# Exercises: Deployment to Cloud

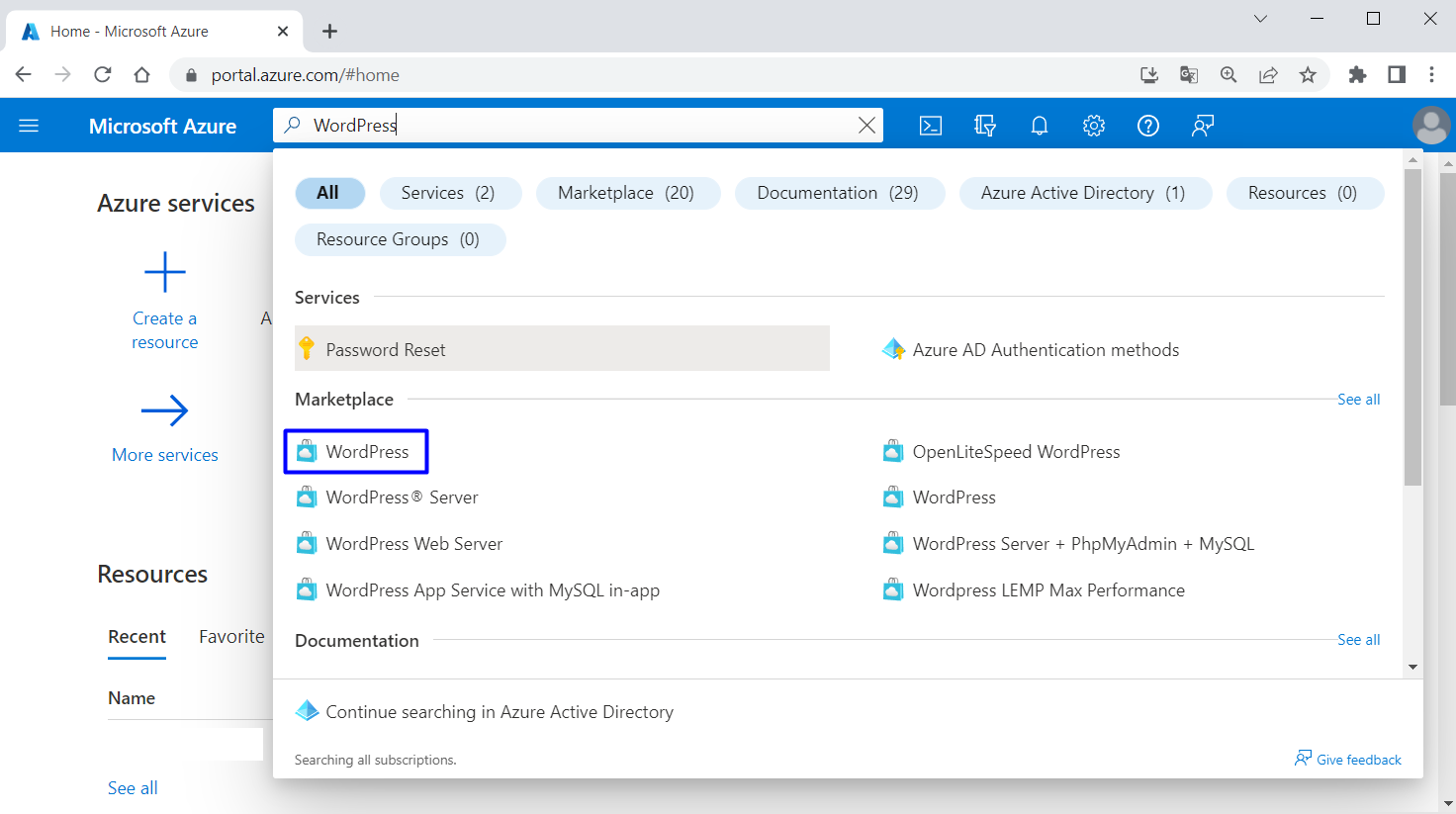
Exercises for the "Containers and Clouds" course @ SoftUni

## Create WordPress Site Using Azure Portal

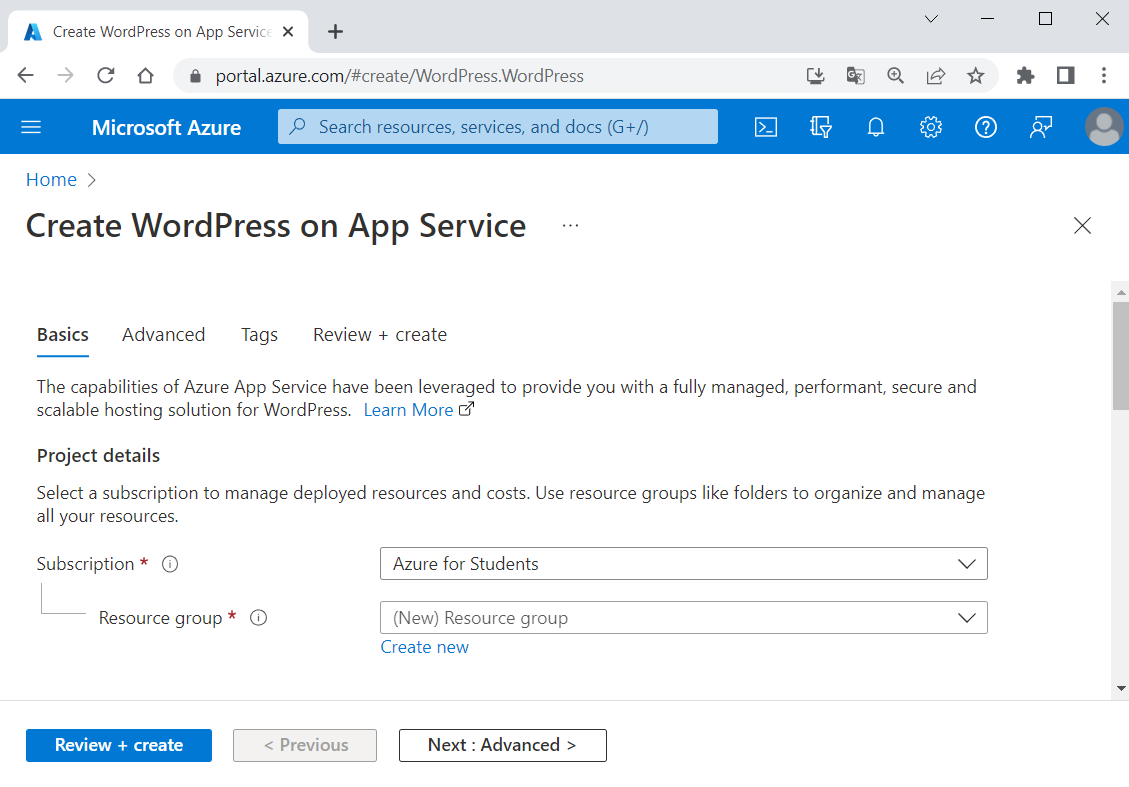
Our task now is to create a WordPress **site** **in** Azure. WordPress can be run on a **few different Azure services**: AKS, Virtual Machines, and App Service. Let's **deploy out first WordPress site** to Azure App Service with Azure Database for MySQL **–** Flexible Server.

### Step 1: Create the Site

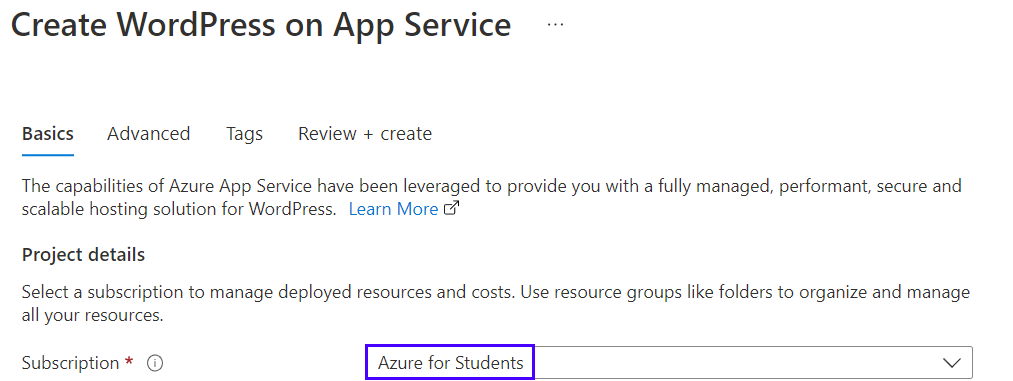
Go to Azure Portal on <https://portal.azure.com> and **log in**. Then, in the **search box**, type "WordPress" and select [WordPress] from the **Marketplace**:



You should be on the "Create WordPress on App Service" **page**. For your convenience, you can follow this link: <https://portal.azure.com/#create/WordPress.WordPress>



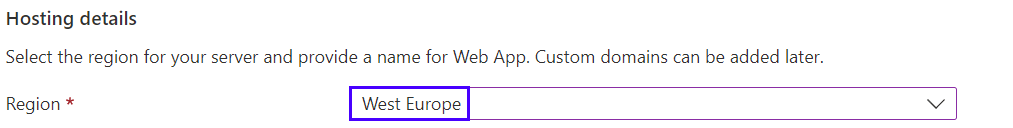
In the "Basics" **tab**, under "Project details", make sure the **correct subscription is selected** – "Azure for Students":



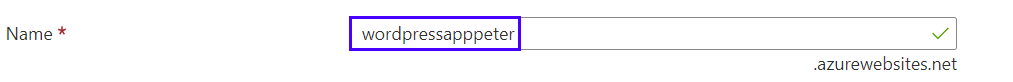
Тhen choose to [Create new resource group]. Type "WordPressResourceGroup" for the **name**:



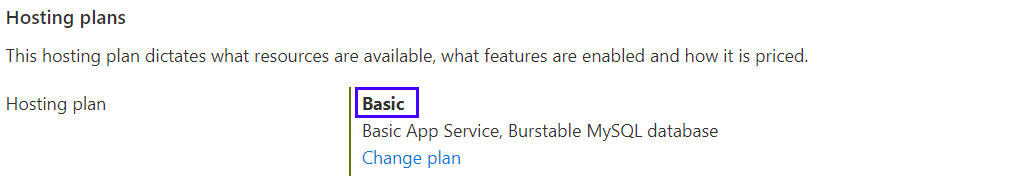
Select a **region** near you that you want to **serve your app** from:



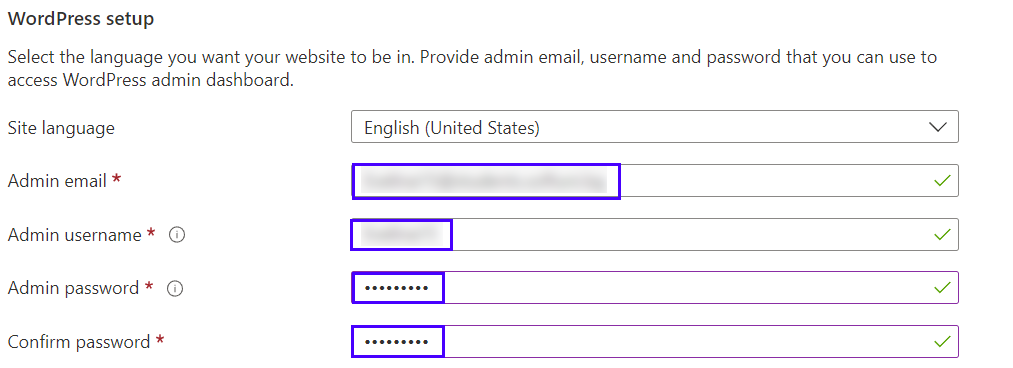
Provide a **name** for the Web App, which should be **unique**. You can name it "wordpressapp*{your name}*":



Select [Basic] for **hosting plan**:



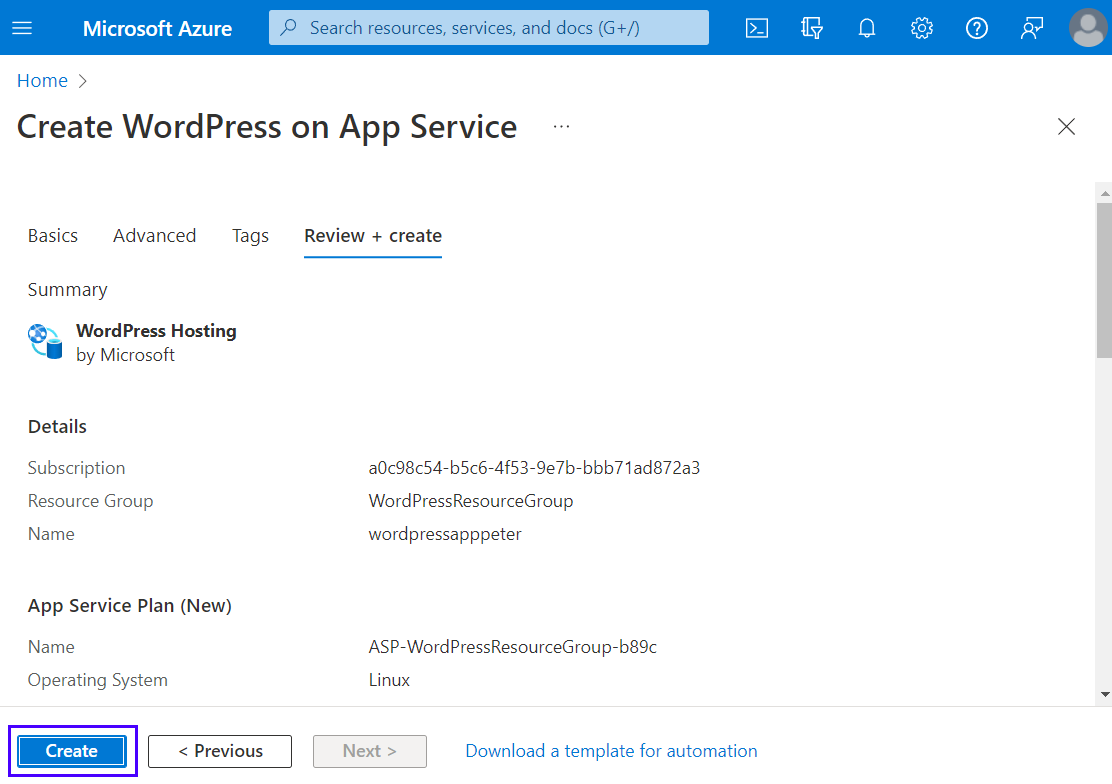
Under "WordPress Settings", type an **admin email**, **admin username**, **admin password** and **admin confirm password**. The **admin email** here is used for **WordPress administrative sign-in** only:



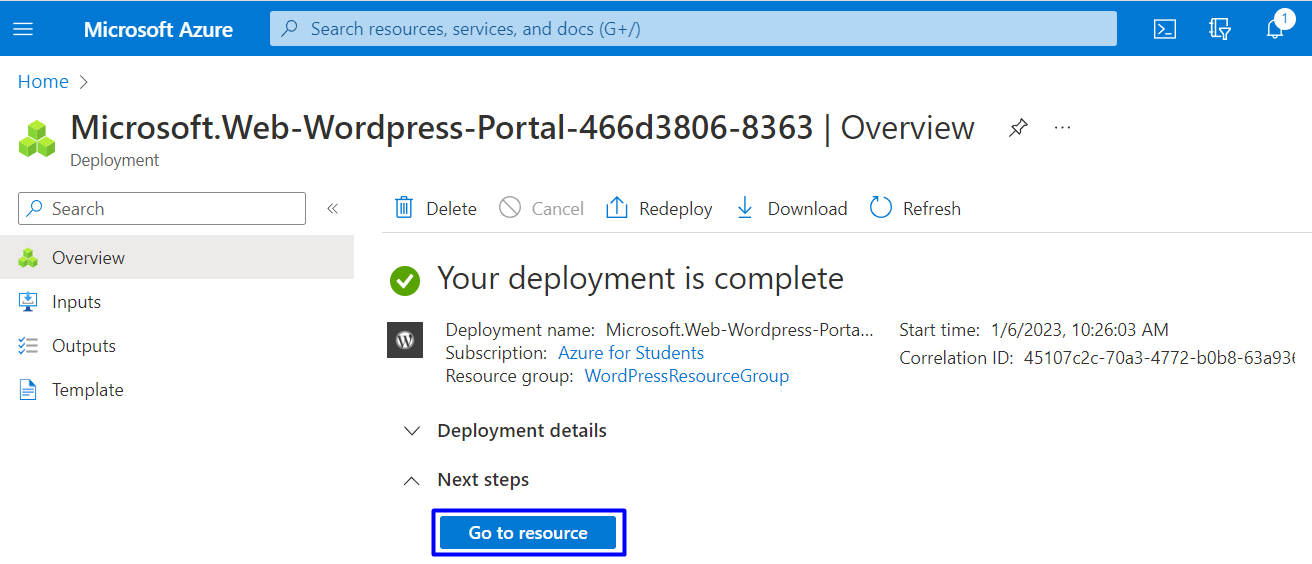
**Remember your** **credentials** for later. Then click on [Review + create]:



Click on [Create] to **create the resources**:

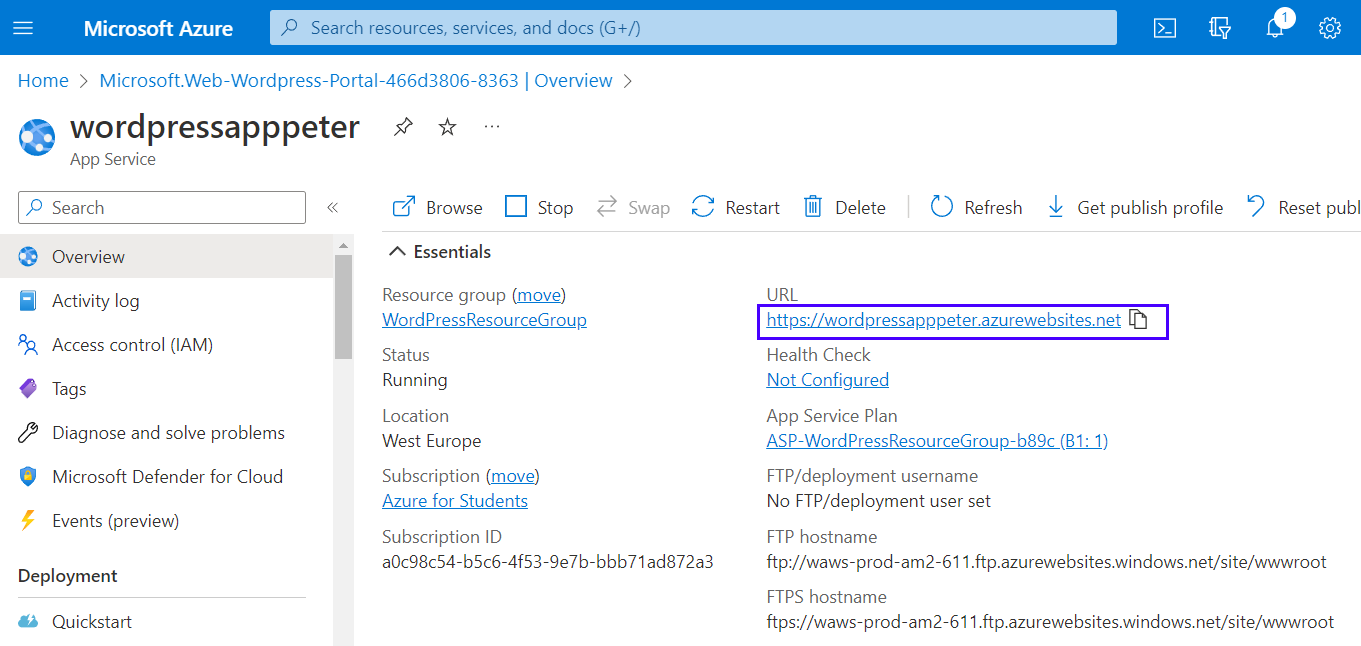


Then **wait for the deployment** to finish and click on the [Go to resource] **button**:



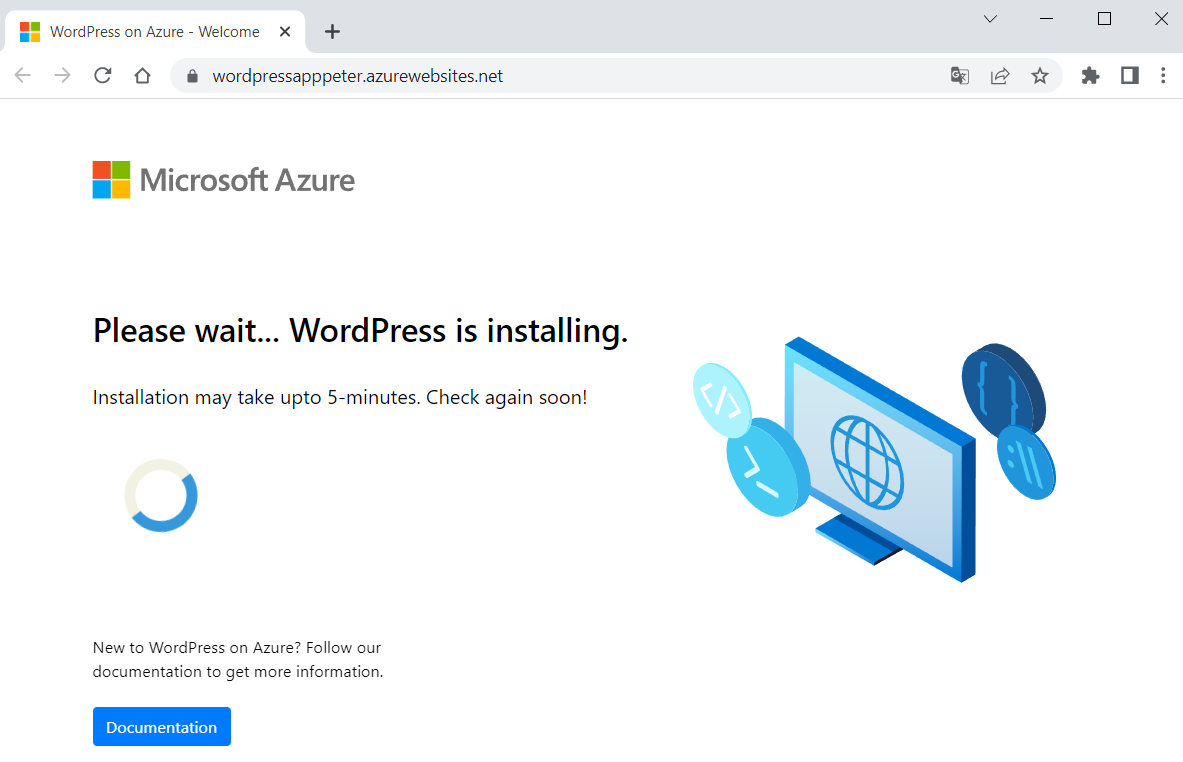
### Step 2: Browse Site

You can **click on the URL** of the site in the "App Service" **page**, in the [Overview] **tab** to access the site:

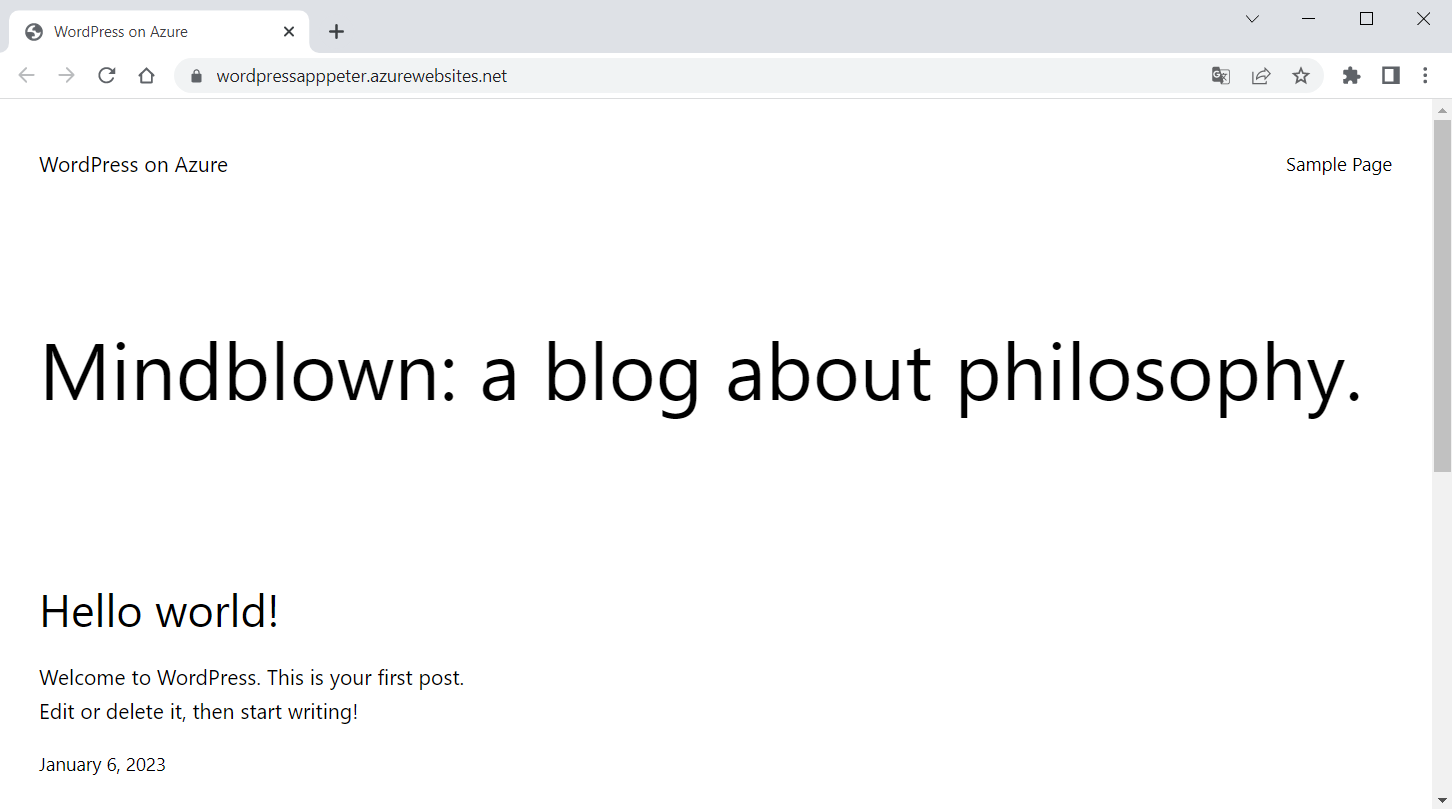


Or you can also **navigate directly** to https://<app-name>.azurewebsites.net.

