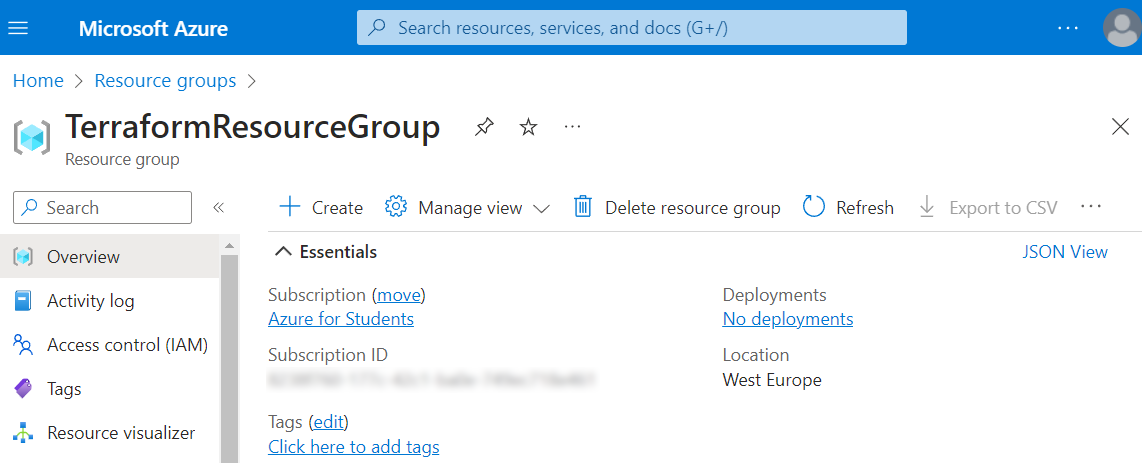
# Exercise: Infrastructure as Code

Exercises for the ["Containers and Clouds"](https://softuni.bg/trainings/4359/containers-and-cloud-january-2024) course @ SoftUni

### Azure Resource Group

Now you have a task to **create a** Terraform **configuration** to **deploy an** Azure **resource group**.

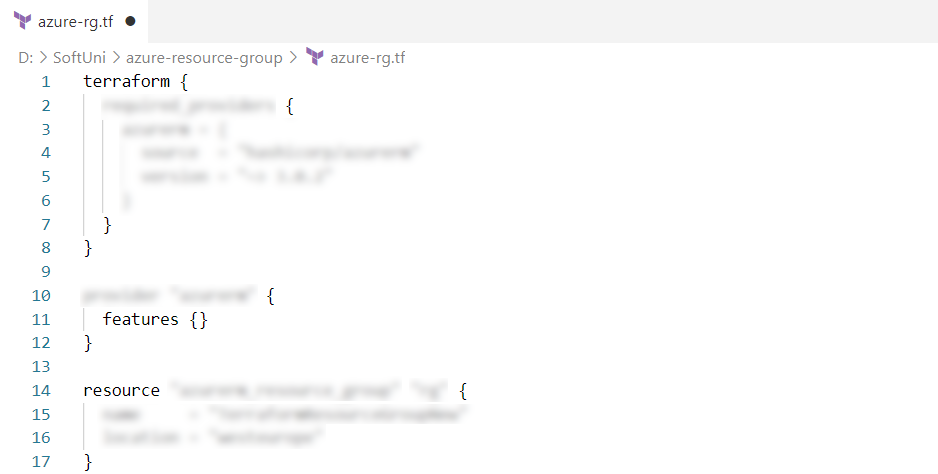


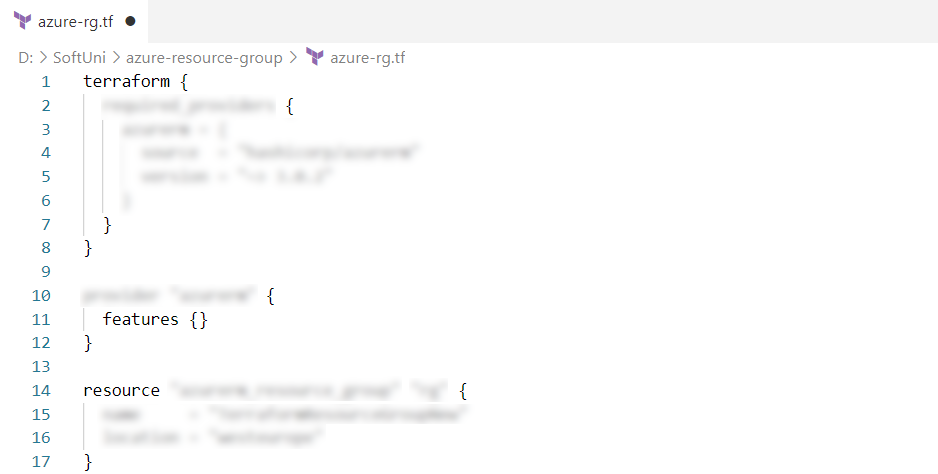
#### Hints

Open a **terminal** (for example PowerShell), **create a Terraform configuration folder** with an **empty configuration file** and **follow the steps below** to fulfill the task:

