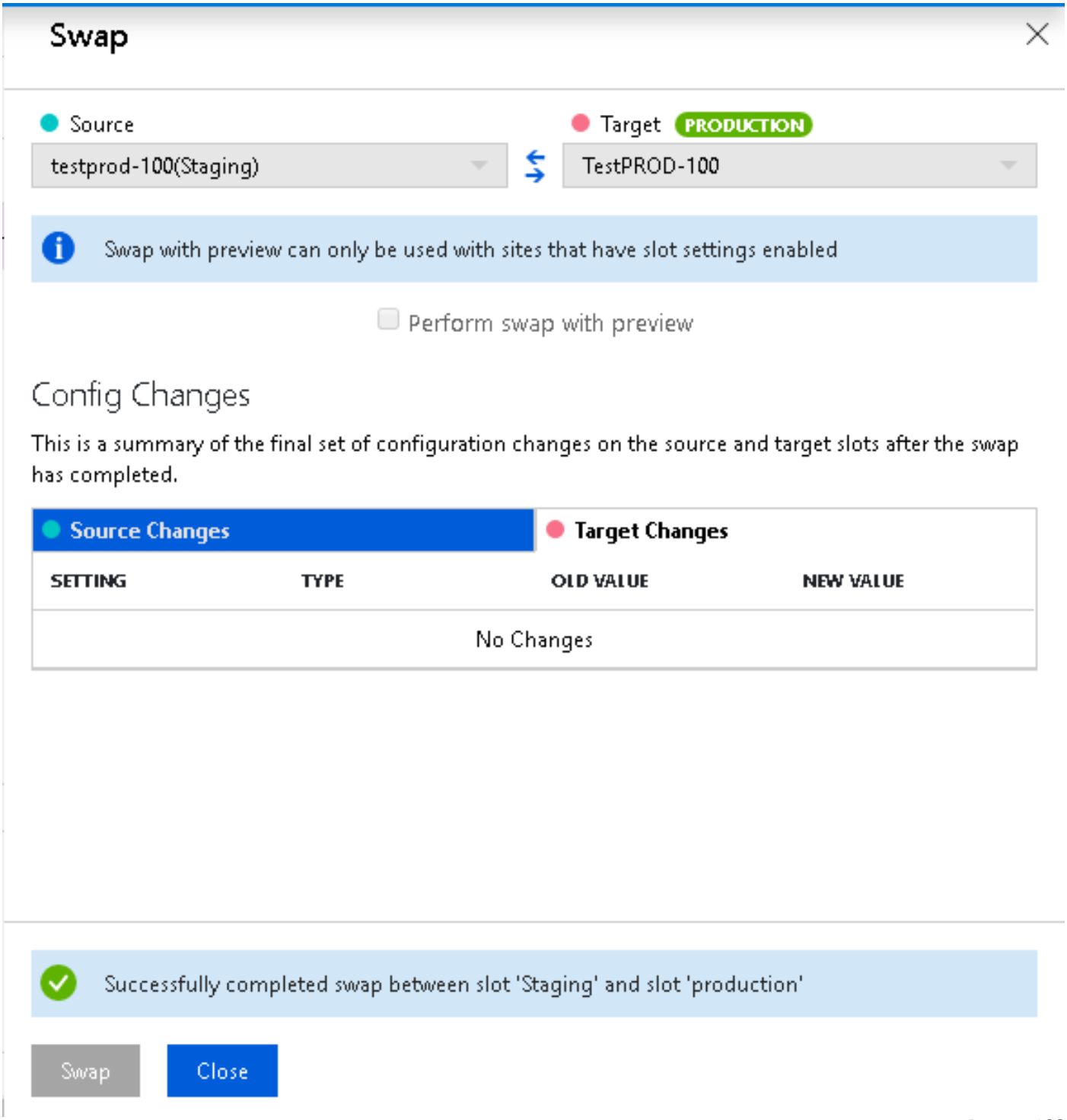
After defining the **Source** and **Target** for **SWAPPING**, we will just click on the **SWAP** button.