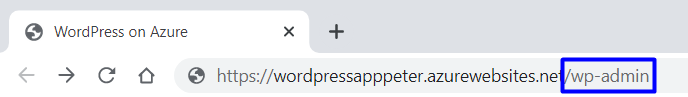
Your **site needs time to** **install**. **Wait** for it:

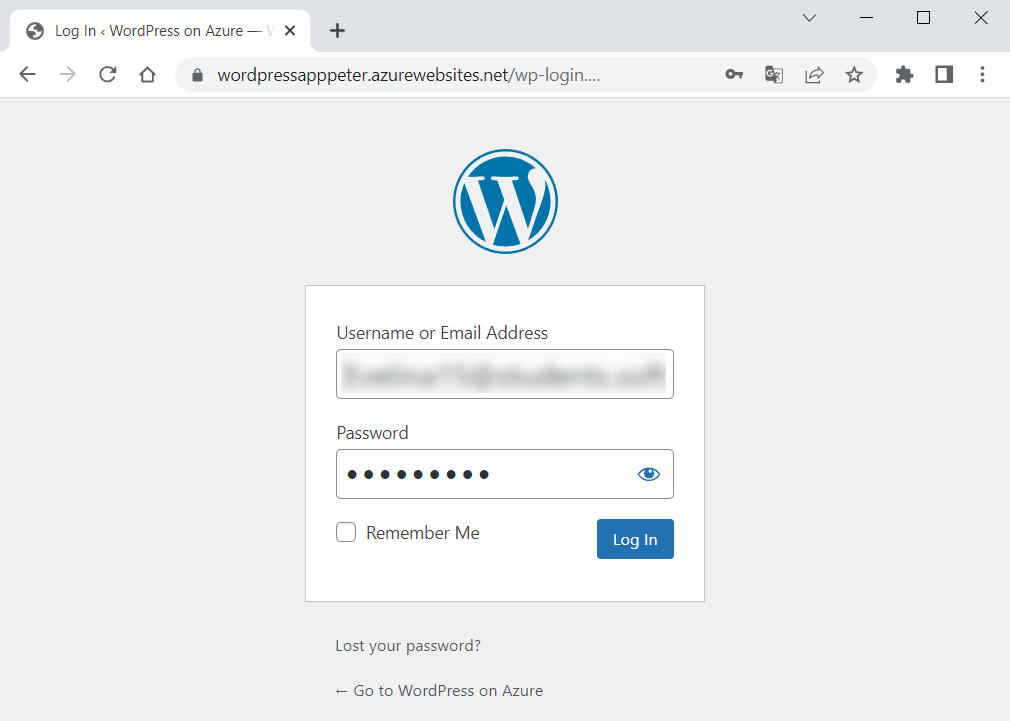


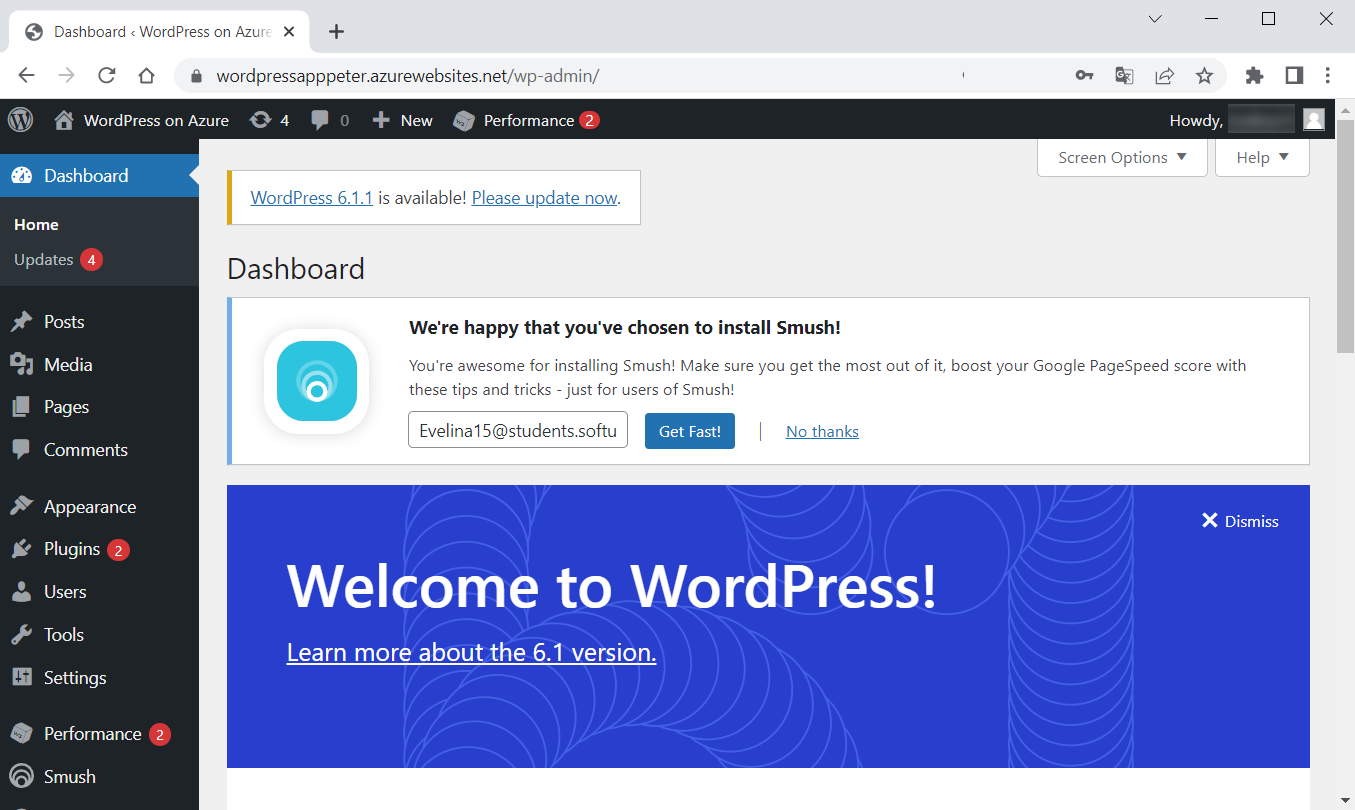
When ready, verify the **app is running properly**. If you **receive an error**, allow a few more minutes for the site to load and then **refresh the browser**.



To **access the** WordPressAdmin **page**, browse to /wp-admin and **use the credentials** you created in the "WordPress Settings" **step**:

 🡪

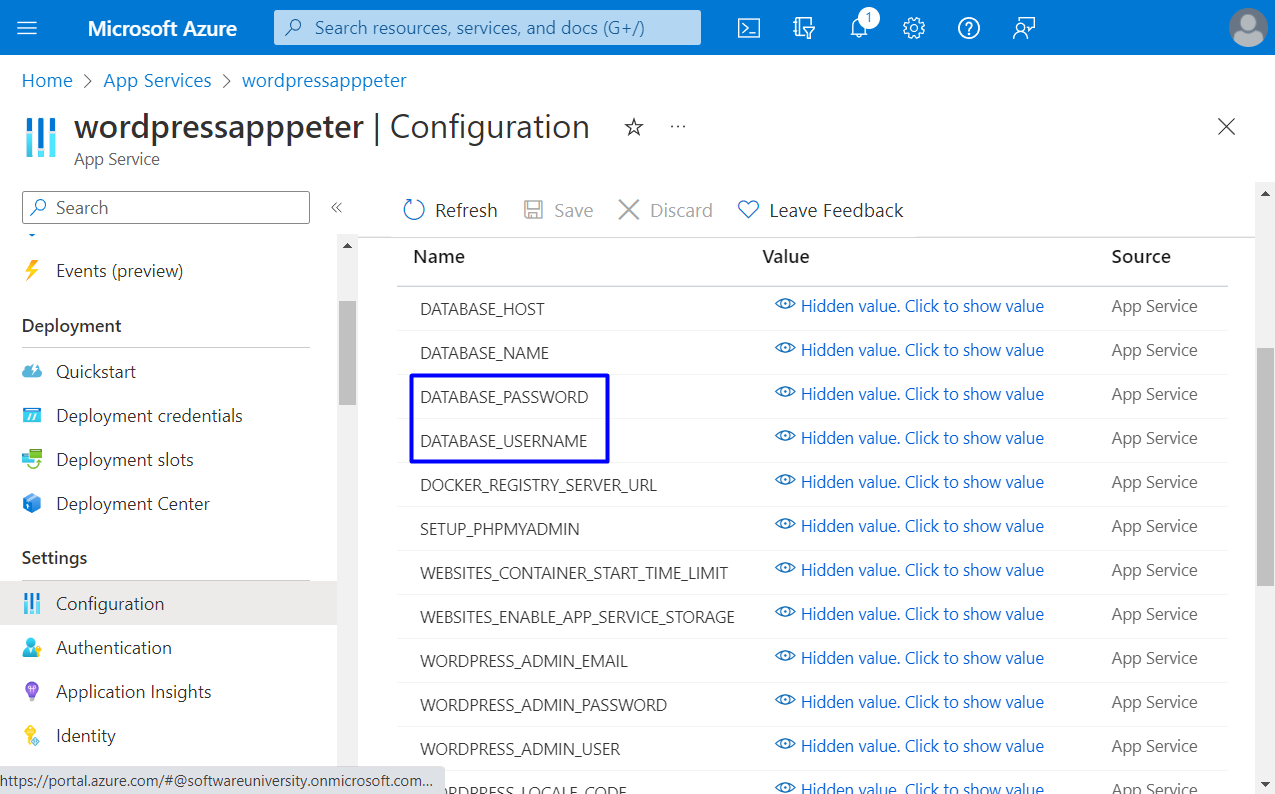
 🡪



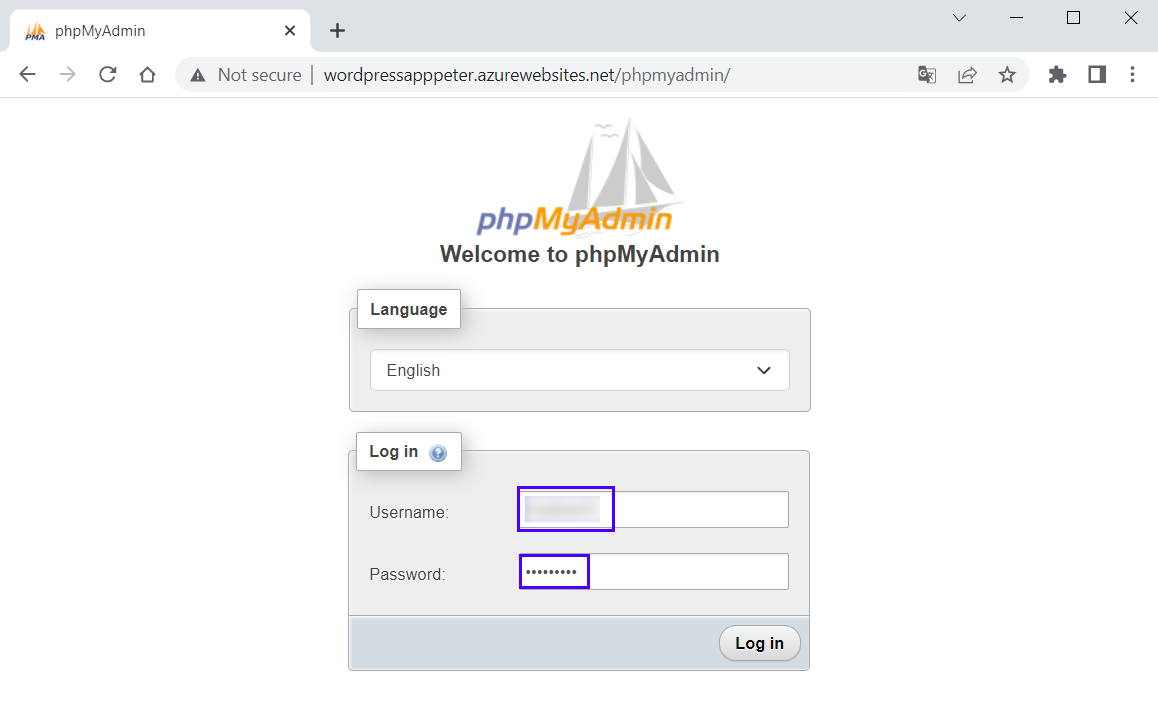
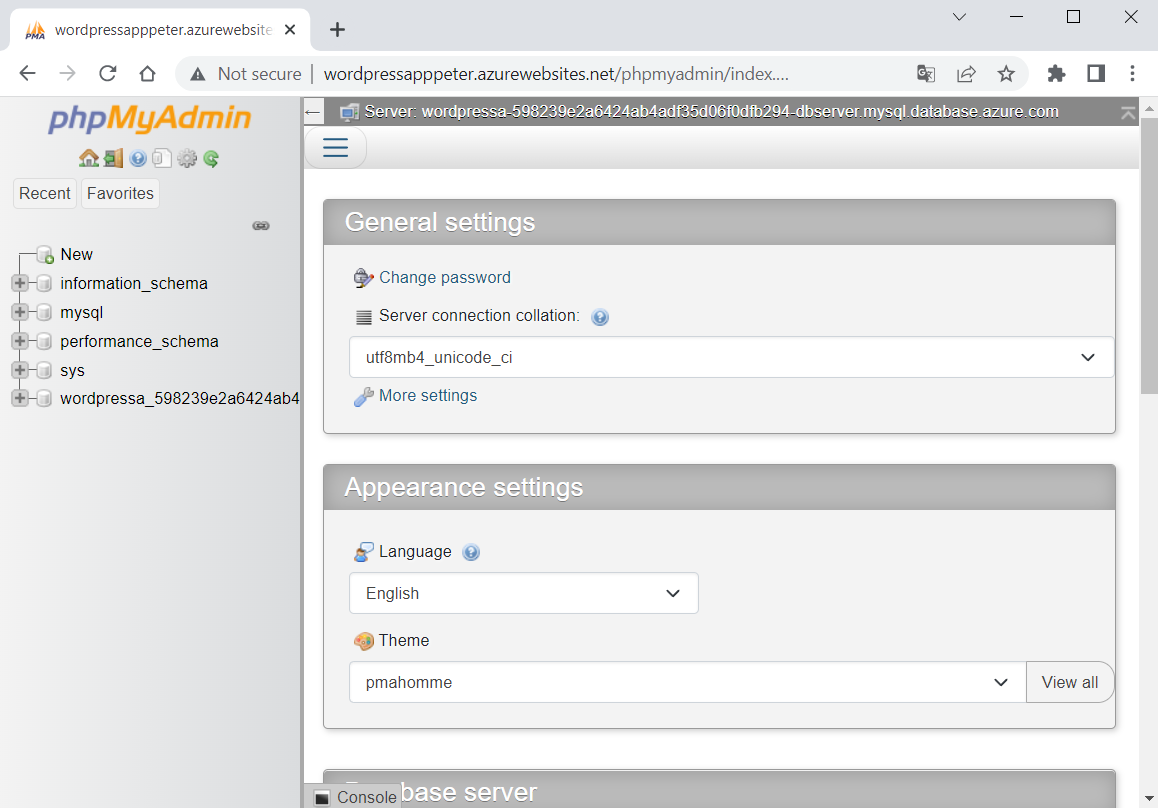
### Step 3: Access the Database

You can also **access the** MySQLFlexibleServer **database** but **not directly** because it is created behind a **private Virtual Network**.

To **access or manage the database**, we should first **get its credentials**, which are **generated automatically**. To retrieve these values after the deployment, go to "Application Settings" **section** of the "Configuration" **page** in Azure App Service and **copy the values** of DATABASE\_USERNAME and DATABASE\_PASSWORD:



**Access the database** by using phpMyAdmin that's deployed with the WordPress site: navigate to https://<sitename>.azurewebsites.net/phpmyadmin and **log in with the credentials**:

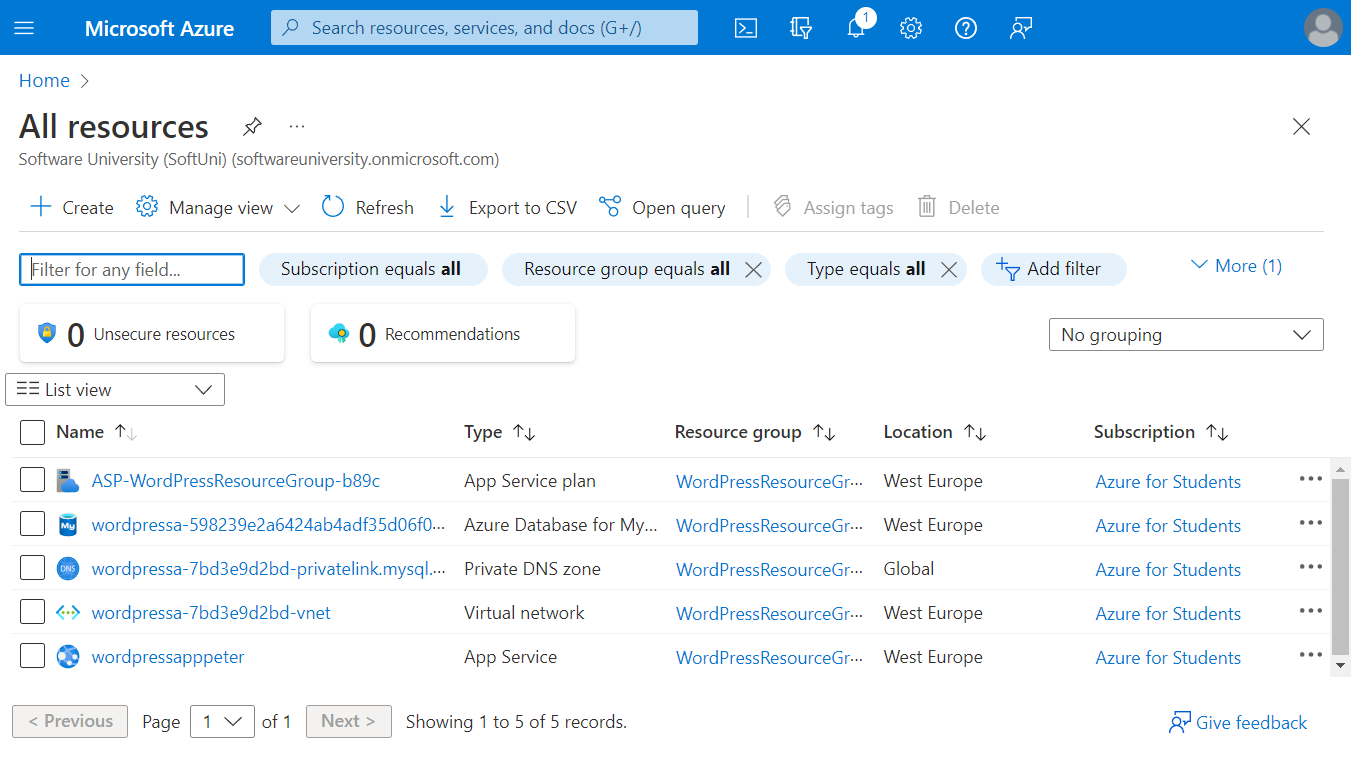
 🡪 

You can **explore the database** and its **tables**.

### Step 4: Explore App in Azure Portal

In Azure Portal you can **see and manage all app resources** that you created.

Look at the **resources** on the "All resources" **page** (you can search for it in the **search bar**):

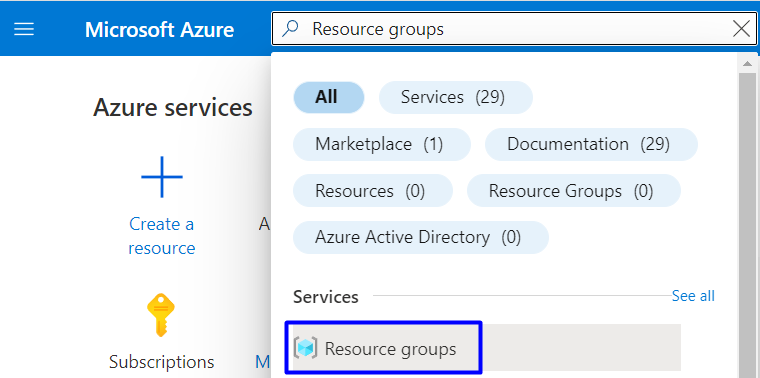


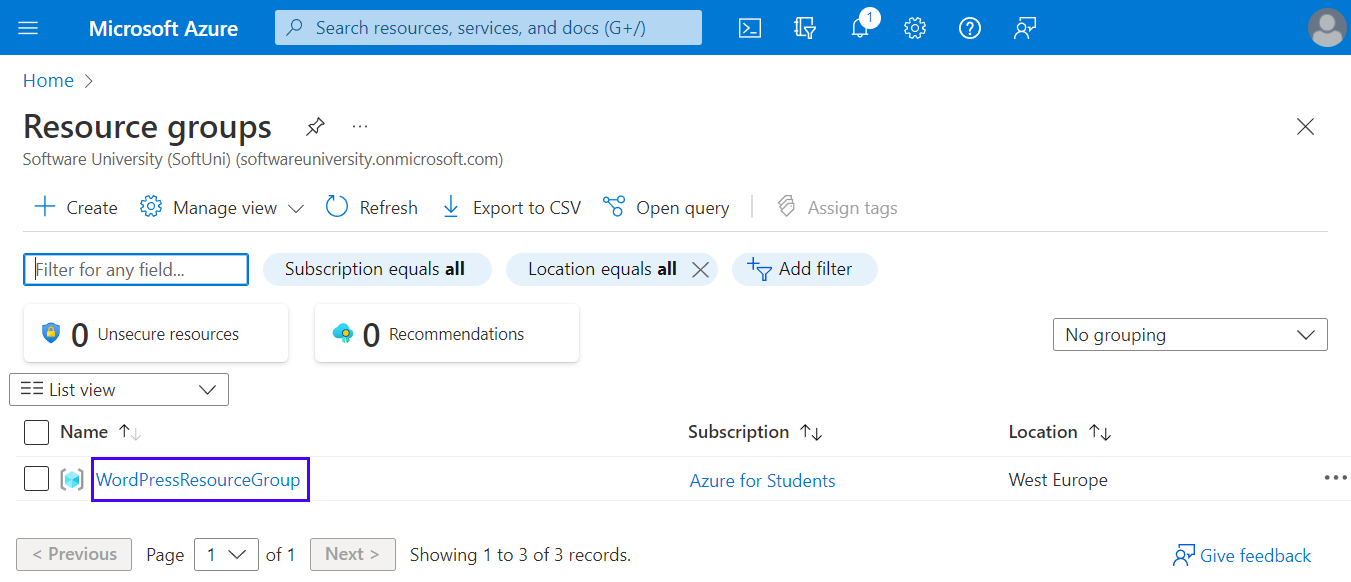
Look at each of resources and **explore** Azure Portal.