1. **Authenticate** using the Azure CLI, i.e. **log in to** Azure, as Terraform **must authenticate** to create infrastructure
2. **Write the configuration** for creating an Azure **resource group**
   * You need an Azure **provider**, available here: <https://registry.terraform.io/providers/hashicorp/azurerm/latest>
   * The Azure **provider** needs a feature {} **block** in the **configuration**
   * At the end, the **resource group** should be created using the "azurerm\_resource\_group" Terraform **resource**, whose **required arguments** can be seen here: <https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs/resources/resource_group>

The **configuration file** looks like shown below. The **resource group name and location** are for you to choose:





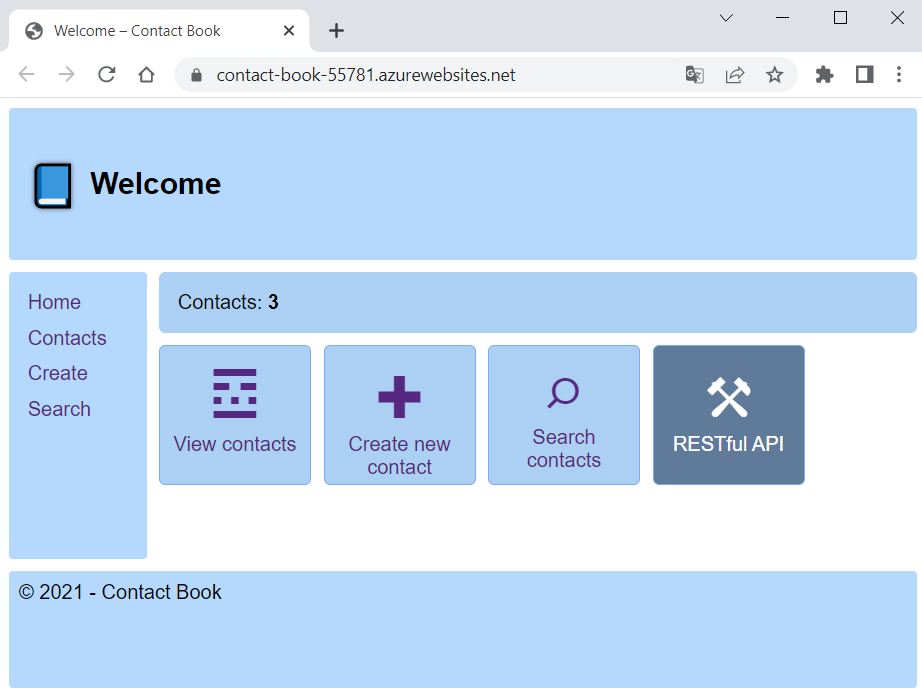
1. **Initialize**, **format**, **validate** and **apply** **your** Terraform **configuration**
2. **Navigate to** Azure Portal in the browser and validate that a **resource group was created**

Later you can **delete the resource group** from Azure again **using** Terraform.

As we know how to **create an** Azure **resource group with** Terraform, let's see how this would be **useful for us in the next task**.

### Azure Web App

You are already **familiar with** AzureWebApps and now you should **use** Terraform to **create a resource** **group**, then **create an** AppServicePlan and finally **deploy the** "ContactBook" **app to** Azure from a GitHub **repo**.



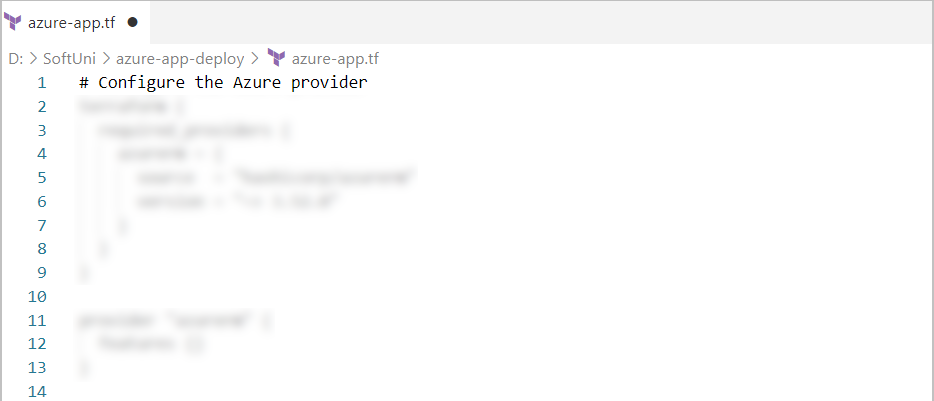
"Contact Book" is a **Node.js app** **without a database**, available here: <https://github.com/nakov/ContactBook>.