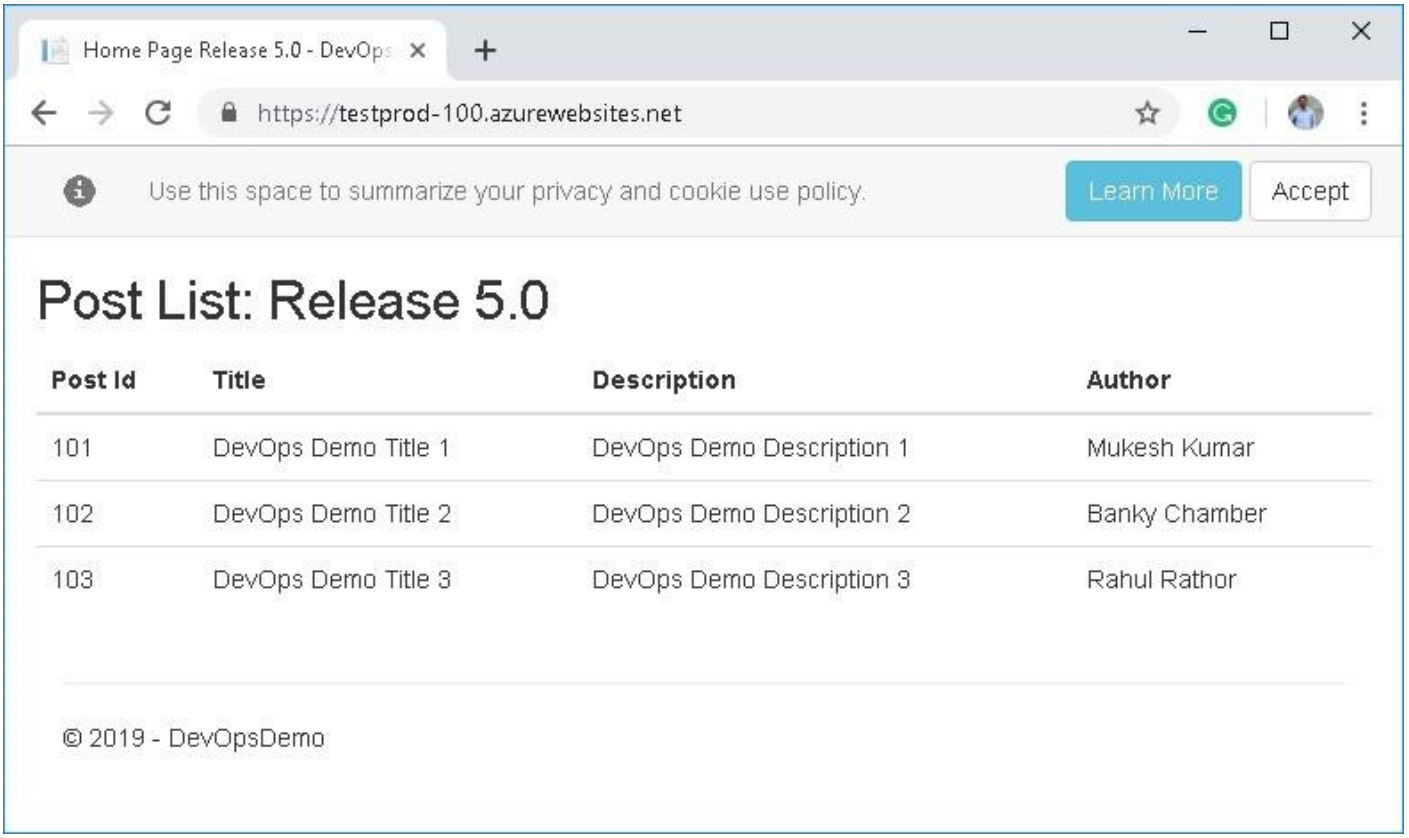
It will take a few minutes to **swap** between **Staging** and **PROD**.