### Step 5: Delete App and Clean Up Resources

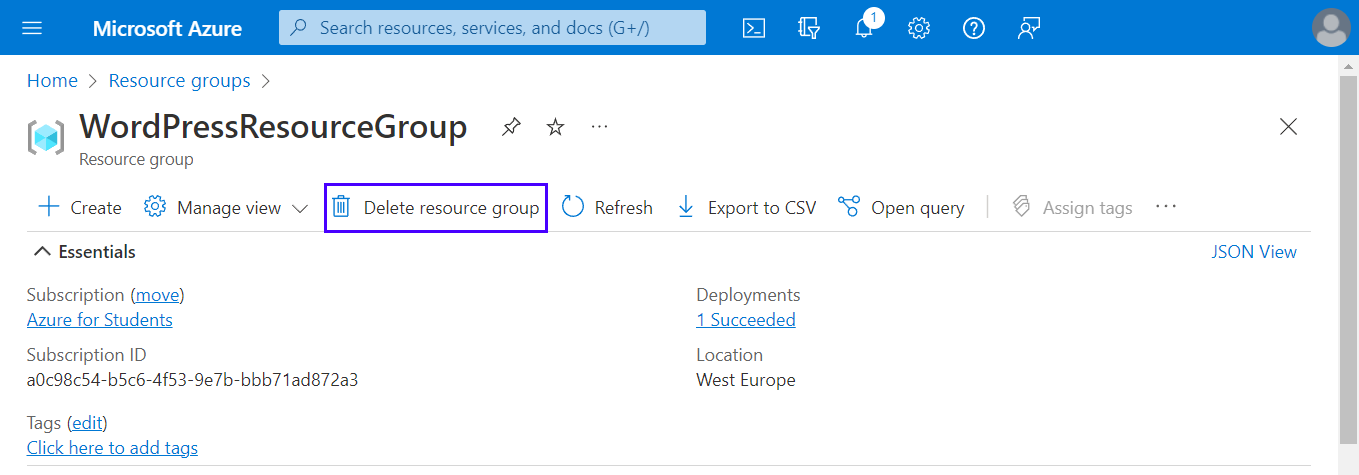
When you have successfully **uploaded your app to** Azure and **don't need it anymore**, it is time to **remove it**.

You can **delete all of the app resources** from your Azure **subscription** by **deleting the resource group**. Find **all resource groups** by going to Azure Portal and typing "Resource groups" in the **search bar**:

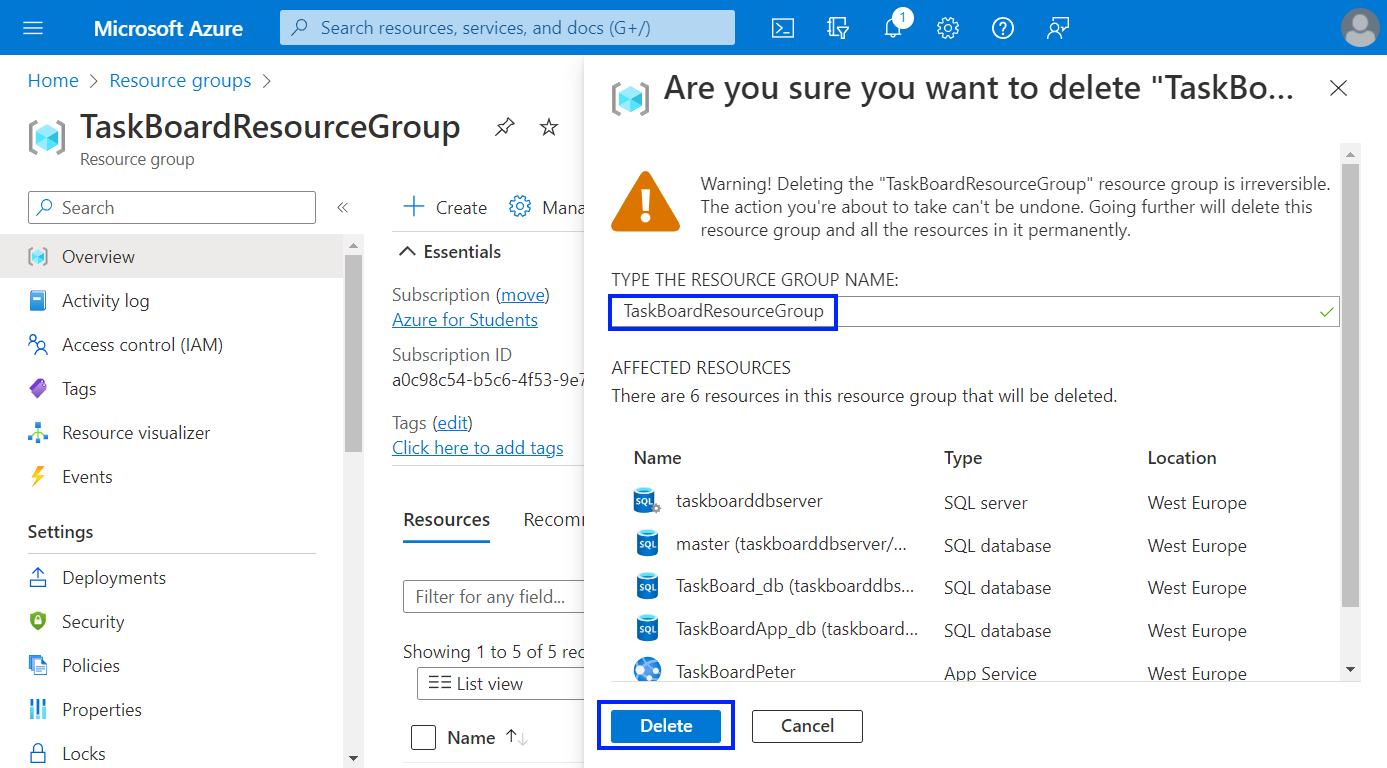
 🡪



You have your "WordPressResourceGroup" **group**. Delete it by selecting [Delete resource group]:



Next, type the **resource group name** and click [Delete]:

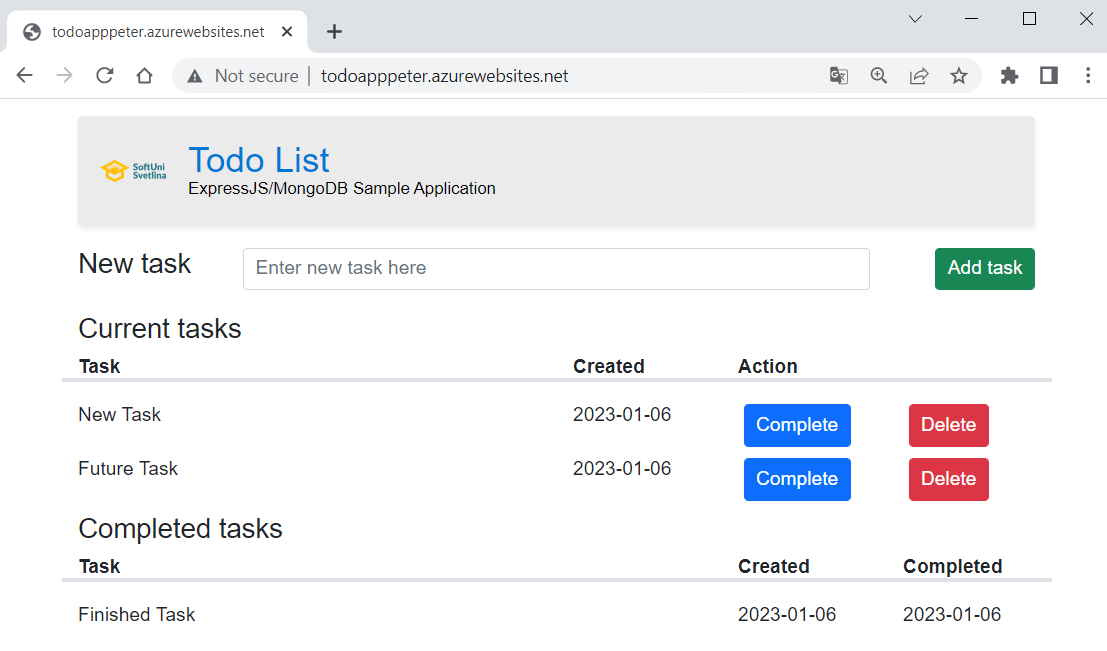


The **resource group** will be **deleted** after a while. All **resources of your app** will be **deleted** with it.

Now you have a **fully-functional** WordPress **site** in Azure and you know how to **manage it and its resources**.

## Deploy a Node.js + MongoDB Web App to Azure

You are provided with a Node.js **app** with a MongoDB **database** called "TODO App" in the **resources**. It looks like this when **run**:



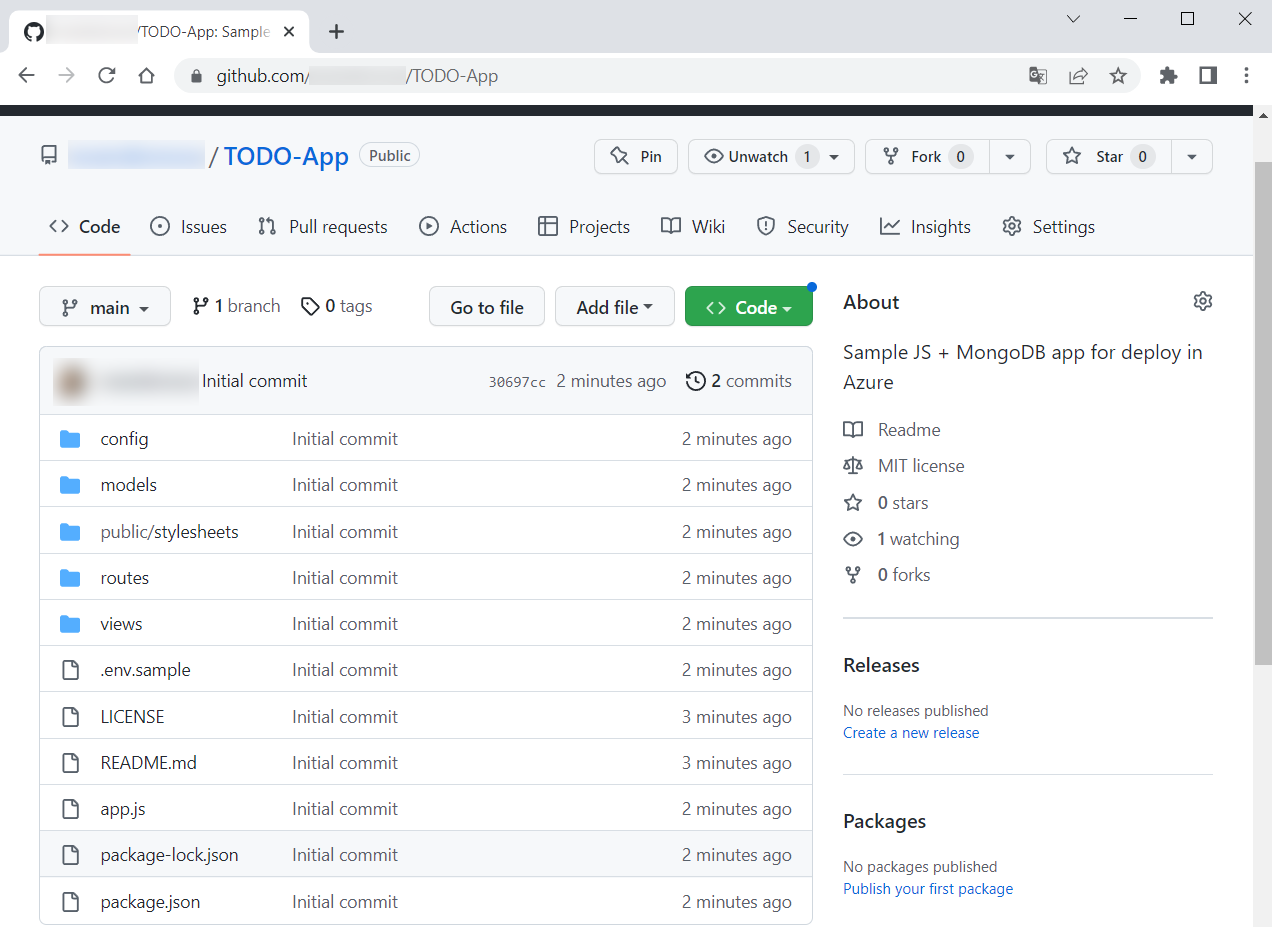
You should **upload the app to** GitHub, as a start, and then **deploy it to** Azure, using Azure App Service and Azure Cosmos DB for MongoDB.

The **process** is pretty similar to **deploying an ASP.NET app with a SQL database** – the only difference is the **settings of the app**.

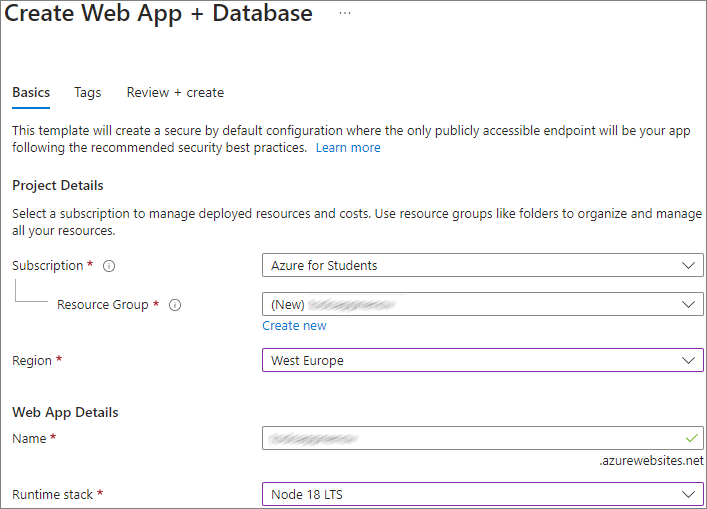
### Hints

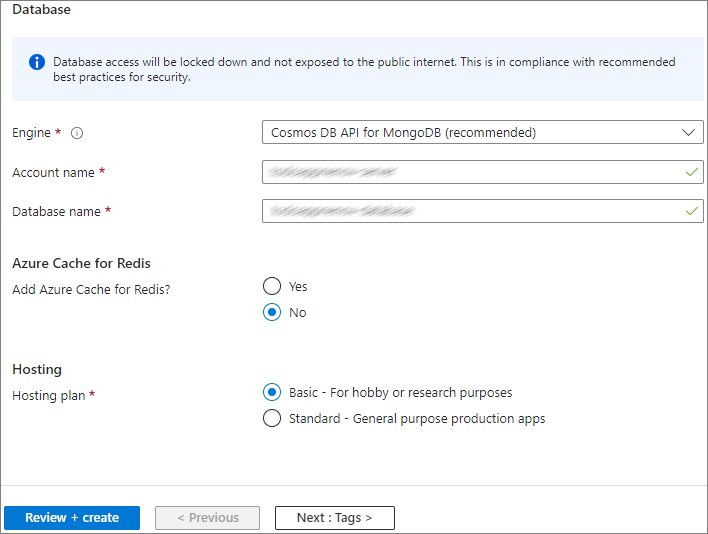
You should fulfill the below **steps** to **run the app in** Azure:

1. Upload the "TODO App" **project** **code** from the **resources** to a GitHub **repository**.

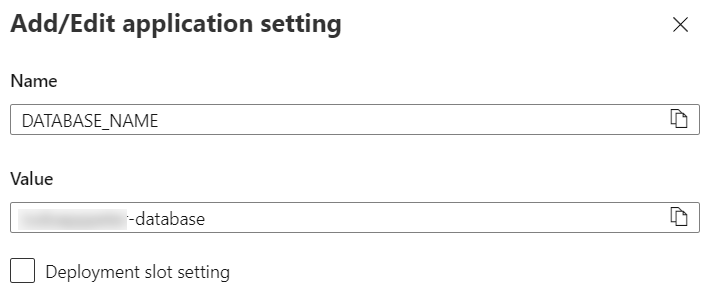


1. In Azure Portal, go to the "Create Web App + Database" **page** and **fill in the fields**. Create a **new resource group**. Give a **suitable name** to the Azure app. Choose [Node 18 LTS] as **runtime stack**. Don't change the **default selected database engine** (Cosmos DB **API** for MongoDB) and **copy the generated database** **name** because you will need it later.

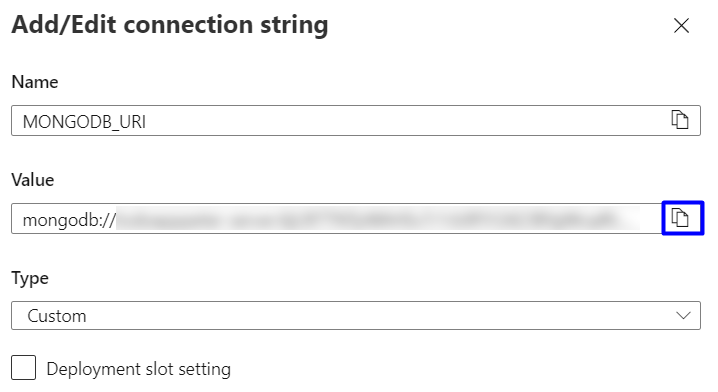




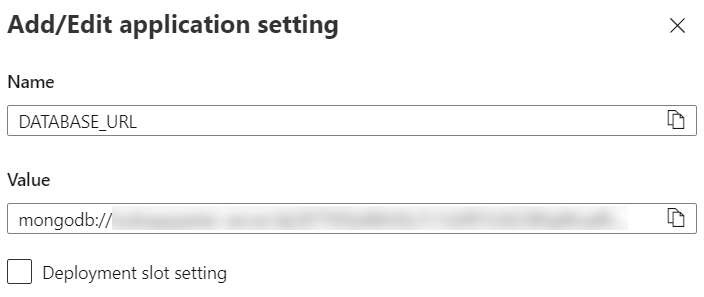
1. Go to the "Configuration" **page** of the App Service **app** and create a [New application setting] with:
   * **Name**: "DATABASE\_NAME"
   * **Value**: the automatically **generated database name** you copied earlier (i.e. <app-name>-database)



1. Next, select the "MONGODB\_URI" **connection string** on the same page and **copy its value**

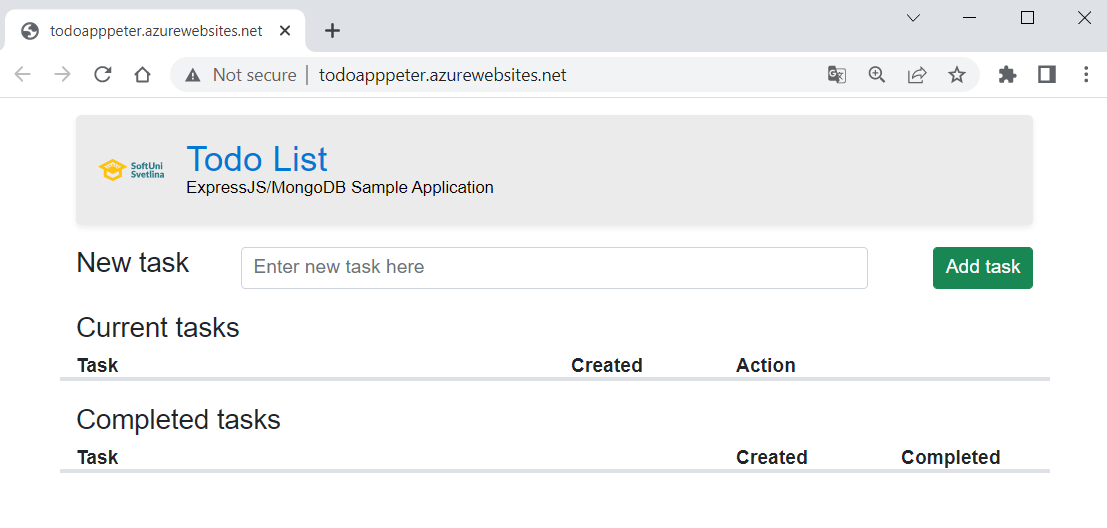


1. Create one more **new application setting** with:
   * **Name**: "DATABASE\_URL"
   * **Value**: the "**MONGODB\_URI**" **connection string** you copied earlier (i.e., mongodb://...)



Don't forget to **save the settings**.

1. Your app is configured to work with the **Azure database**, so you should just go to the "Deployment Center" **page** and **deploy the app** from the GitHub **repository**.
2. A GitHub **workflow** should be created and it should **show a status of "**Complete". It takes about 15-30 minutes.
3. You should be able to **access the app** on https://<app-name>.azurewebsites.net and **work with it**.

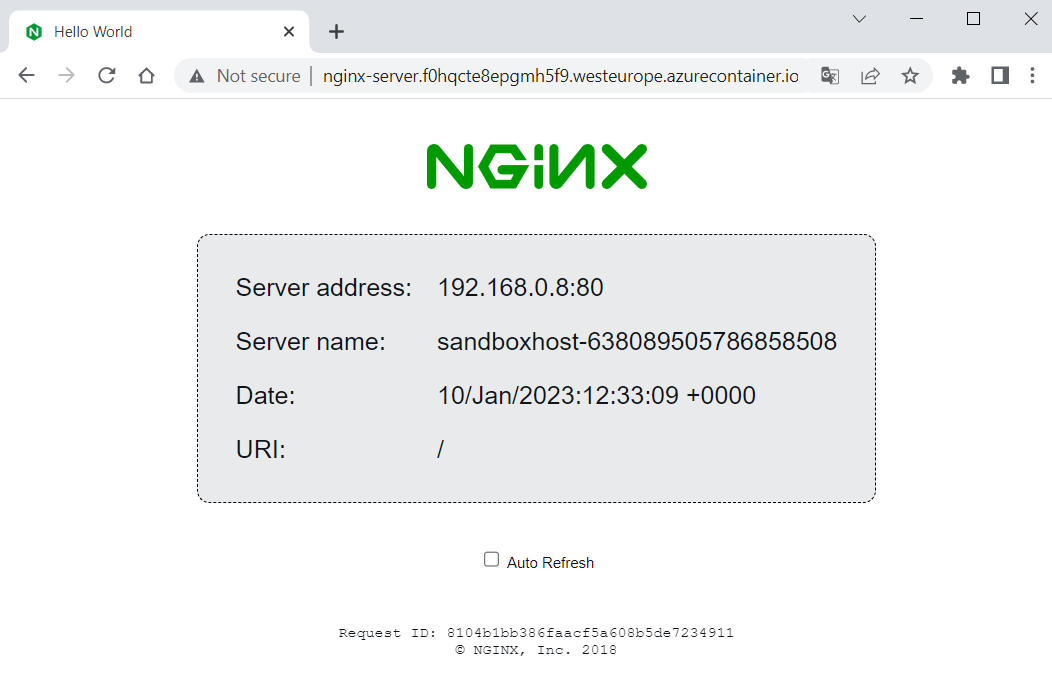


**NOTE:** Don't forget to delete the resources in your Azure account.

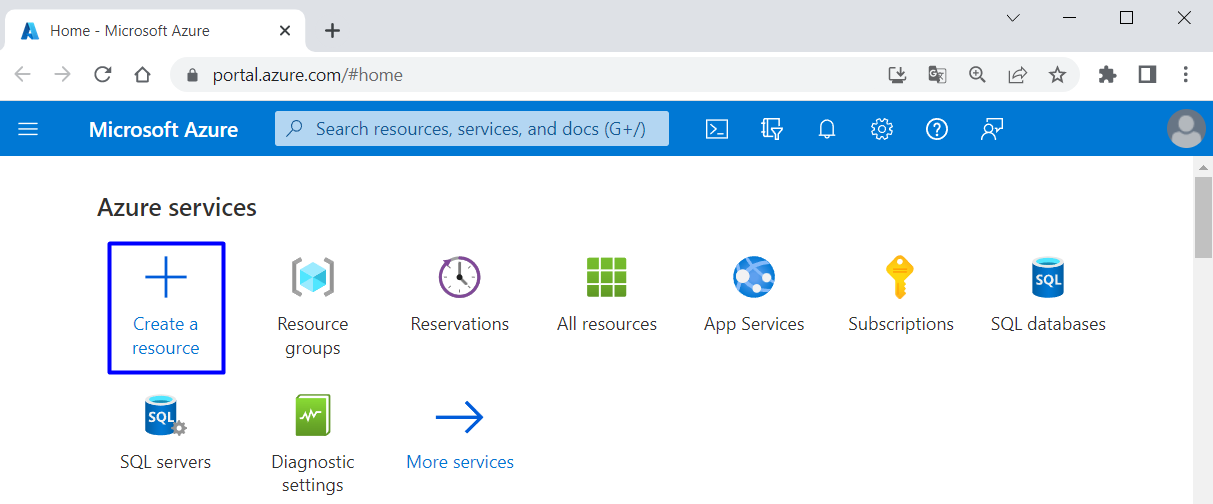
## Deploy NGINX Server Container to Azure using Azure Portal

In this task, we will **deploy a container instance** to Azure Container Instances using the Azure Portal. Azure Container Instances allows us to **run serverless and isolated Docker containers** in **Azure** and make its application available with a **fully qualified domain name** (FQDN).