#### Hints

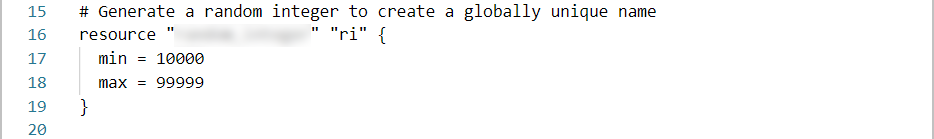
To **fulfill your task**, you need to **create a** Terraform **configuration file**. Find the Terraform **resources** you need in the Terraform Registry and use them: [https://registry.terraform.io](https://registry.terraform.io/).

The **configuration** you should write:

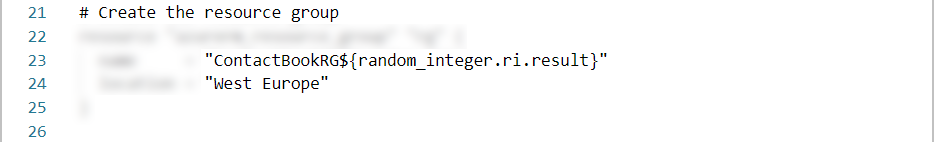
* Uses and configures an **Azure provider** (as in the previous exercise)



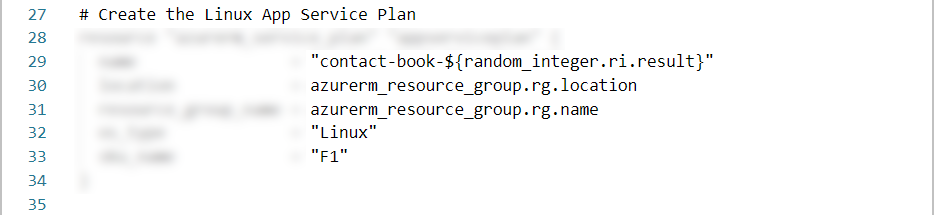
* **Generates a random integer** with **minimum** **and** **maximum** **number range** to be used for **creating unique resource names**



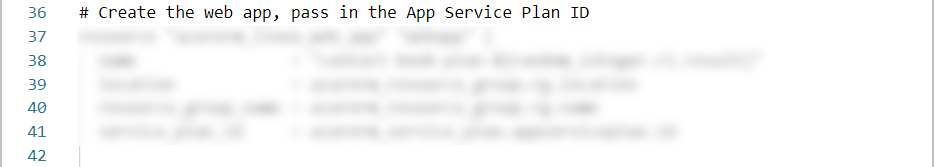
* Creates a **resource group**, whose name **uses the randomly-generated integer** by a **reference to the above resource**

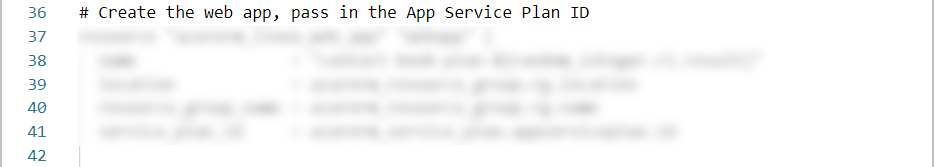


* Creates an App Service Plan with **name**, **location** (**reference the location** from the **resource group**), **resource group name** (reference the **name of the resource group**), **operating system** (set to "Linux") and **type of SKU** (set to "F1")



* Creates an Azure **Linux Web app** with **name**, **location**, **resource group name** and the **id of the service plan** (use **references** to the above resources)

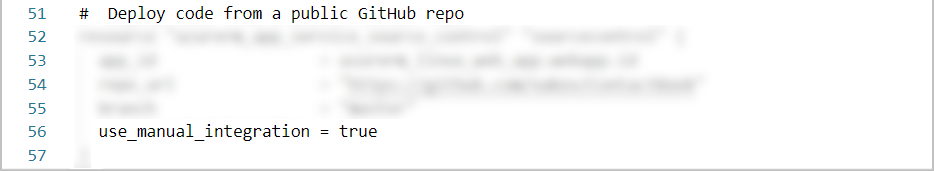




* + In addition, you should **add site configurations** including the **app's Node.js version** and a restriction for the **app to not always be on** (as we use the **free pricing plan**)