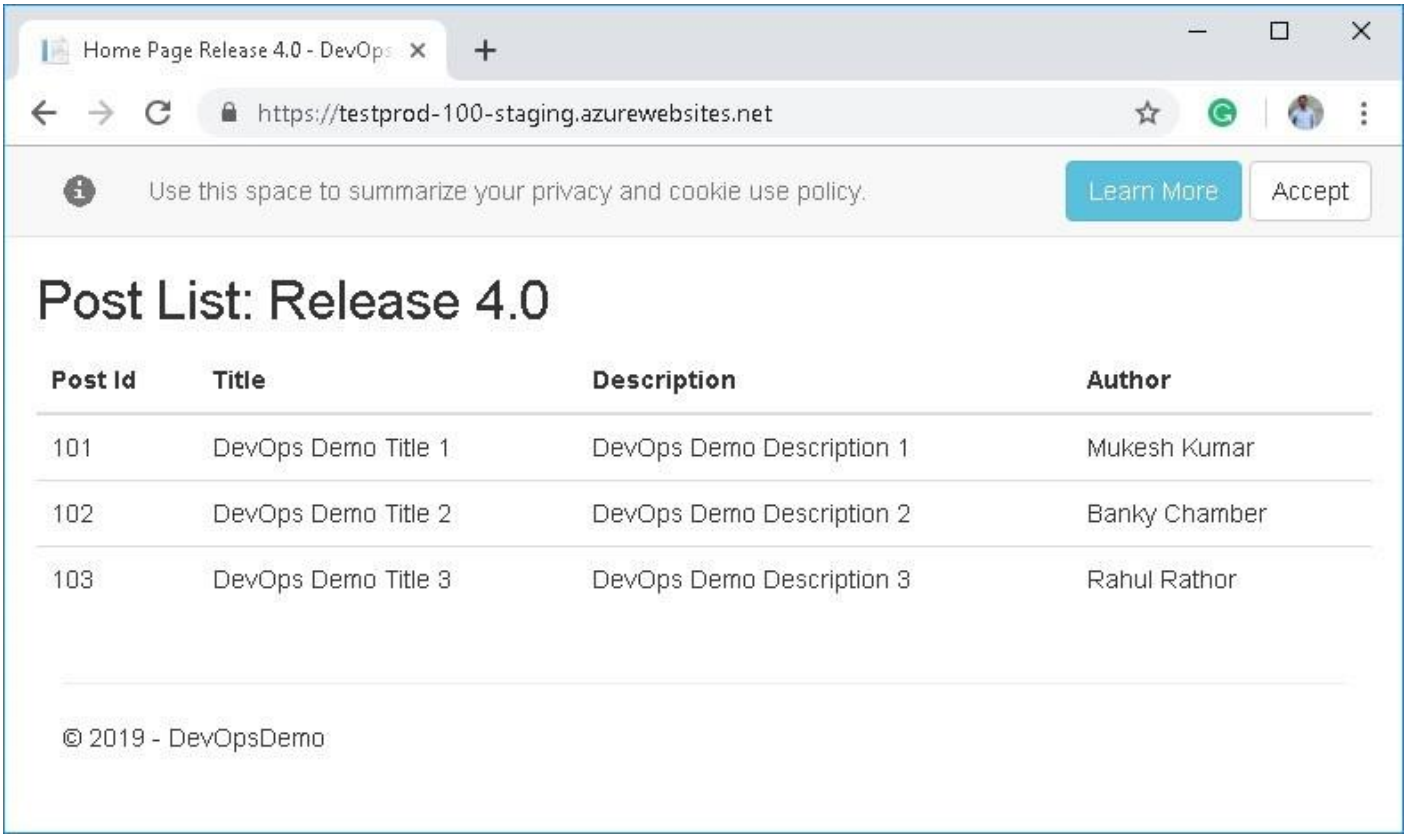
Once **swapping** is complete, it will give us a proper **success** message for confirmation.



So, open the **PROD (TestPROD-100)** environment and see the changes. Good, we have got the latest changes on the **PROD environment as Release 5.0**.



Let's move on and check the **Staging** environment. We will find that earlier **PROD changes have deployed on the Staging environment**. **It means, what used to be Staging has become the PROD and what used to be PROD has become the Staging.**



Summary

DevOps is one of the fastest growing areas for tech jobs. Continuous Integration (CI) and Continuous Delivery (CD) is one of the important operations of project development and management. Not only CI/CD helps automate the build, test, and deployment process but also saves ton of manual work.

In this book, we discussed what DevOps and Azure DevOps are, what CI/CD is in Azure DevOps and how to deploy and manage projects in Github using Azure DevOps.

In DevOps project step chapter, we discussed how to create an ASP.NET Core project, Test project and add it to Github.

Next, we discussed how to create the organization of the project, followed by setting up continuous integration.

We discussed how to create an Azyre App Service followed by Continuous Delivery that had three, Dev, QA, and Prod stages.

In the end, we discussed slots and how to run and test Azure DevOps Pipeline.

So, here we have learned about Azure DevOps and how to implement Continuous Integration and Continuous Delivery using a live Asp.NET Core application, step by step. I hope that you have enjoyed this and learned a lot.