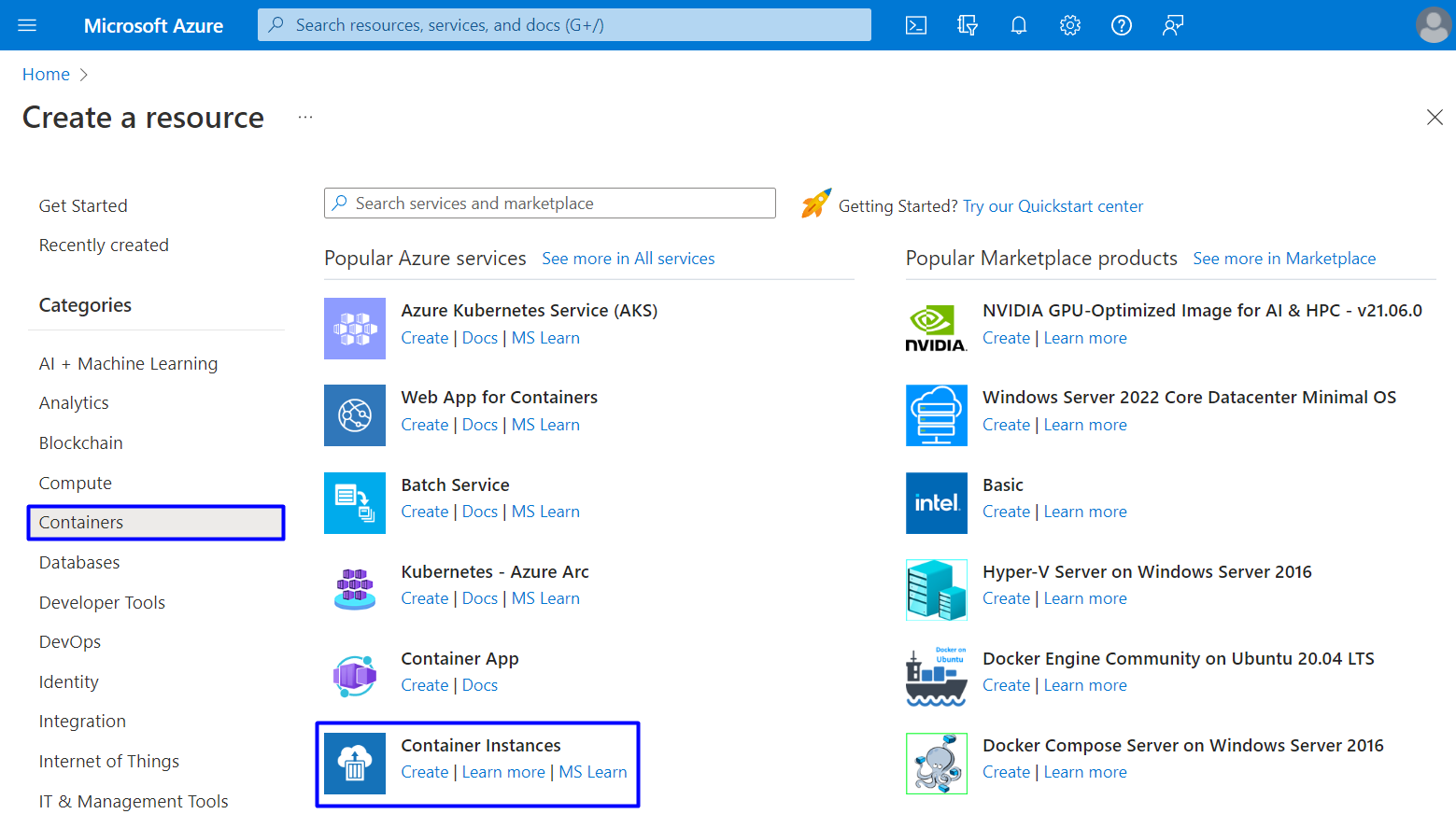
We will use the NGINX **server** **image** from DockerHub: <https://hub.docker.com/r/nginxdemos/hello>. You are already familiar with the **server**. When **deployed to** Azure, it will look like this:



Let's see how to do this. As a first step, open Azure Portal (<https://portal.azure.com>) and **click on** [Create a resource]:

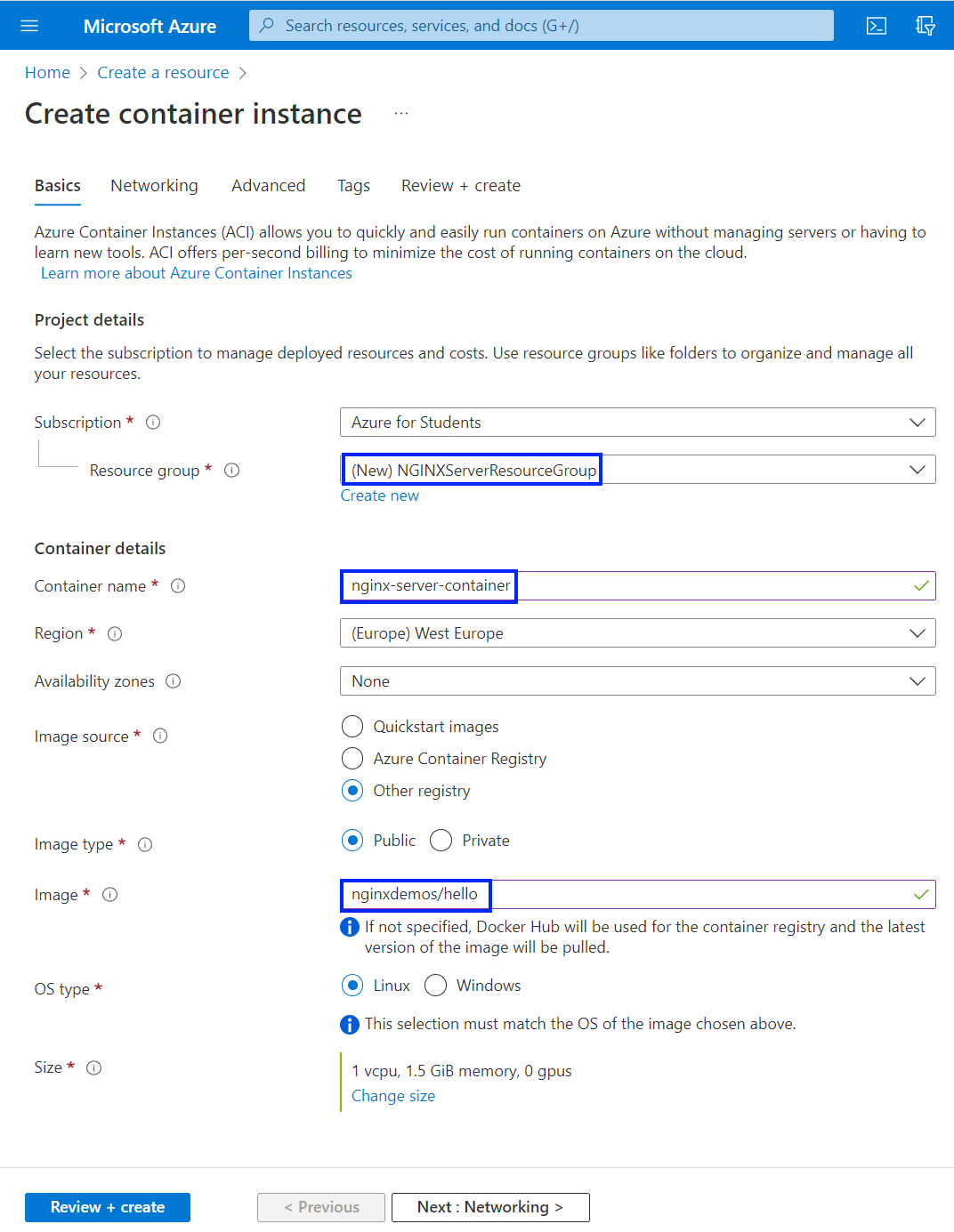


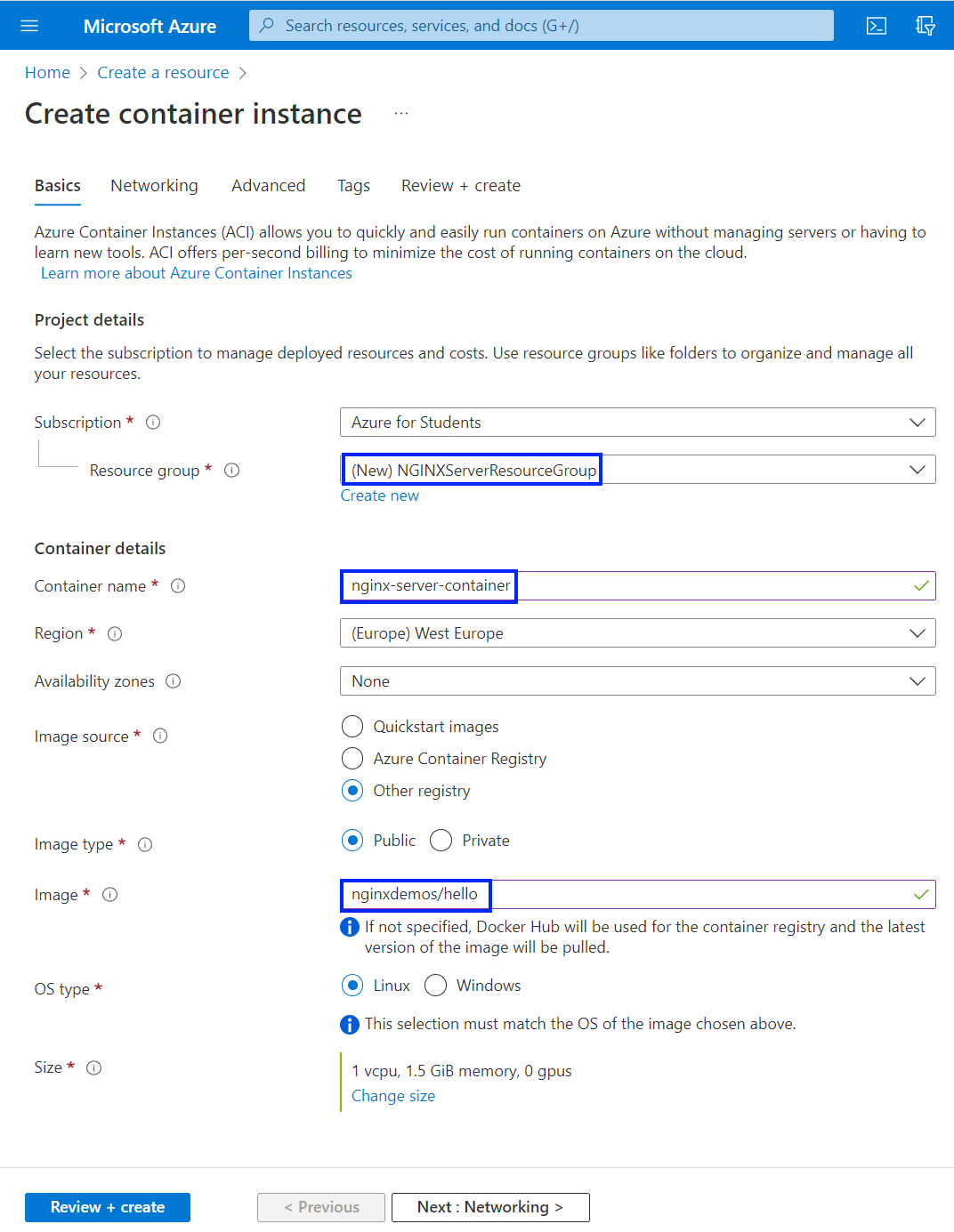
Next, choose [Containers] 🡪 [Container Instances]:



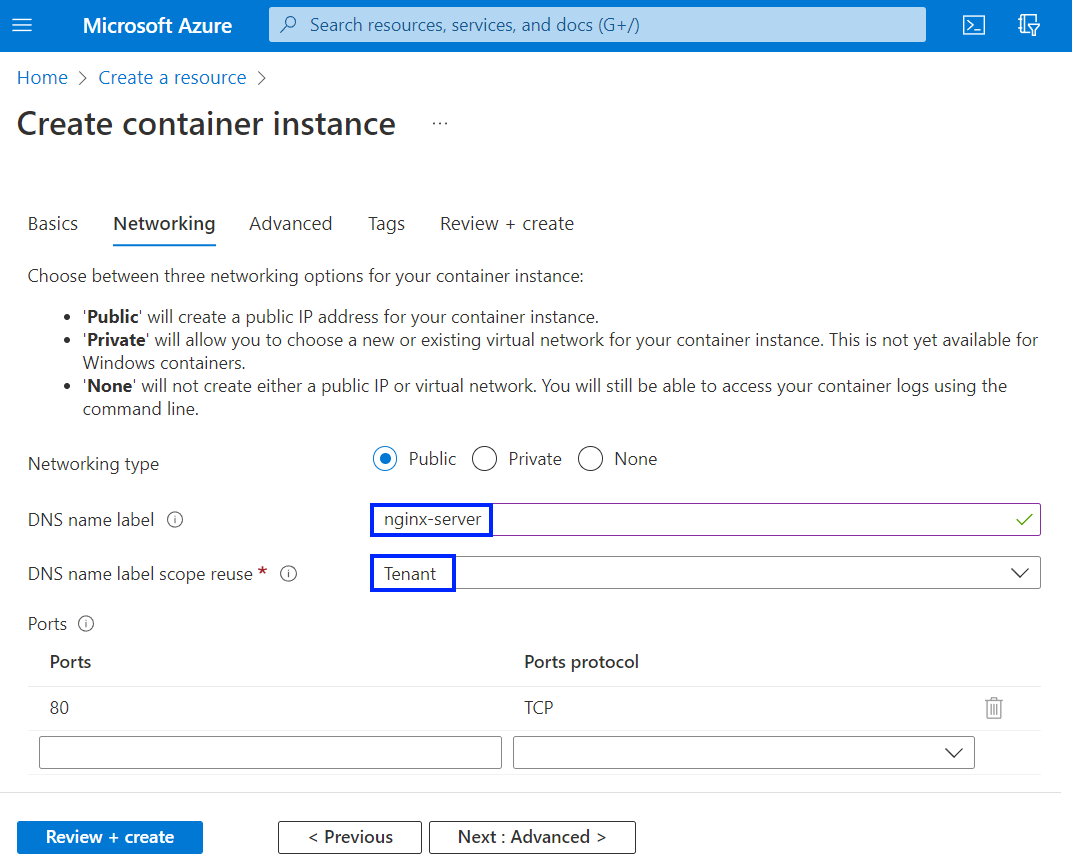
On the "Basic" **tab** of the "Create container instance" **page** you should **fill in the fields** as follows:

* Use your "Azure for Students" **subscription**
* Create a **new resource group** with a suitable name
* Create a **name for your container**
* Choose a **region** near you
* Leave the "Availability zones" to [None]
* For the **image source**, choose [Other registry] (to use an **image from** DockerHub, which is the **default registry** for this **option**) and **fill in the image name**
* Leave the "OS type" to [Linux] and "Size" field with the **given options**





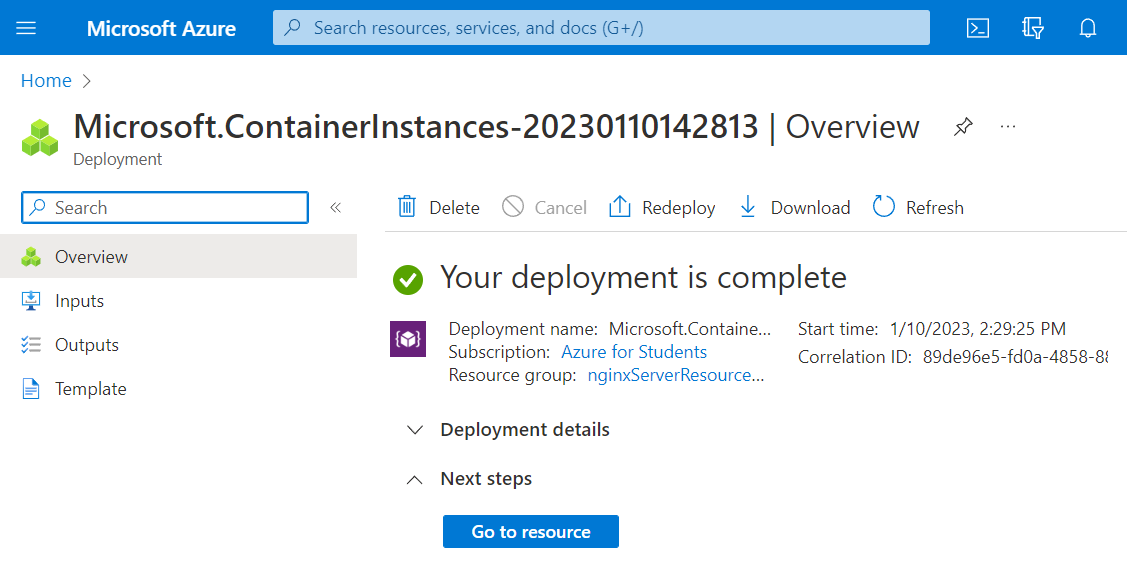
Click on [Next] to go to the "Networking" **tab**. There you should **fill in a DNS name** **for your container** and **choose** [Tenant] for **DNS scope reuse**:



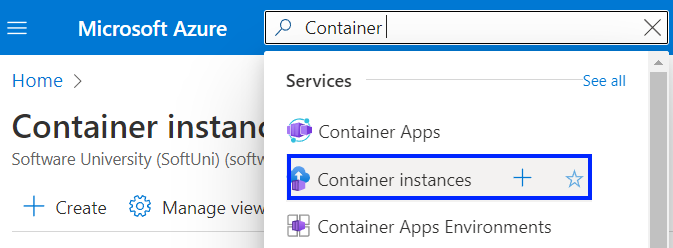
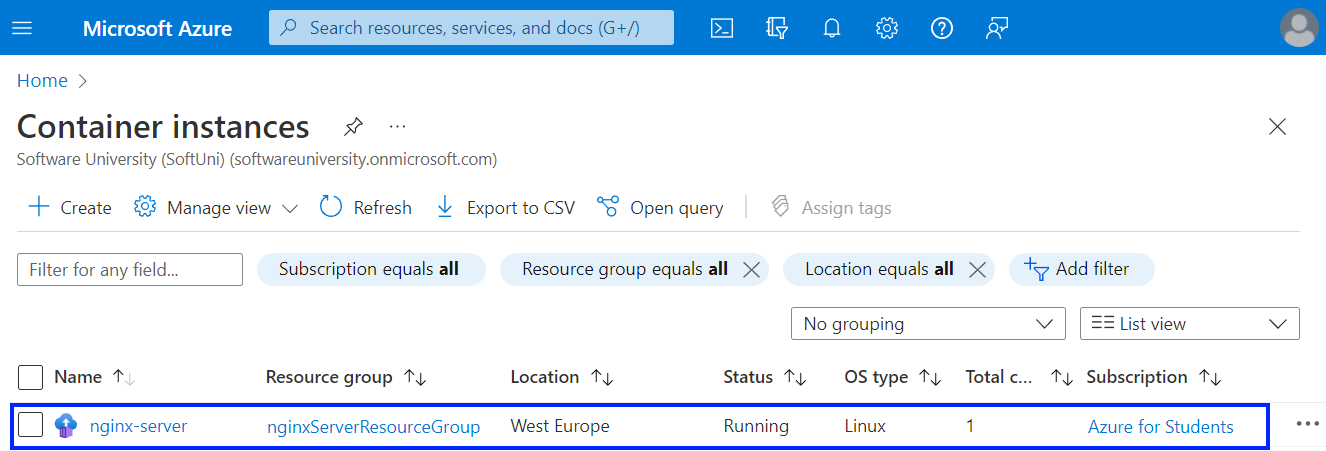
The **name must be unique** within the Azure region where you create the container instance. Your container will be **publicly reachable** at <dns-name-label>.<region>.azurecontainer.io. An **auto-generated hash** is added as a **DNS name label** to your container instance's **fully qualified domain name** (FQDN).

If you receive a "DNS name label not available" **error message**, try a **different DNS name label**.

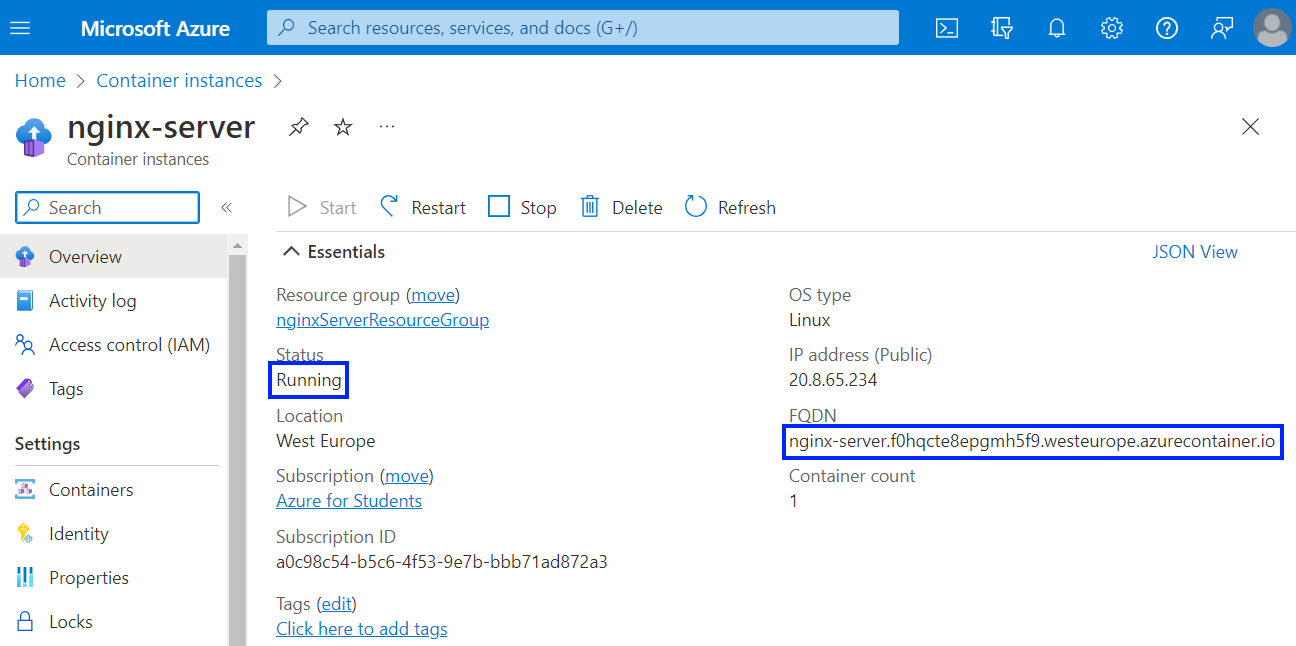
**Leave all other settings** as their defaults and select [Review + create], then [Create]. Wait for the **deployment** **to complete**:



Then go to nginx-server **container instance** from the "Container instances" **page**:

 🡪 

Look at the FQDN and the **status** **of the container** on the "Overview" **page**:

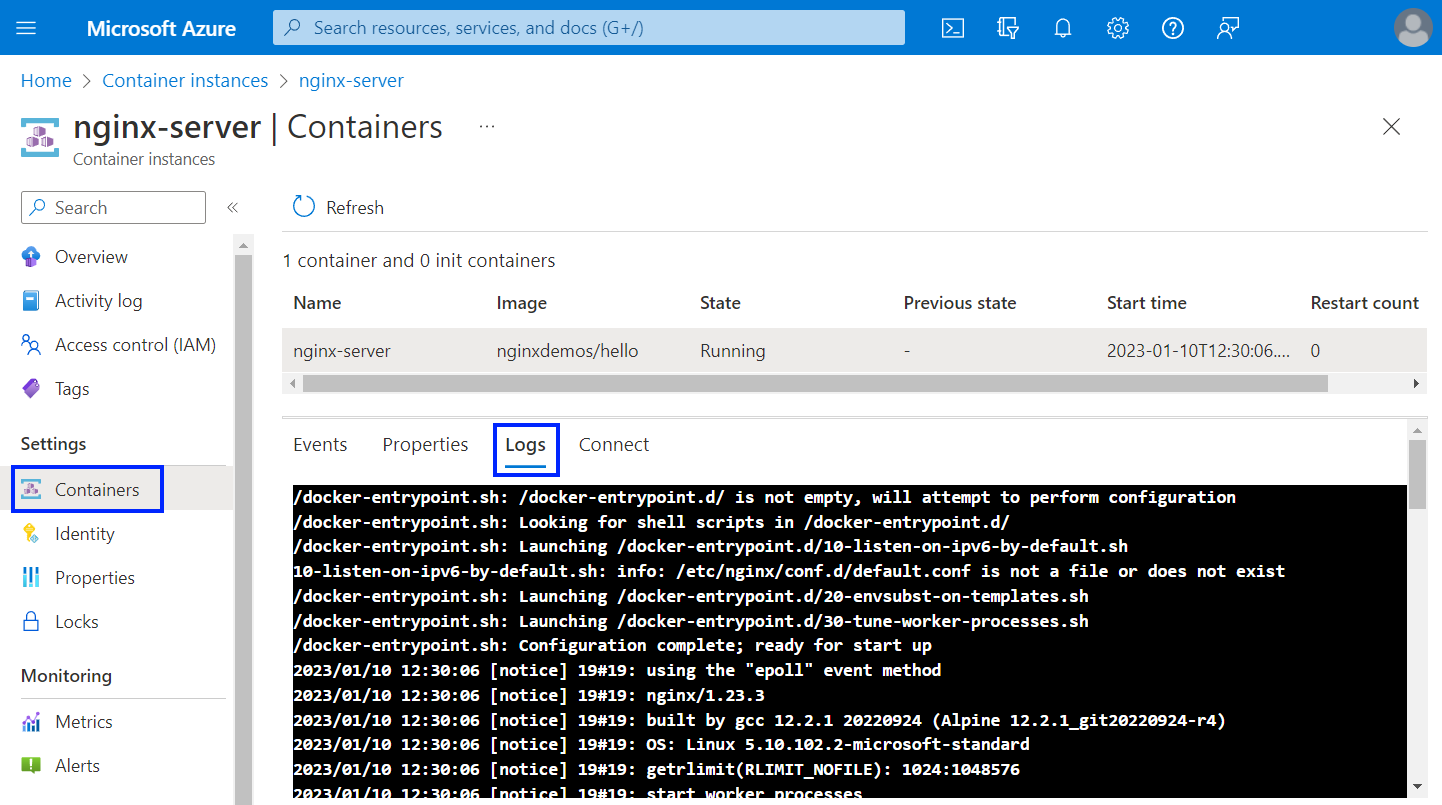


When the **status** **is** "Running", navigate to the **container's FQDN in your browser** and you should see your NGINX **server**:

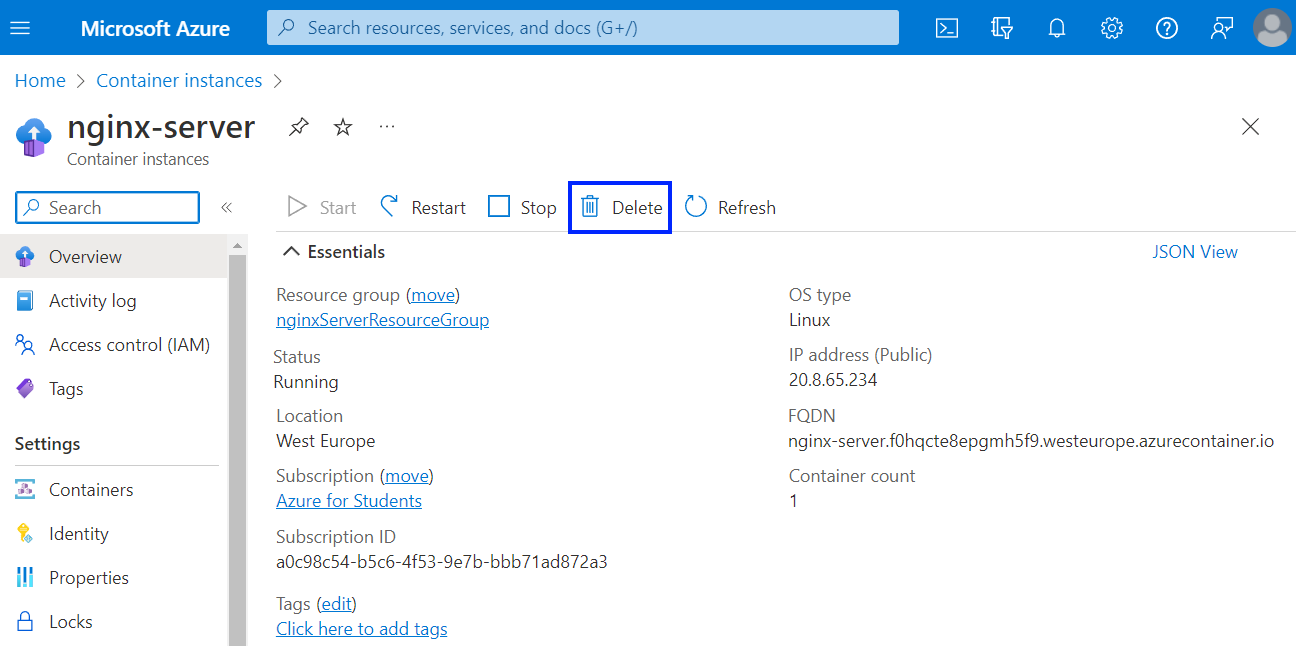


**NOTE**: if you **receive an error**, **wait for the container** a little bit more. After **up to 5 minutes**, it should be **loading** **successfully**.

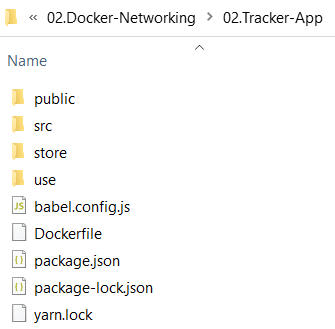
If you have **any other problems**, you can look at the **container logs** in [Containers] 🡪 [Logs]:



You can **delete the container** by **deleting its resource group** in the familiar to you way. You can also **delete only the container** by going to [Overview] and clicking on [Delete]:



## Deploy the "Tracker" App to Azure

Your task now is to **run** **a** **simple JavaScript front-end app based on** Vue.jsfor keeping track of daily duties in a Docker **container**. It does not need **anything but an image** to run. It does not use a database or any other storage.

You're provided with **its files** it in the **resources**, together with a Dockerfile which runs the **app on** NGINX **server** (on the right):

You should **deploy the app** to Azure Container Instances.

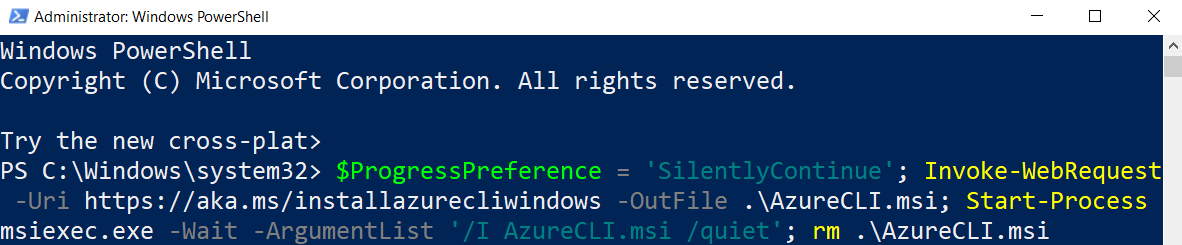
### Install Azure CLI on Your Local Computer

The Azure Command-Line Interface (**CLI**) is a cross-platform **command-line tool**, which can be used to **connect to Azure** and **execute administrative commands** on **Azure resources**. It can be run from **inside a Docker container**.

Because the Azure Cloud Shell does not include the Docker **daemon**, you must **install both** the Azure CLI and Docker Engine on your **local computer** for this task. We will **use it**, as well, but later when we are done with Docker.

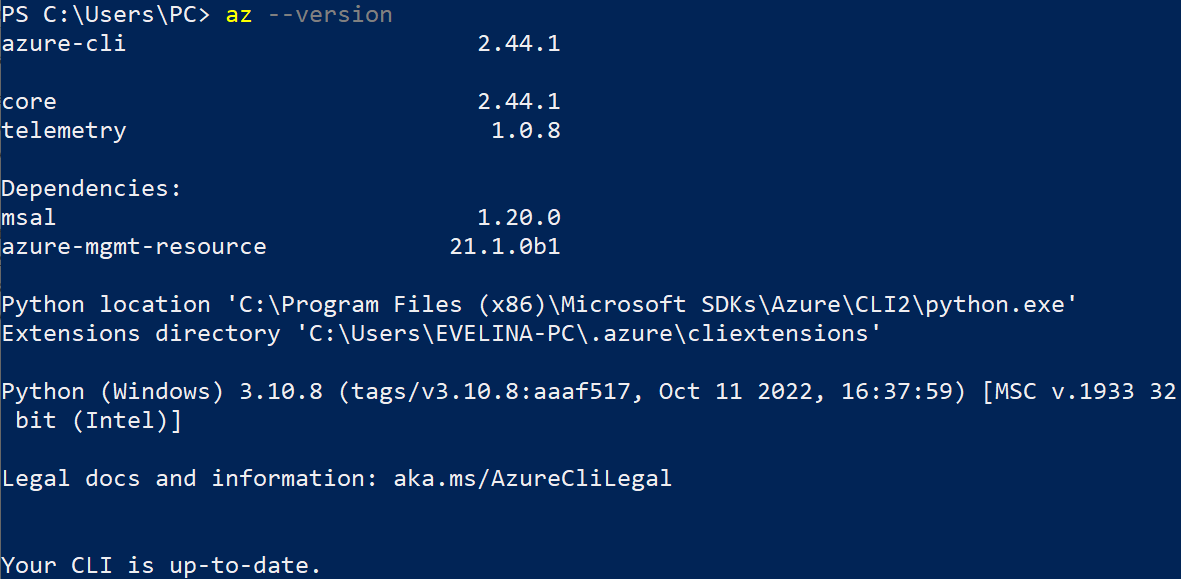
You already have Docker, so let's **install** AzureCLI. This can be done **through** PowerShell with the following **command**:

|  |
| --- |
| $ProgressPreference = 'SilentlyContinue'; Invoke-WebRequest -Uri https://aka.ms/installazurecliwindows -OutFile .\AzureCLI.msi; Start-Process msiexec.exe -Wait -ArgumentList '/I AzureCLI.msi /quiet'; rm .\AzureCLI.msi |



**NOTE:** PowerShell must be **run as administrator** for the **installation**. After that, run a **new PowerShell instance** in the standard way and **work with it**.

You can check your Azure CLI **version** with:

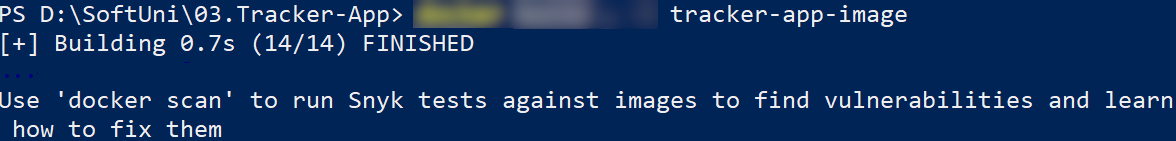


Now you can go on with **creating a container image**.

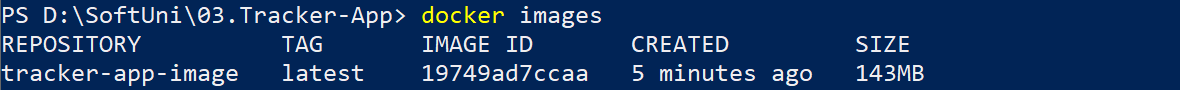
### Step 1: Create a Container Image for Deployment

In this step, you will **package the** "Tracker App" **into a container image** that can be **run** **using** AzureContainerInstances.

You are provided with a Dockerfile, so you should just **create the image**, for example "tracker-app-image". You should know how to do it:



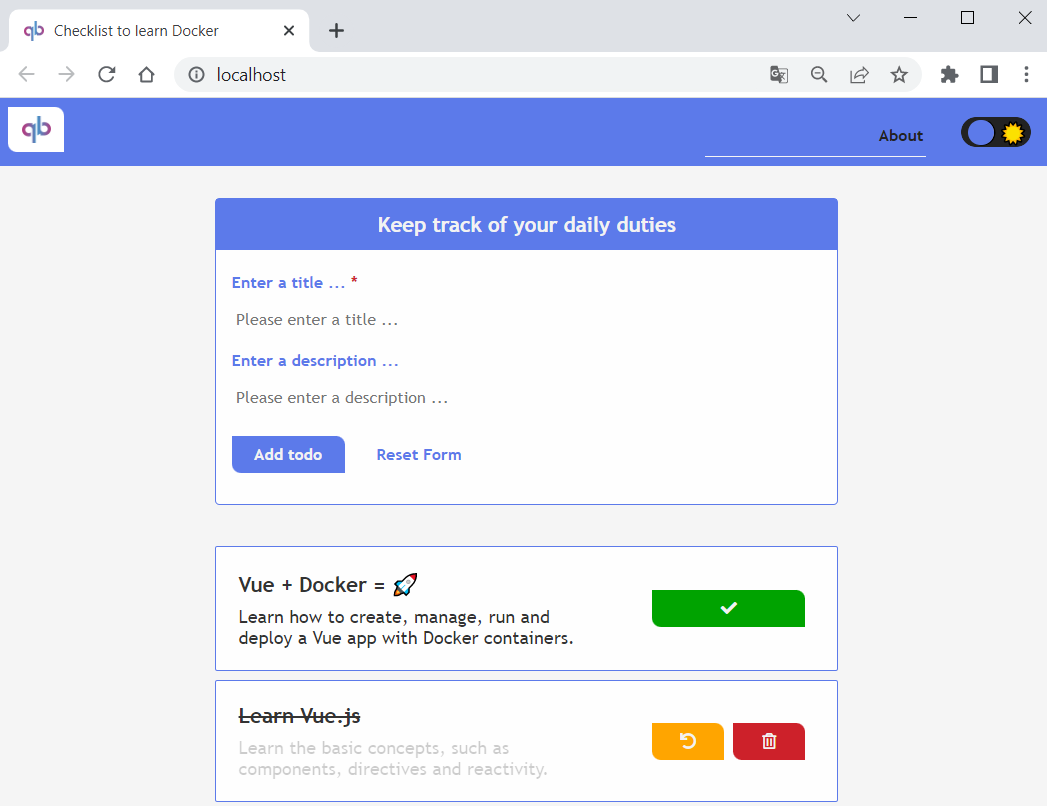
The **newly created image** should appear:



You can now use this **image to run a container locally** to see if it works:



↓



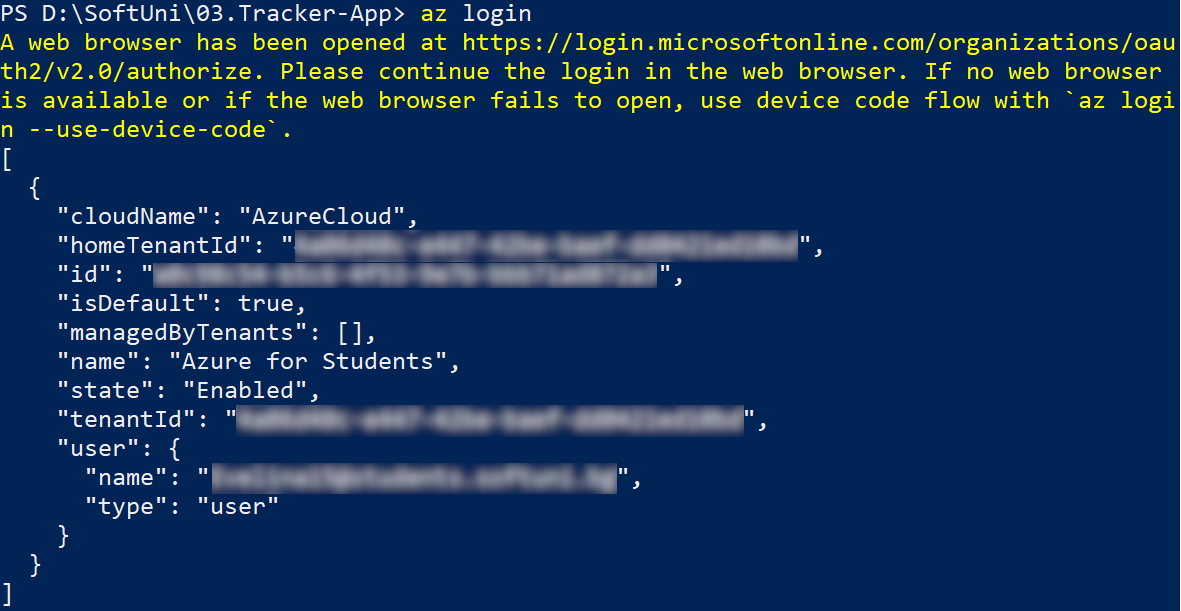
We have verified that the **container image runs locally**, so let's see how to **push it to** AzureContainerRegistry.

### Step 2: Create an Azure Container Registry and Push a Container Image

An Azure Container Registry is a **managed registry service**. It stores and manages **private container images** and other artifacts, similar to the way Docker Hub **stores public Docker container images**.

Now we should **create an** Azure **container registry** and **push our image** to it.

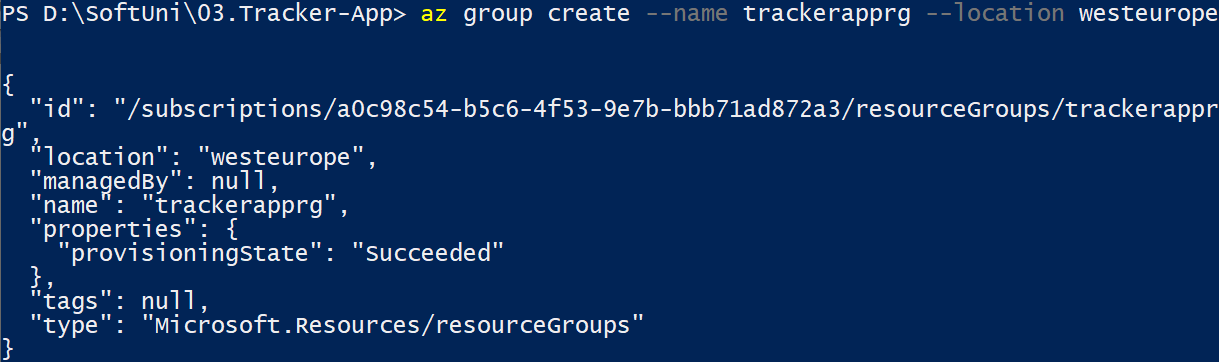
First, **login to** Azure through Azure CLI with the az login **command**. A **web browser will be opened** and you should **enter your credentials**. Then **login** should be successful:



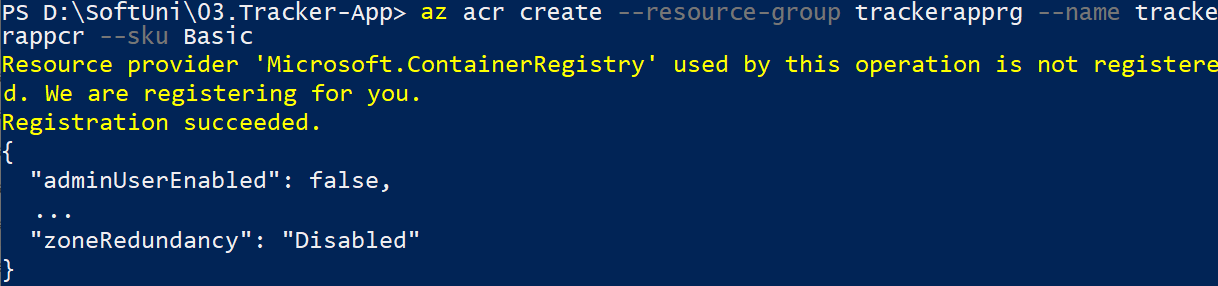
#### Create Azure Container Registry

Now you need a **resource group** to **deploy the container registry** that we will create to.

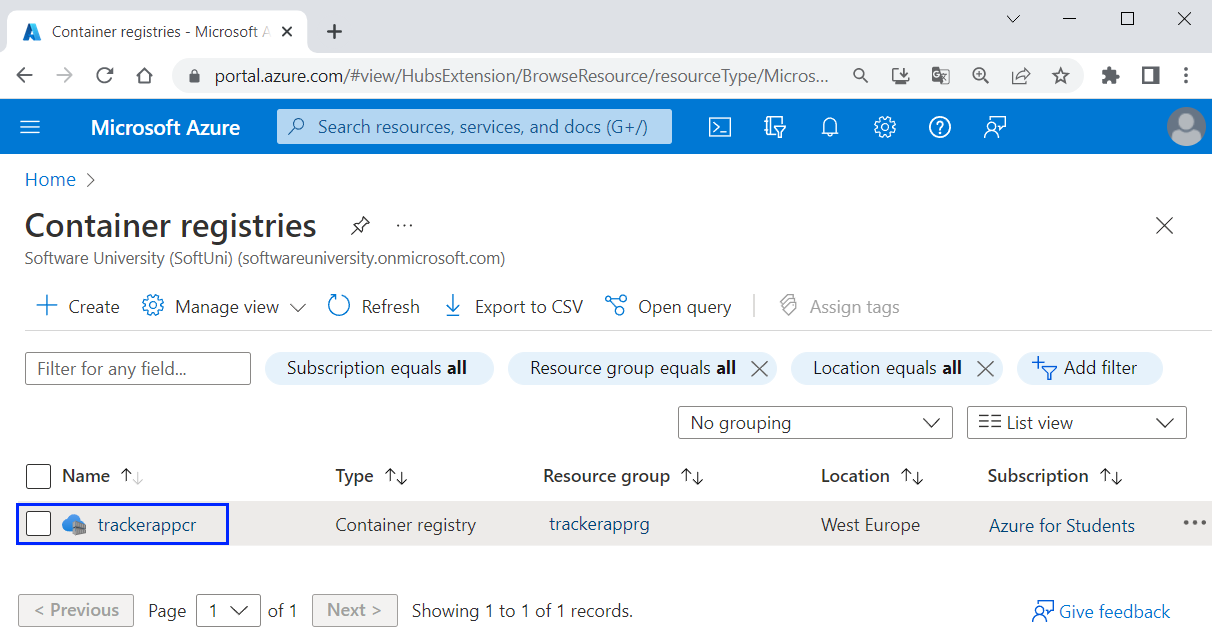
**Create a new resource group** in Azure with the az group create command. **Name the group** in a way you want and set it in the "westeurope" **region**:

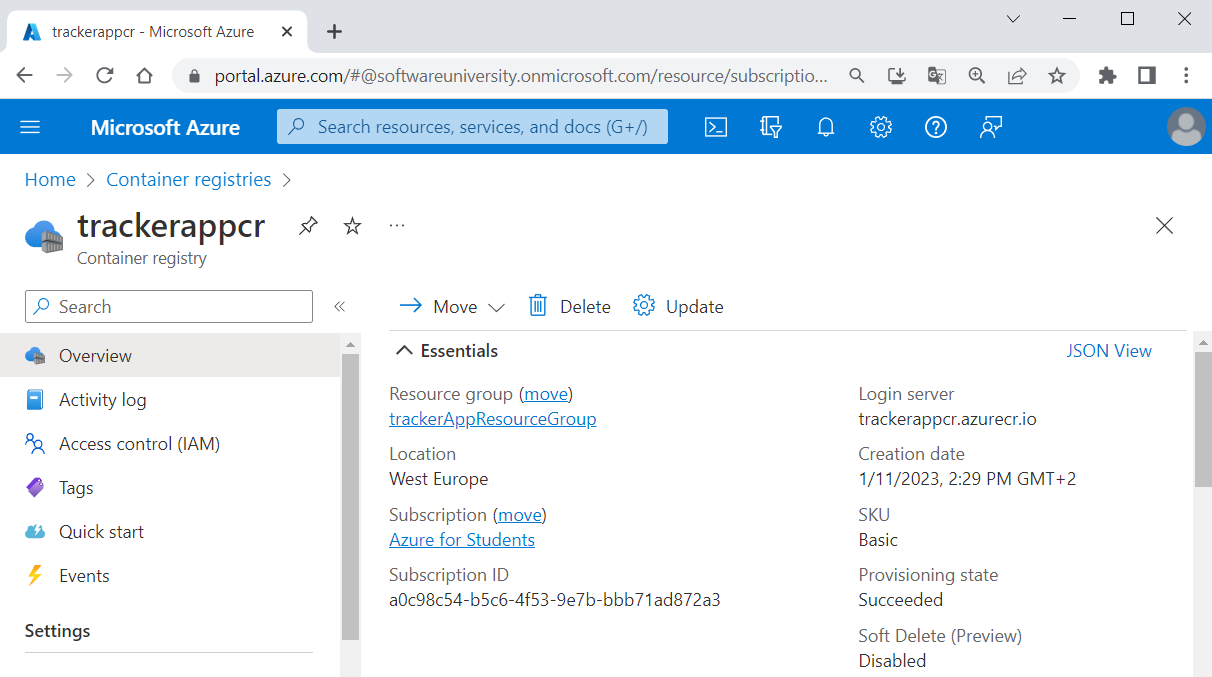


Once you've created the resource group, **create an Azure container registry** with the az acr create command. The **container registry name** must be **unique within Azure**, and contain **5-50 alphanumeric characters**:



You can also **see the container registry** in Azure Portal by going to the "Container Registries" **page**:





Next, you need to **log in to the container registry** you created, so that you can **push images** to it. You should use the az acr login **command** and **provide the name of the registry** you created:

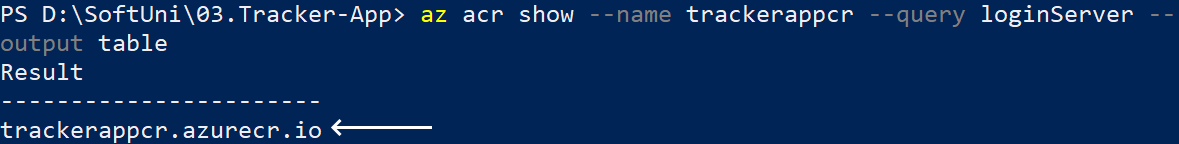


Login should be **successful**.

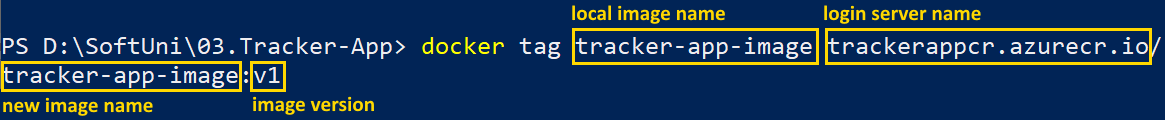
#### Tag Container Image

To **push a container image** to a **private registry** like Azure Container Registry, you must first **tag the image** with the **full name of the registry's login server**.

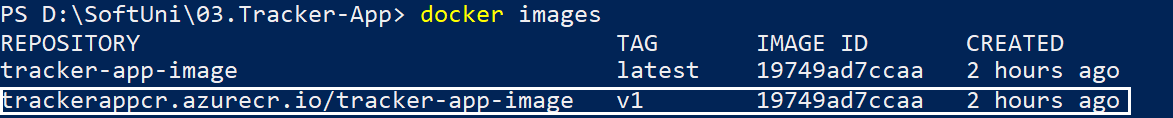
Get the **full login server name** for your **Azure container registry**:



Now you should **tag the container image** we created in the previous step with the **login server of your container registry**. Also, add the :v1 **tag** to the **end of the image name** to indicate the **image version number**. Do it like this:



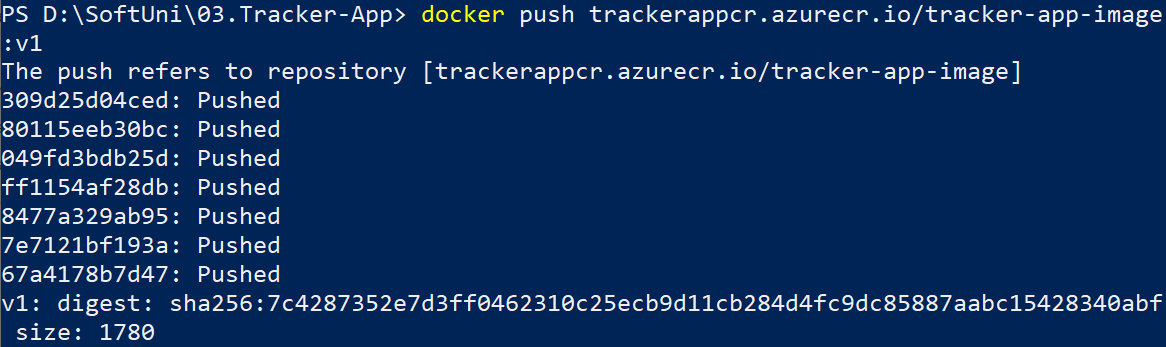
You can look and see that the **new image is created**:



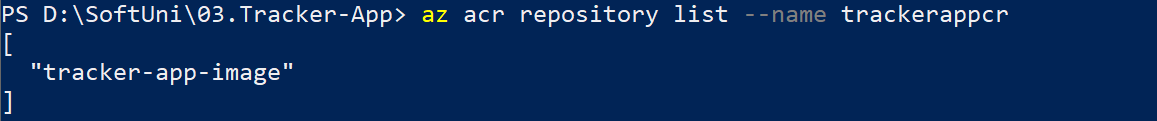
Now you **have your image**.

#### Push image to Azure Container Registry

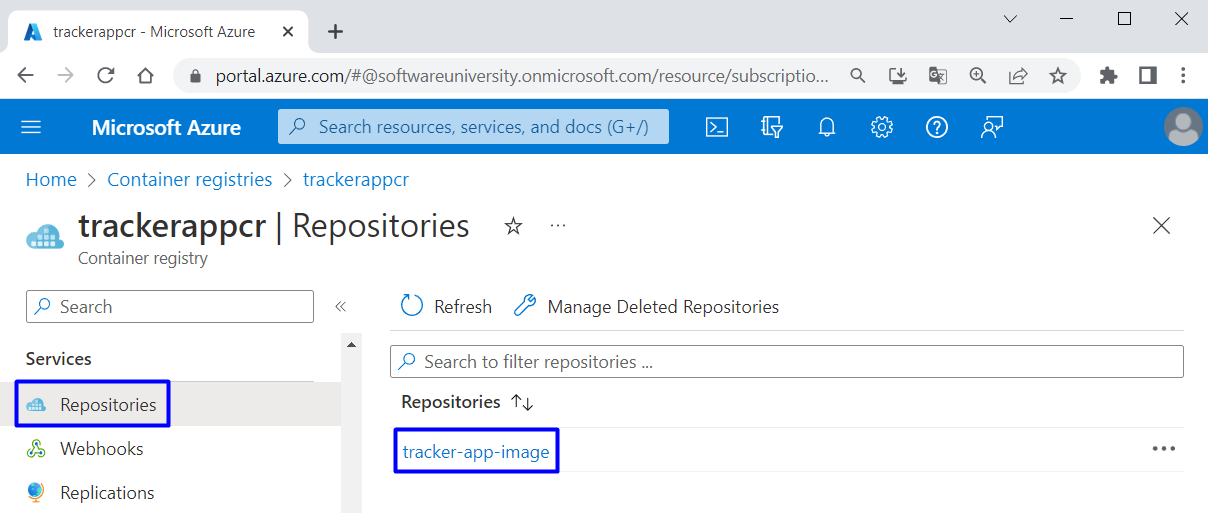
Now you can finally **push your new image** to Azure Container Registry. Do it as shown below:

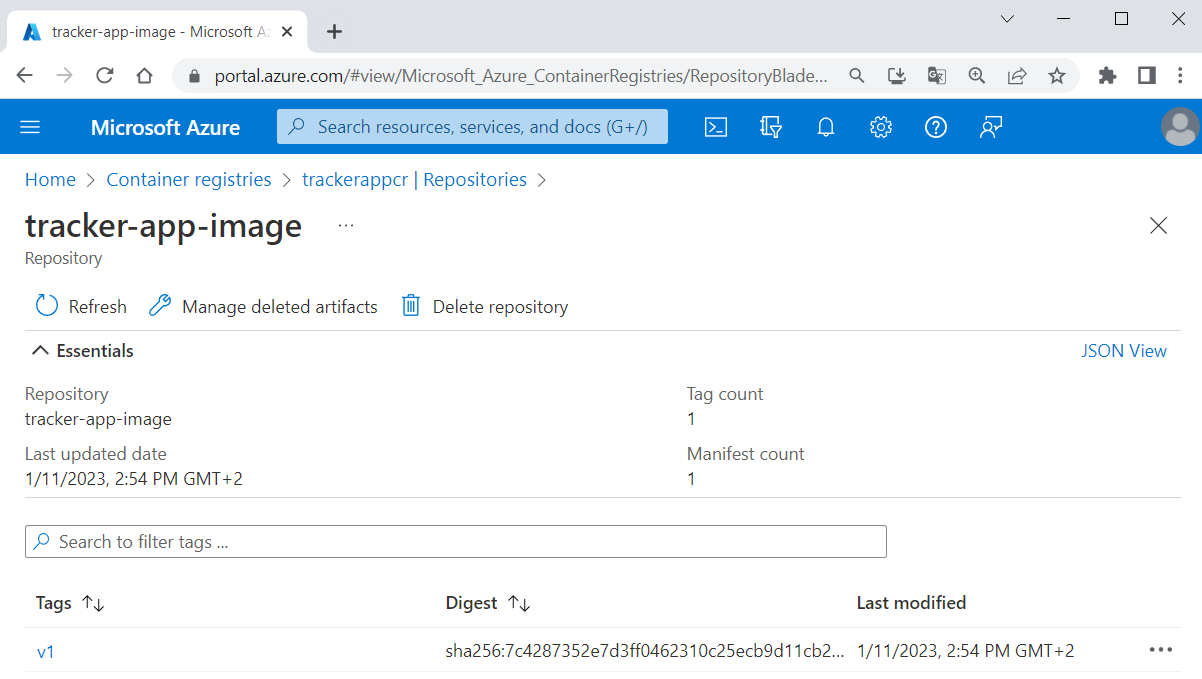


Your **image should be pushed successfully**. To verify this, **list the images** in your **container registry**:



You have you **image in** Azure Container Registry. You can also **check in** AzurePortal when you select the **container registry** 🡪 [Repositories]:





In this step you **prepared an Azure container registry** for use with Azure Container Instances, and **pushed a container image** to the registry.

### Step 3: Deploy a Container Application

Finally, we have everything needed to **deploy a container to** Azure Container Instances.

#### Get Registry Credentials

When you **deploy an image** that's **hosted in a private Azure container registry** like the one created, you must **supply credentials** to access the registry. For this to happen, you shall **create and configure** an Azure Active Directory (AD) **service principal** with pull **permissions** to your **registry**.

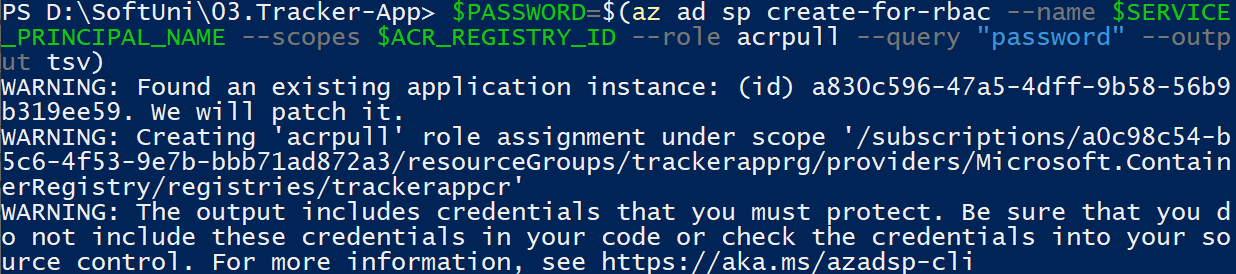
To do this, you will need the **following commands**:

|  |
| --- |
| $ACR\_NAME='$containerRegistry'  $SERVICE\_PRINCIPAL\_NAME='$servicePrincipal'  *# Obtain the full registry ID*  $ACR\_REGISTRY\_ID=$(az acr show --name $ACR\_NAME --query "id" --output tsv)  *# Create and configure the service principal with pull permissions to your registry*  $PASSWORD=$(az ad sp create-for-rbac --name $SERVICE\_PRINCIPAL\_NAME --scopes $ACR\_REGISTRY\_ID --role acrpull --query "password" --output tsv)  $USER\_NAME=$(az ad sp list --display-name $SERVICE\_PRINCIPAL\_NAME --query "[].appId" --output tsv)  *# Output the service principal's credentials*  echo "Service principal ID: $USER\_NAME"  echo "Service principal password: $PASSWORD" |

You should **modify them** for your environment:

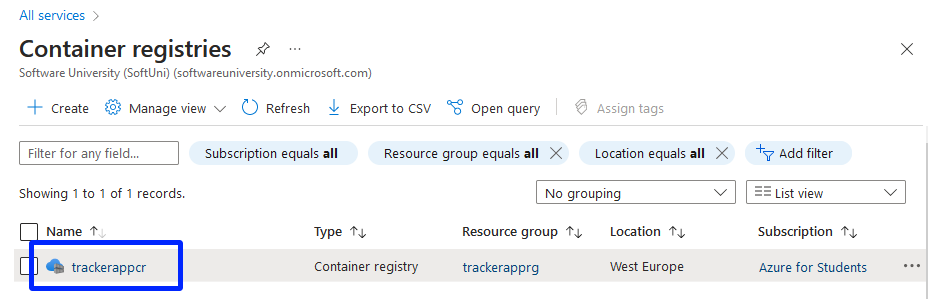
* ACR\_NAME should keep the **name of your Azure container registry**, which you already created
* SERVICE\_PRINCIPAL\_NAME should keep a **new name for your service principal**, which should be unique within your Active Directory tenant

**Execute the commands** one by one with the changes:

You should **have no errors** and finally you should have your **service principal id** **and** **password**. Once you have these **credentials**, you can **configure your applications and services** to authenticate to your container registry as the service principal. **Copy them** because you will need them.

**NOTE:** If you get the "**Insufficient privileges to complete the operation.**" error message, go to "Container registries" in the **Azure Portal** and select the **trackerapp** container registry**:**



After that, select **Access keys** and enable the **Admin user:**



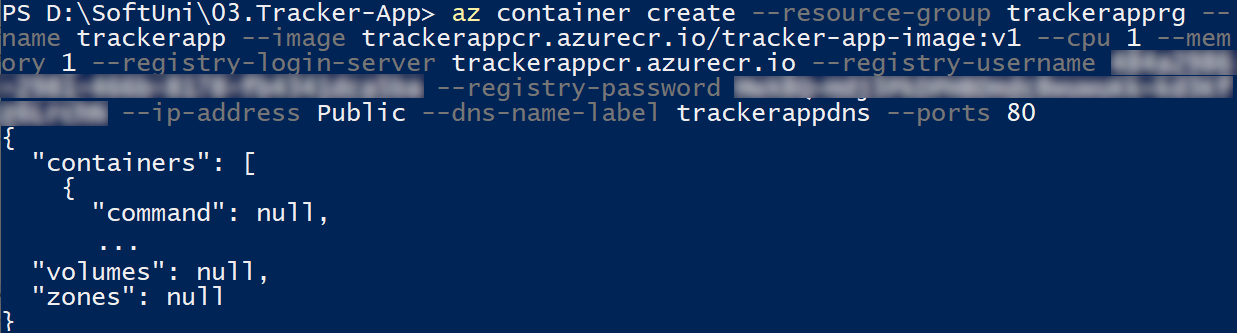
#### Deploy Container

Now use the az container create **command** to **deploy a container in Azure**. Use this command and **replace** <resource-group-name>, <image-name>, <acrLoginServer>, <service-principal-ID> and <service-principal-password> with **your corresponding values**. Also, **replace** <container-name> with a **name for your container** and <aciDnsLabel> with a **new DNS name** that you like:

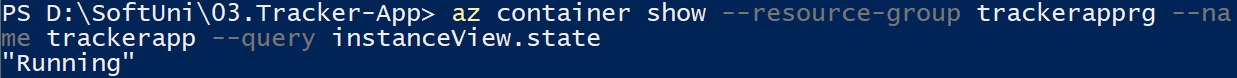
|  |
| --- |
| az container create --resource-group <resource-group-name> --name <container-name> --image <acrLoginServer>/<image-name>:v1 --cpu 1 --memory 1 --registry-login-server <acrLoginServer> --registry-username <service-principal-ID> --registry-password <service-principal-password> --ip-address Public --dns-name-label <aciDnsLabel> --ports 80 |

**NOTE:** If you have enabled the **Admin user** in the previous step, just replace **<service-principal-ID> with <username>** and **<service-principal-password>** with **<password>**

**Container deployment** should be **successful**:



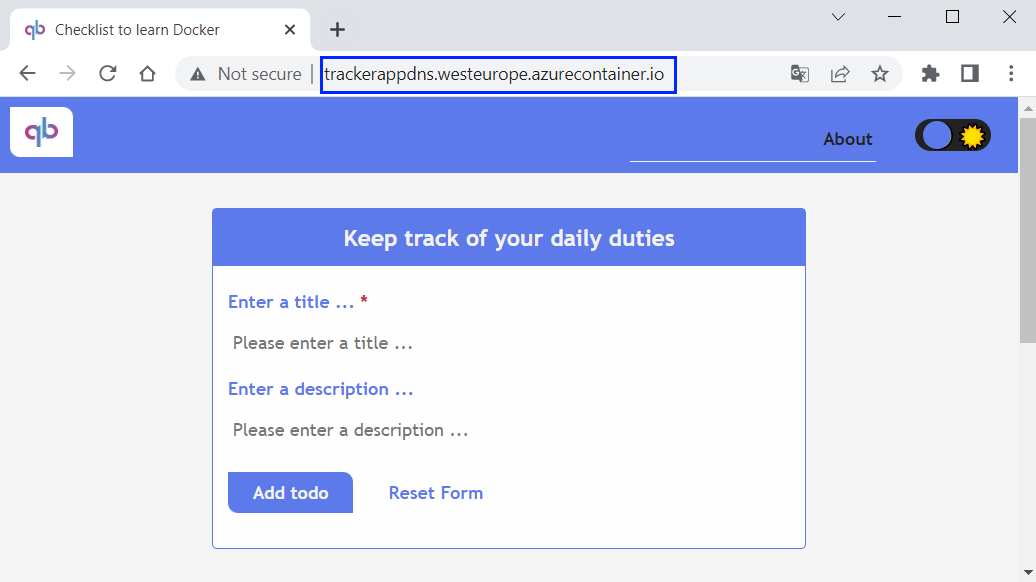
To **view the container state** **of deployment**, use:



Once the **deployment succeeds**, display the **container's fully qualified domain name** (FQDN):



Use the FQDN to **access your app in the browser** (it contains the **DNS** **label** you chose):



It should be **fully working**.

#### Explore and Remove Container

You can look at the **created container instance** in Azure Portal too:

## Deploy TaskBoard App to Azure through Azure Portal

In this task, we will deploy the TaskBoard **ASP.NET app** to Azure App Service and **connect to an** Azure SQL Database again, but this time it will be **through** AzurePortal. We will use Azure **+** GitHub **integration**, which enable linking between the **code in the repo** and the **Azure app in production** and use of GitHub Actions **workflows**.

### Step 1: Upload Project to GitHub

The first step is to **upload** the "TaskBoard" **project code** from the **resources to GitHub**. **Do it** in a way you like. When ready, it should look like this:

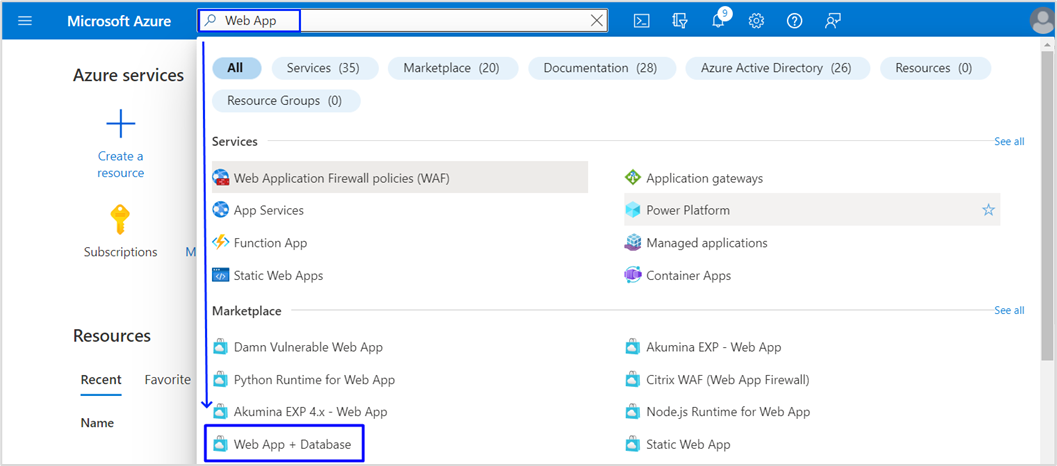


Now we are ready to **start working with** Azure.

### Step 2: Create Azure Resources

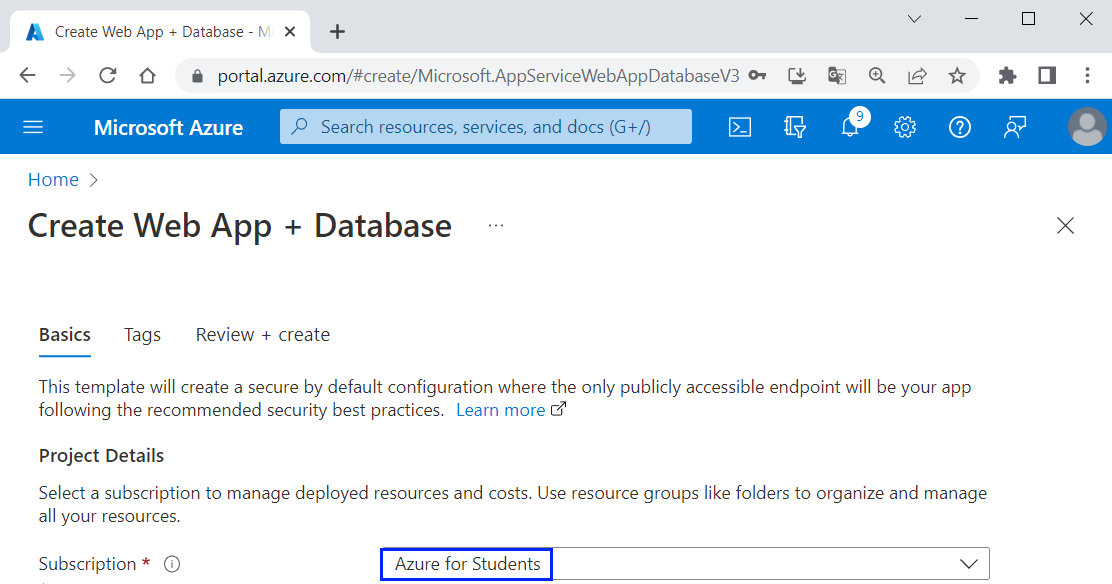
In this step, we will **create the** Azure **resources** that our app needs – App Service and Azure SQL Database, from Azure Portal. **Log in** to Azure Portal.

In the **search bar**, type and select the "Web App + Database" **item** from the Marketplace:

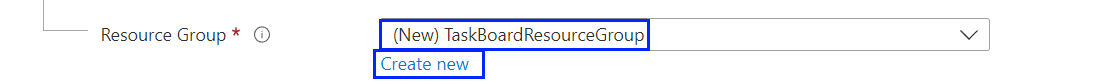


In the "Create Web App + Database" **page**, fill in the **fields**. You should:

* **Not change the subscription**



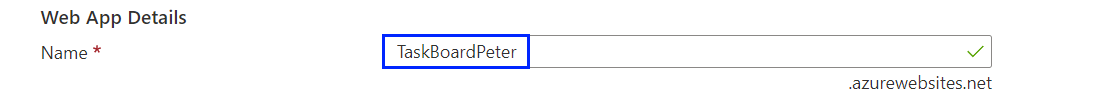
* Create a **new resource group** named "TaskBoardResourceGroup"



* Choose a **region** near you, for example [West Europe]



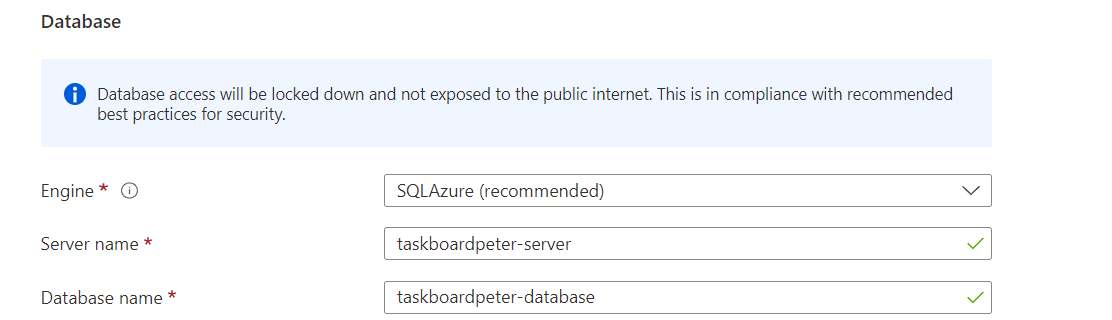
* Set a **unique name** for your **Web app** – you can use "TaskBoard*{your name}*":



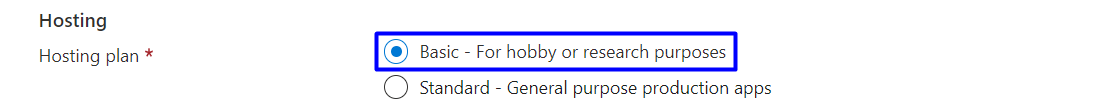
* Choose [.NET 6 (LTS)] for **runtime stack** because the TaskBoard **app** is written on it:



* Make sure that **database data** (filled in by default) is correct: **database engine** should be "SQL Azure" and **server** and **database names** should be **suitable**:



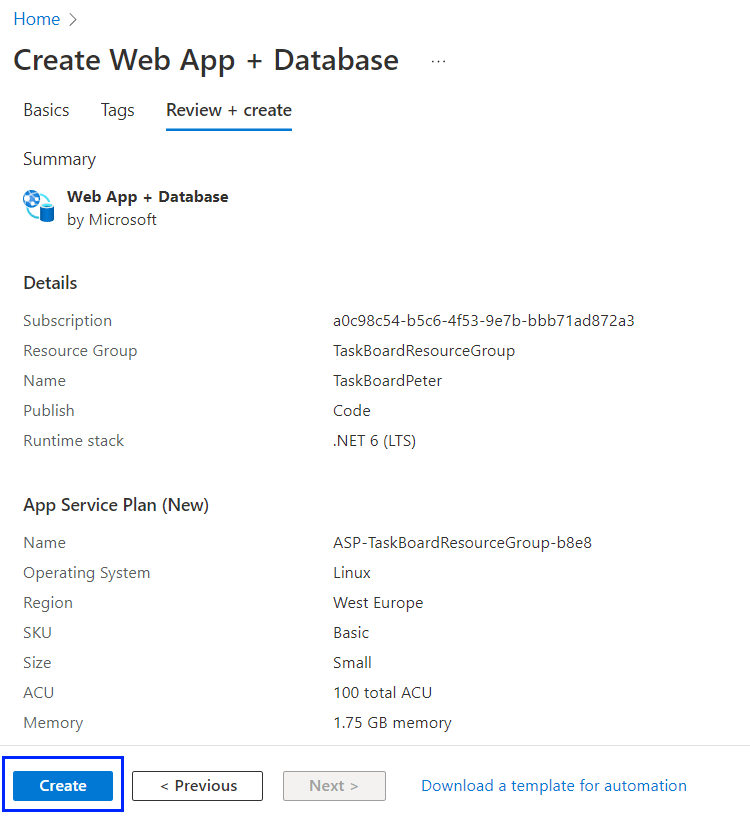
* Choose a "Basic" **hosting plan**:



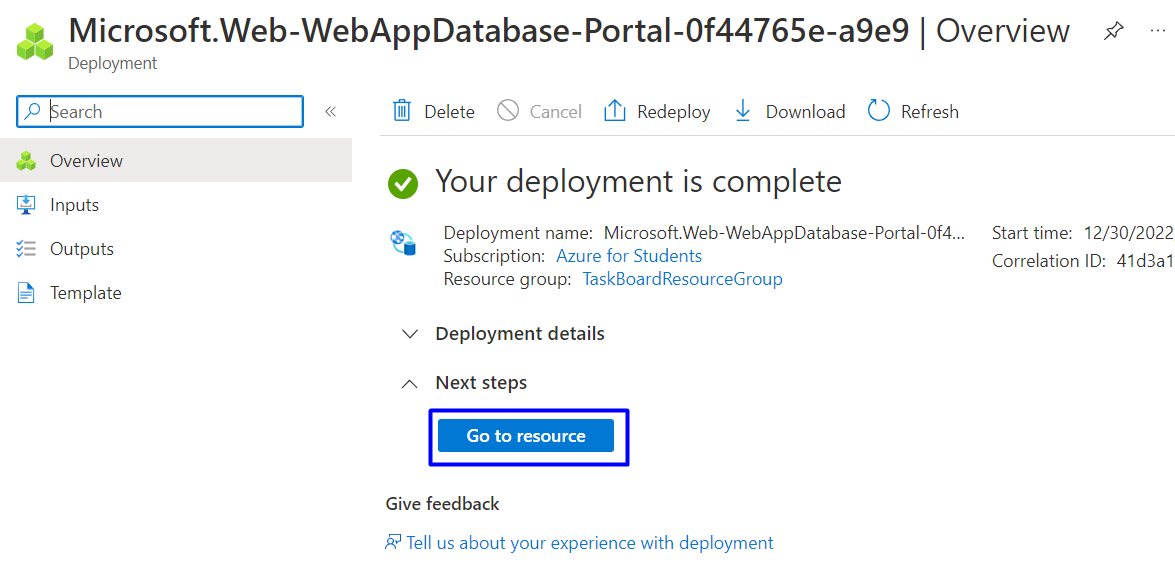
* Now click [Review + Create]:



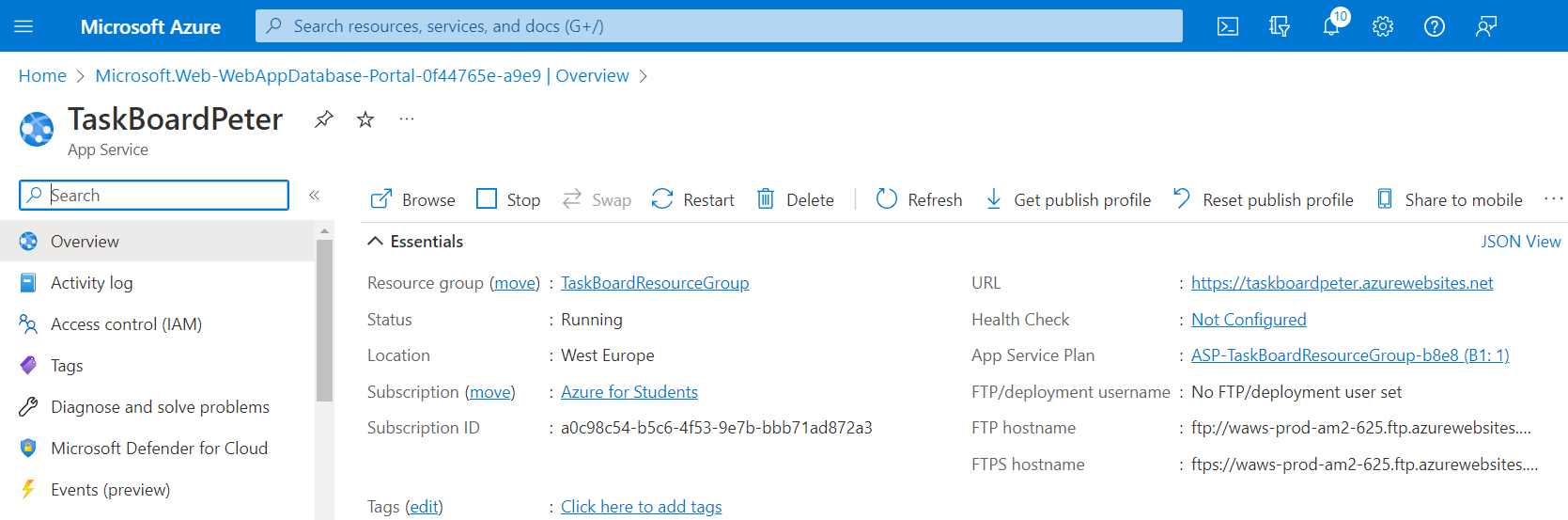
Finally, click [Create]:



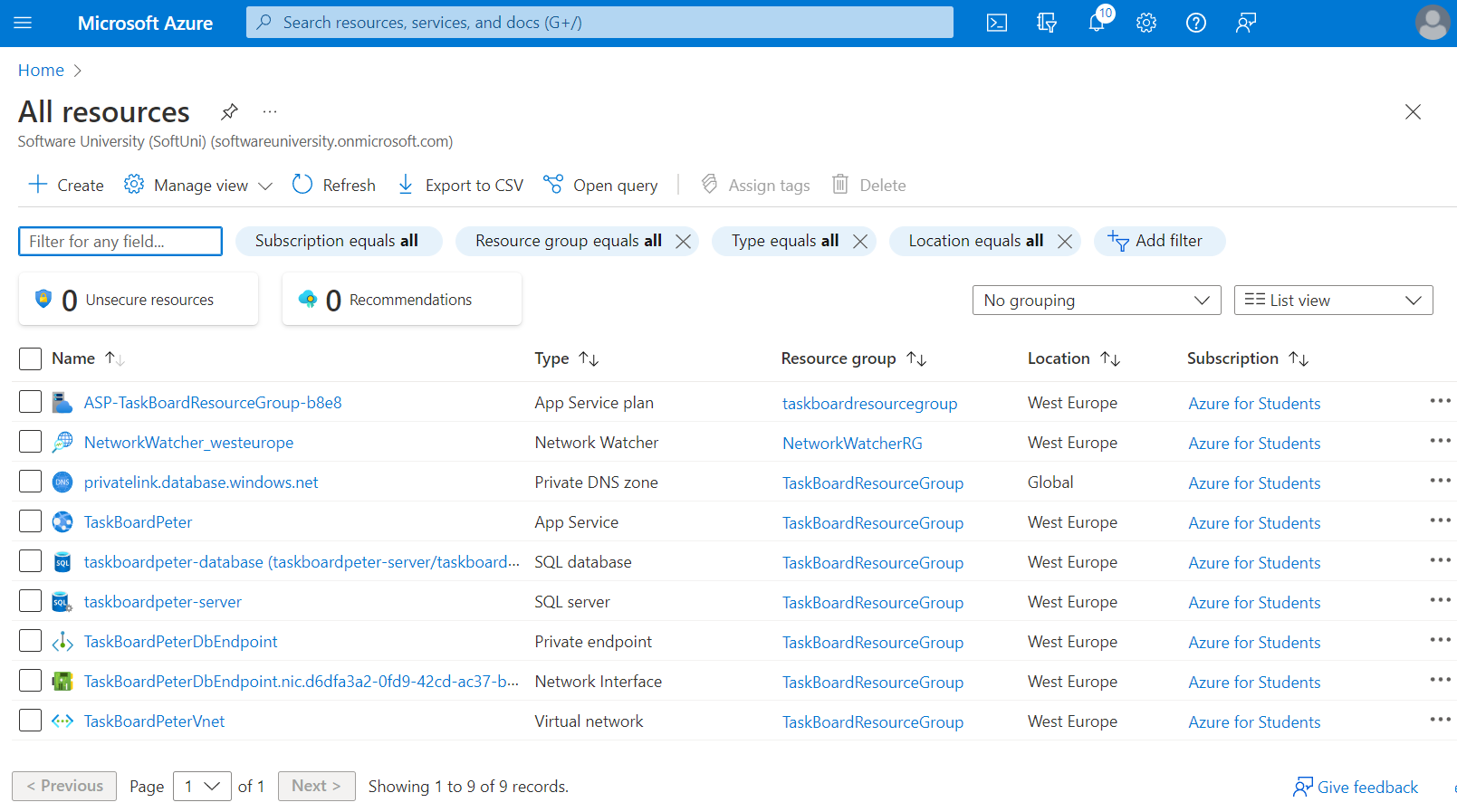
**Wait** until the **deployment is complete** and click on the [Go to resource] **button**:



You can look at the App Service **app** you created:



And you can go to the "All Resources" **page** and look at the **created resources**:

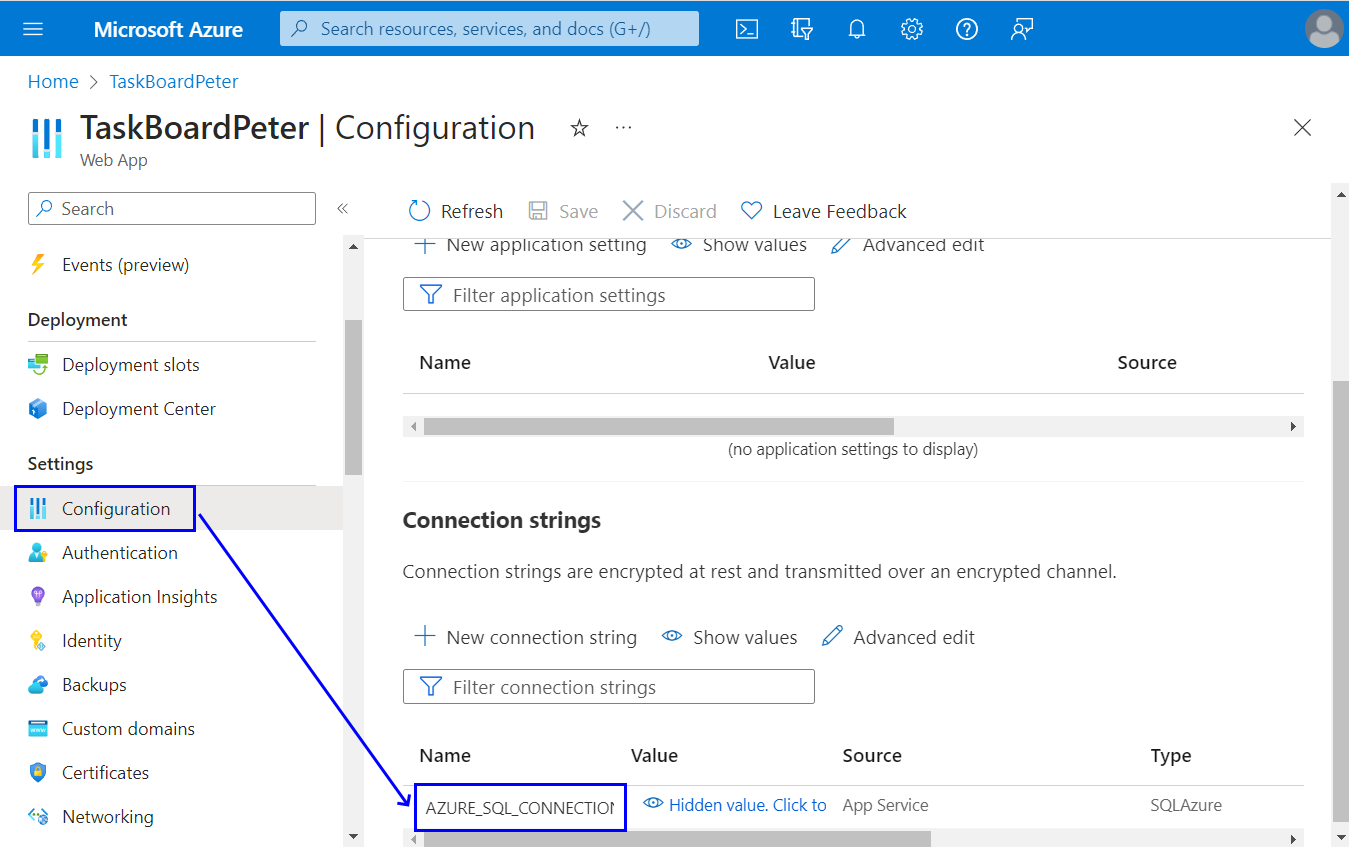


You may notice that **they are a lot more** than when we **added the same app** from Visual Studio.

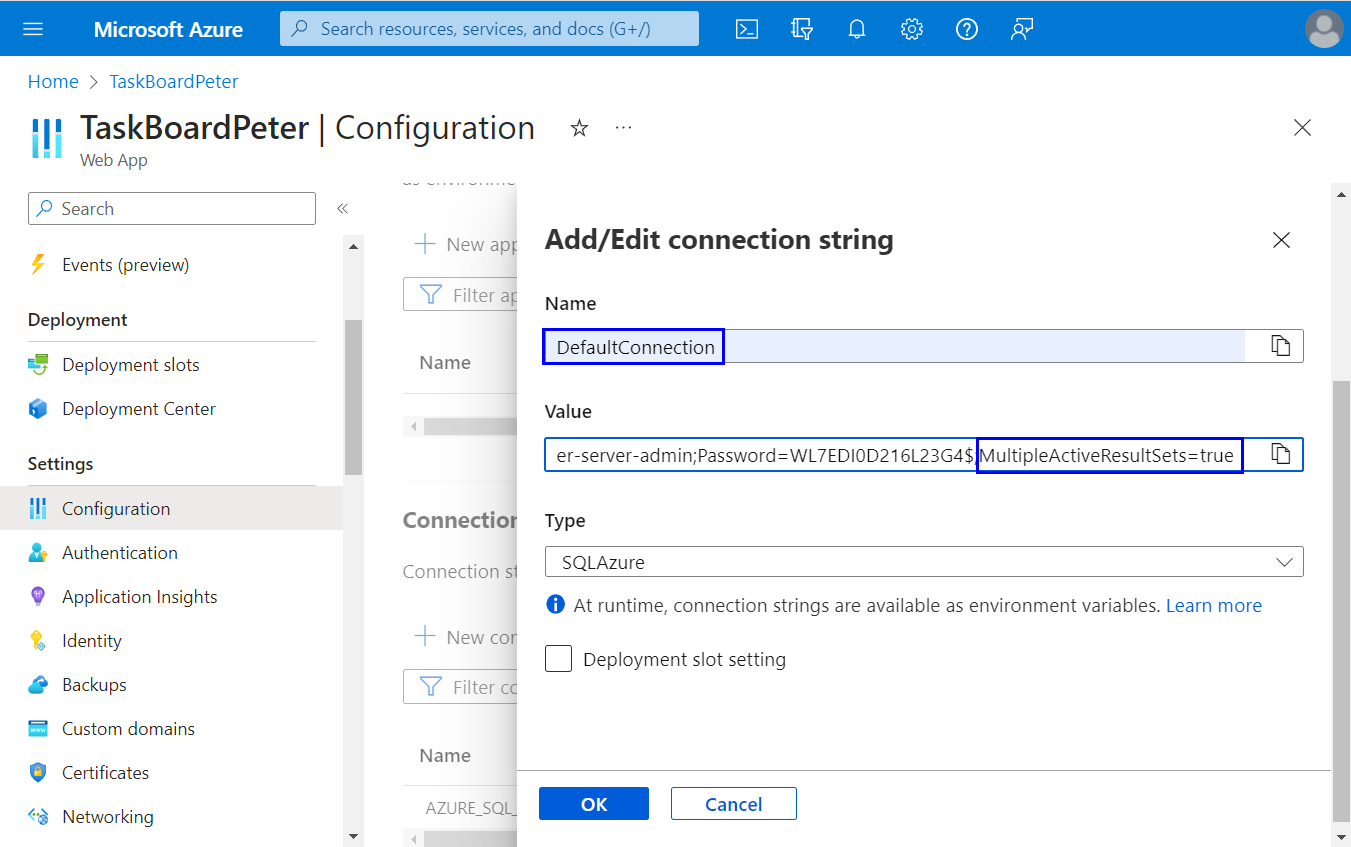
### Step 3: Set Up Database Connectivity

You already have a **connection string** generated for you. You will need it to set it so your **app and the database in** Azure **can connect successfully**.

To **access the connection string**, go back to the "App Service" **page of your app** 🡪 [Configuration], scroll to "Connection strings" and **click on the connection string** you have:



Now change the "Name" **variable** **value** to the one from your **app's** appsettings.json **file** – in our case it's "DefaultConnection":



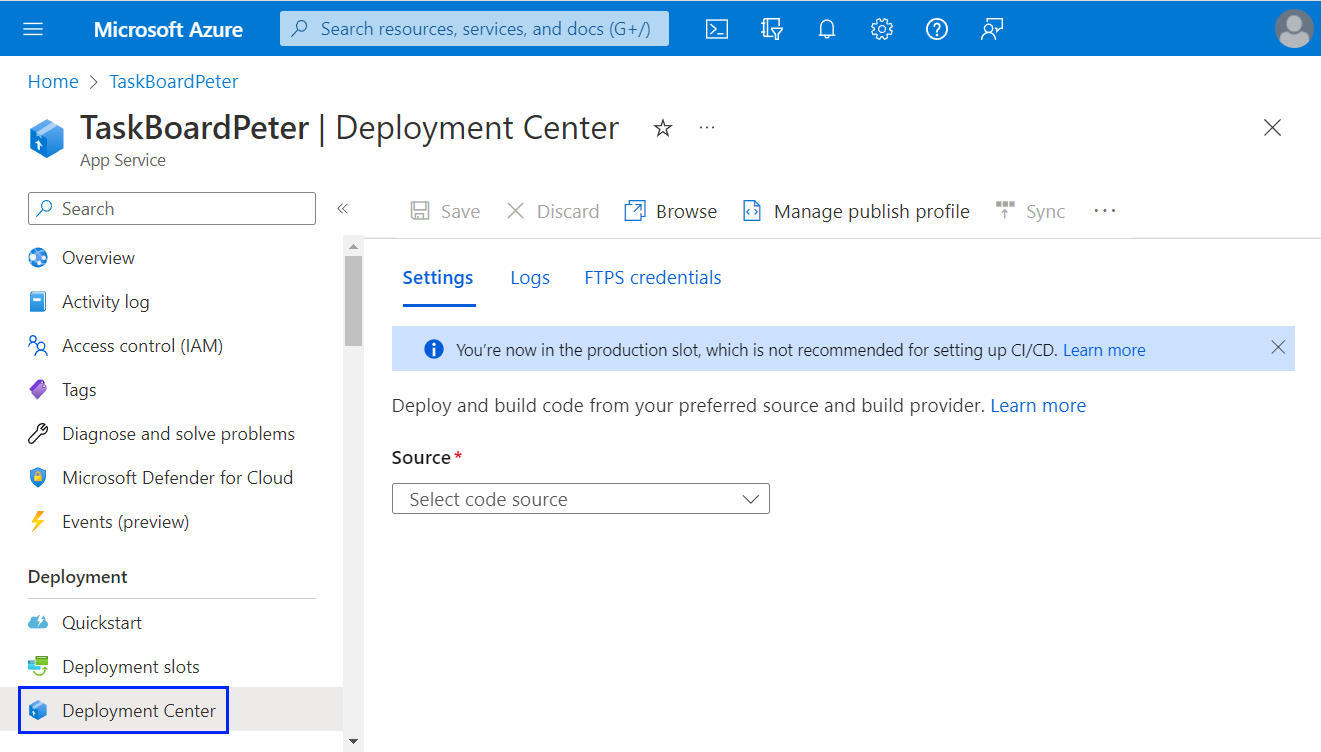
Also, add MultipleActiveResultSets=true at the end of the **connection string** to allow the execution of **multiple batches** on a **single connection**.

Then click on [OK] and [Save] to **update Web app settings**.

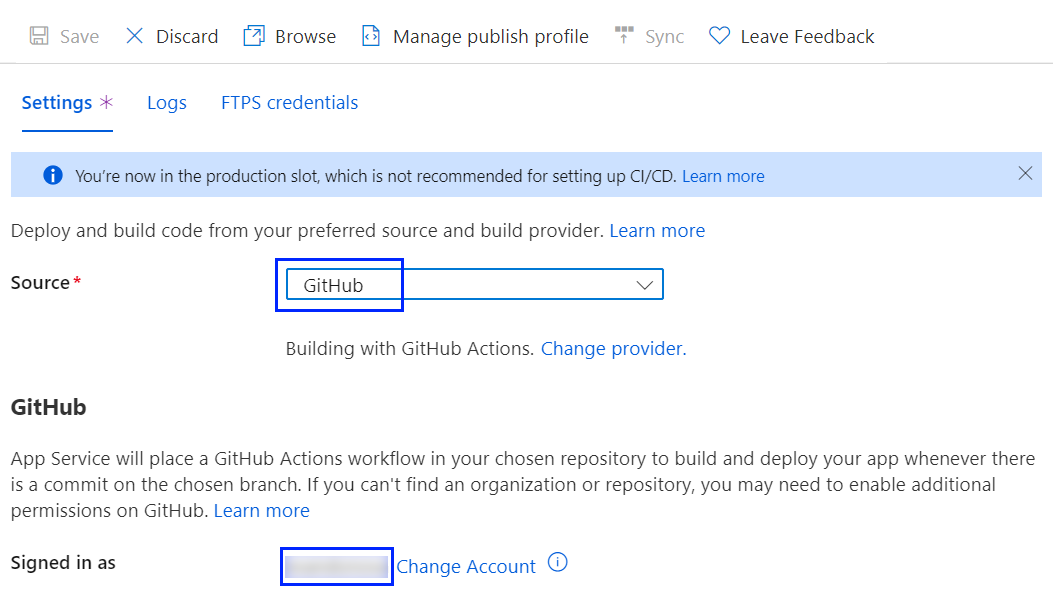
### Step 4: Deploy Code

It's time to **deploy the** TaskBoard **app's code** from GitHub to Azure. When ready, your **changes to the GitHub repo** will be **reflected** in the **Azure app in production**.

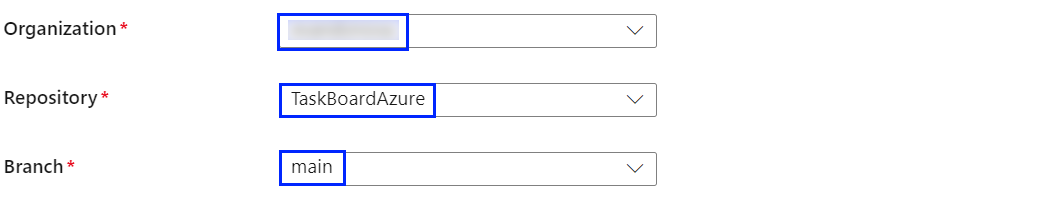
Go to the "App Service" **page** 🡪 [Deployment Center]:



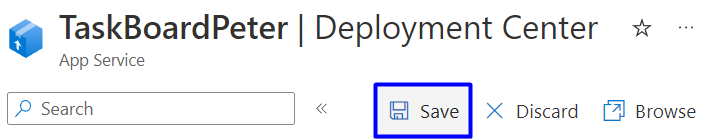
For "Source", choose [GitHub] and **sign in** with your GitHub **account**:



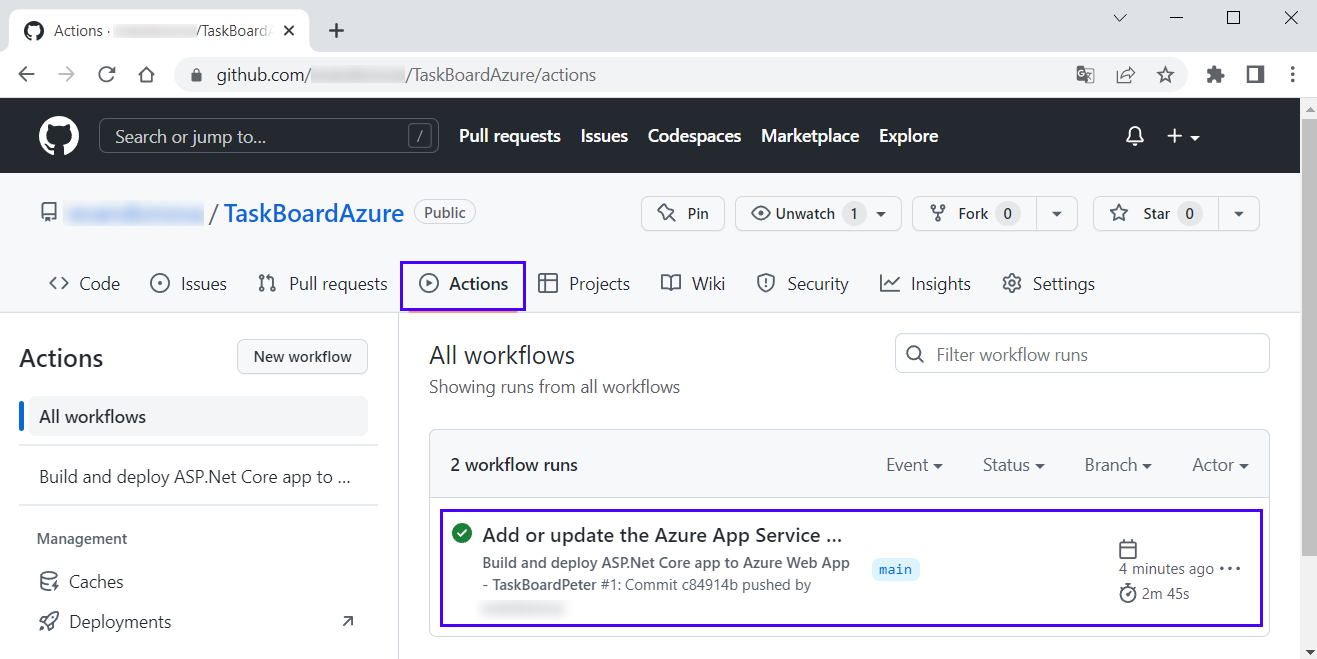
Then **choose the organization** of your project (your **profile**), the **repository** you created and the main **branch**:



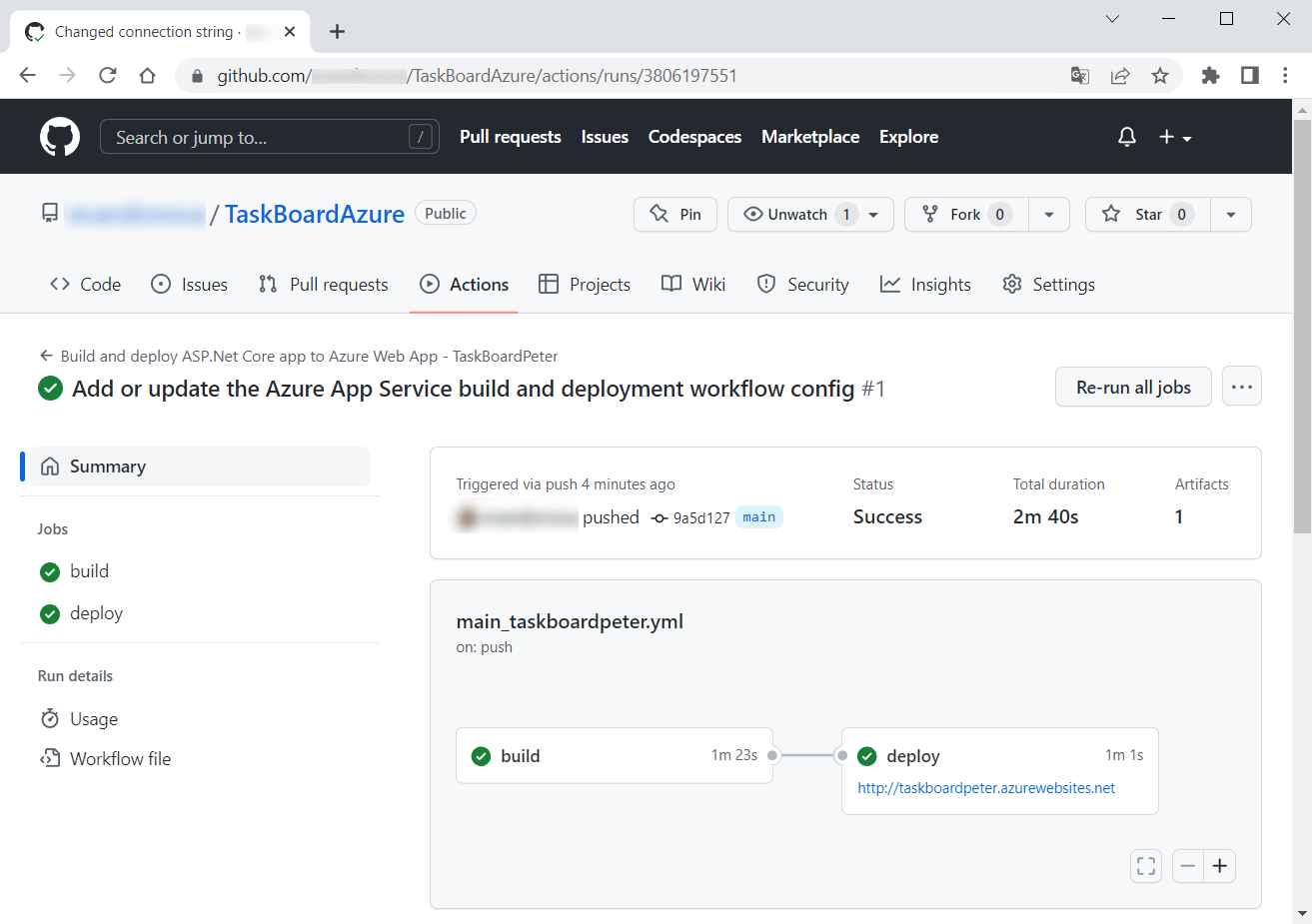
Now click on [Save] to **save the changes**:



And in GitHub Actions, you have a **running workflow**, which should be **successful after a few minutes**.



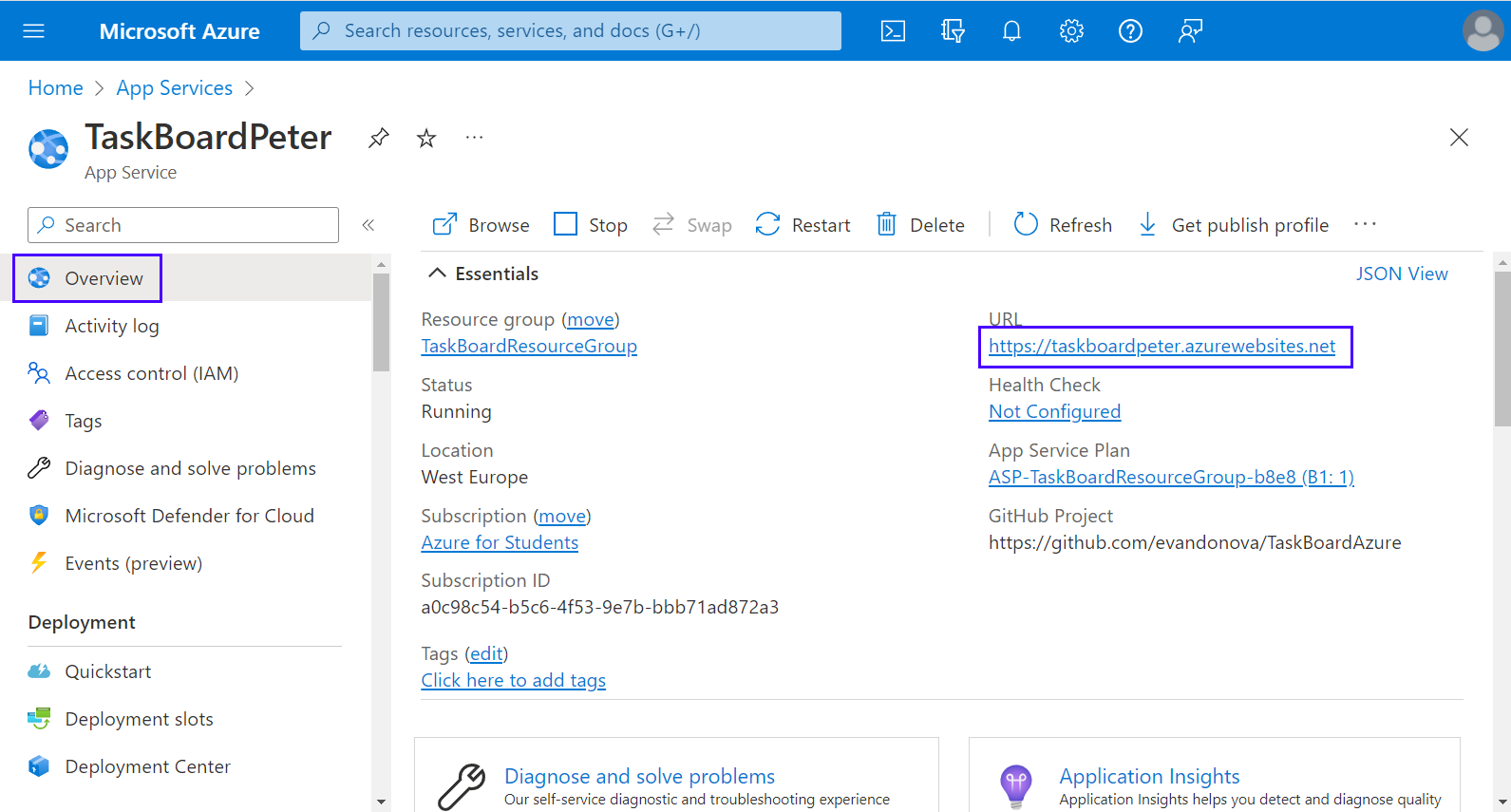
Notice that the **workflow file** defines **two separate stages**, build and deploy:



### Step 5: Browse the App

Finally, you can **access your app** in Azure.

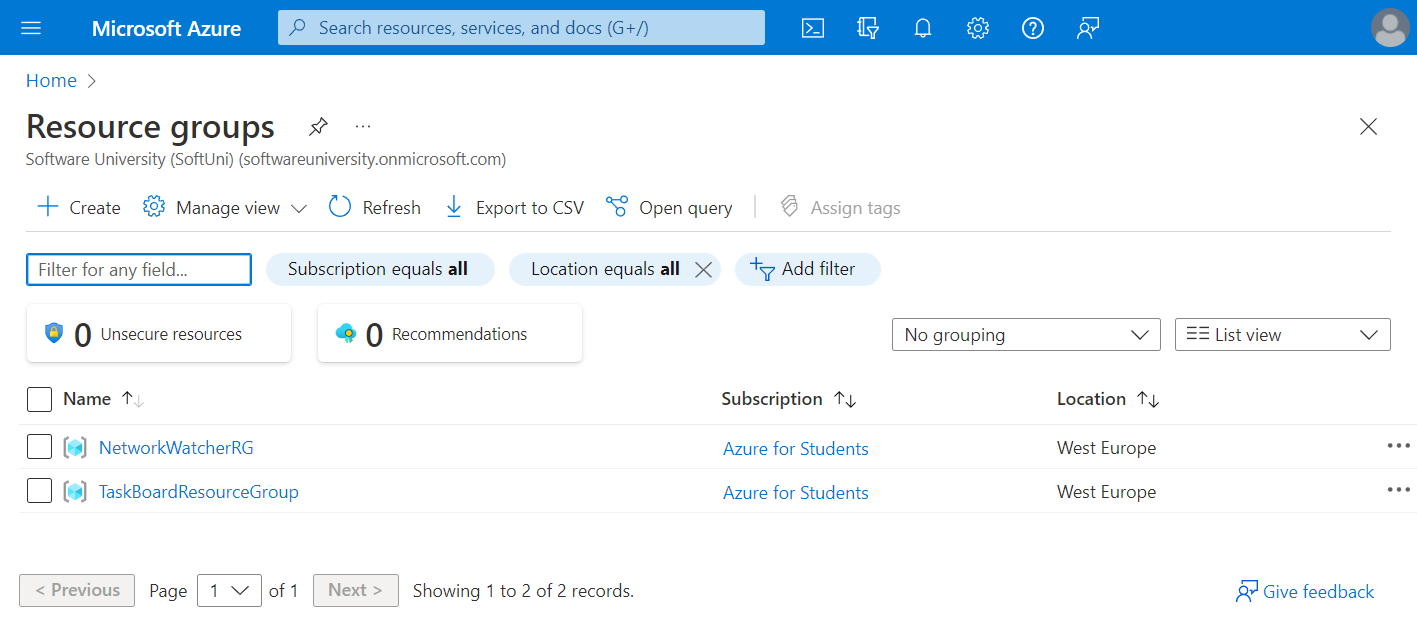
To **get its URL address**, go to the "App Service" **page** 🡪 [Overview] and **click the URL of your** **app**:



You can also **navigate directly** to https://<app-name>.azurewebsites.net. Your **site** should **load** and **work**:



In addition, if you **log in** to Azure Portal and look at the **resource groups** you have, you will see that they are two – you have the "TaskBoardResourceGroup" and the "NetworkWatcherRG" **groups**:

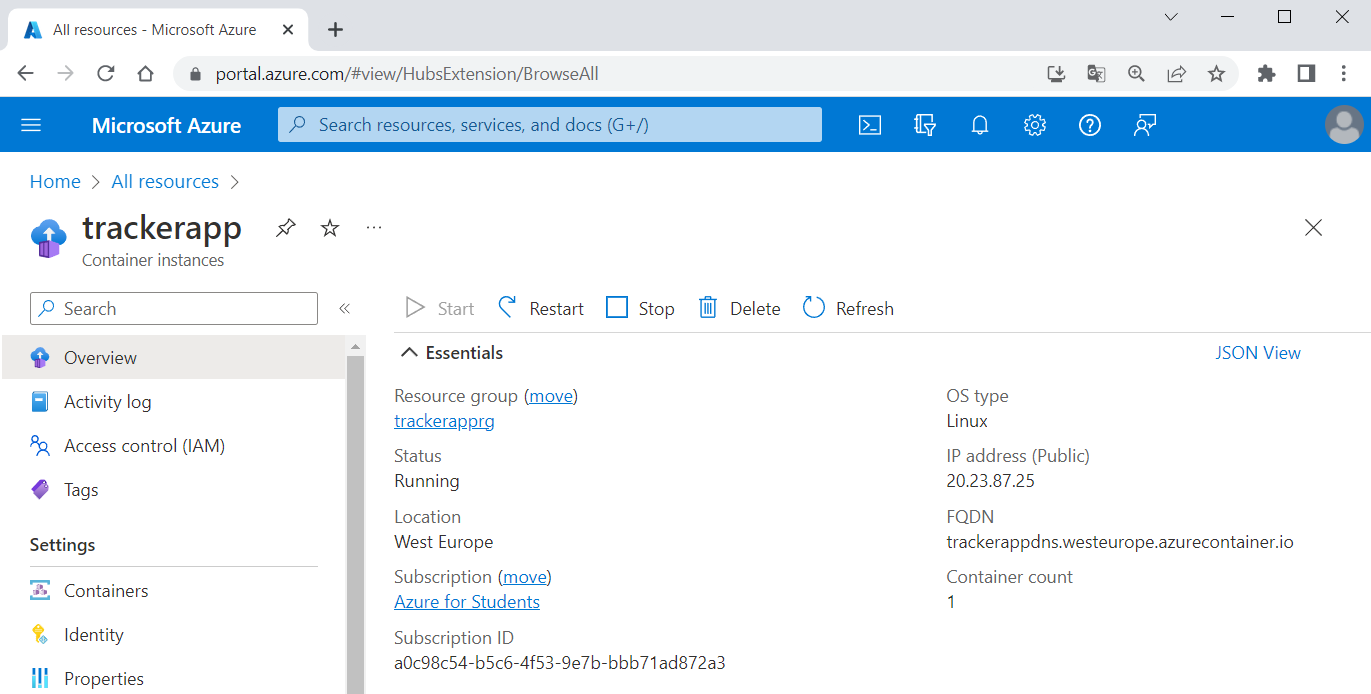
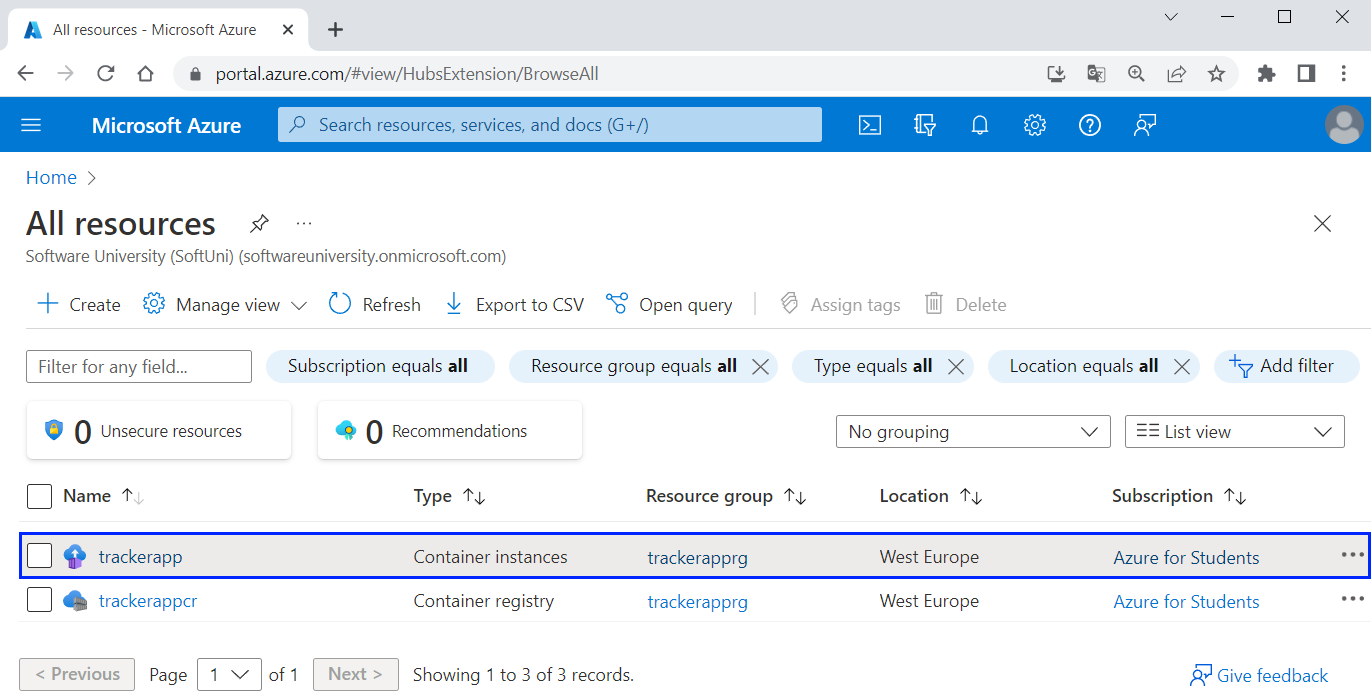


The "Network Watcher" **resource** represents an Azure **region** in the **resource group** and provides tools to monitor, diagnose, view metrics, and enable or disable **logs** **for resources** in an **Azure virtual network**.

You can **delete it** together with the "TaskBoardResourceGroup" when you **don't need your Azure app** anymore.

**Note**: in our case, we have the Database.EnsureCreated() **method** to **create the database** and **apply migrations** if any. If we **don't have** the Database.EnsureCreated() or Database.Migrate() **method**, we should either **add them to the project's database context class** or **fulfill additional steps in Azure and GitHub to migrate the database**: **install EF Core** and **generate a database migration bundle** in the **workflow YAML file in GitHub** and then **generate database schema** from a **SSH terminal in Azure**. You **delete the migration method** in your code, **look online** and **try to do this**, if you want.

Anyway, you now know **how to deploy an ASP.NET Core app to** AzureWebApps from Azure Portal, using a GitHub Actions **repo**, too.



You can also view the **log output of the container** from Azure CLI:



And finally, **clear all resources** by **deleting the resource group** with az group delete:



Now we know how to **create resource groups**, **container registries**, **container instances**, etc. and **deploy a container app** in Azure.