* **Deploys code** from the <https://github.com/nakov/ContactBook> **repo**, providing the **Web app id**, the **URL of the repo** and the **main branch name**

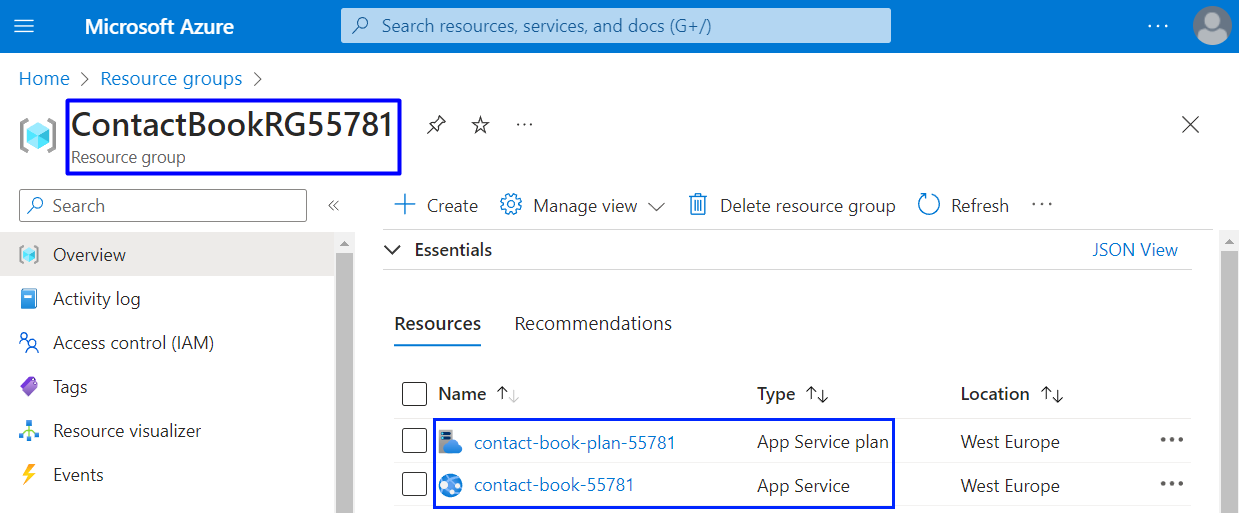


* + Moreover, we should set the use\_manual\_integration **argument** to true, so that we **agree to deploy the app and its updates manually** when we use an **external Git** (a public GitHub repo, which is not our own and we cannot run CI/CD in GitHub Actions)

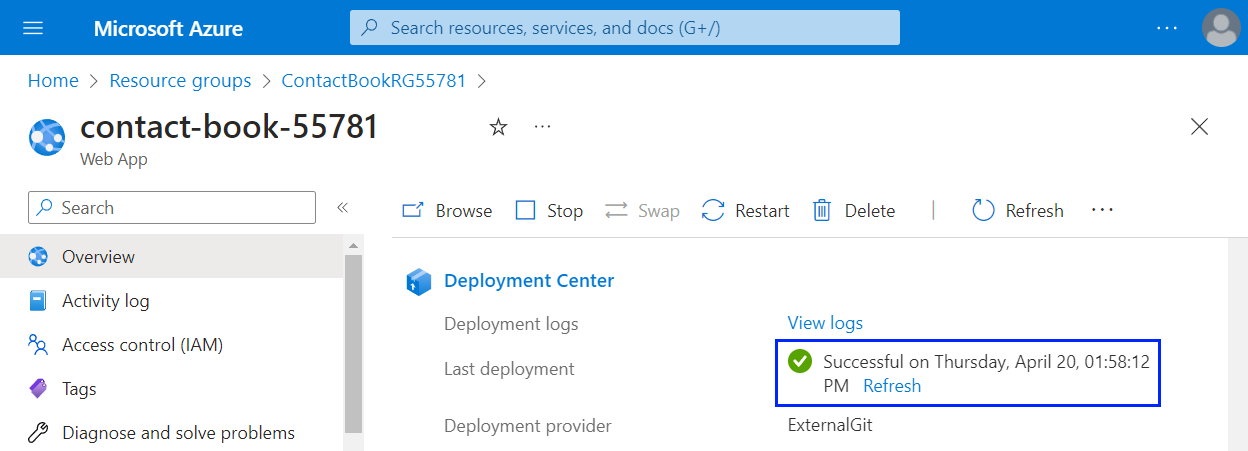
When **ready with the configuration file**, **initialize** Terraform, **format and validate the configuration** and **provision the resources** from the file. Know that this may **take a while**. It should be **successful** at the end:

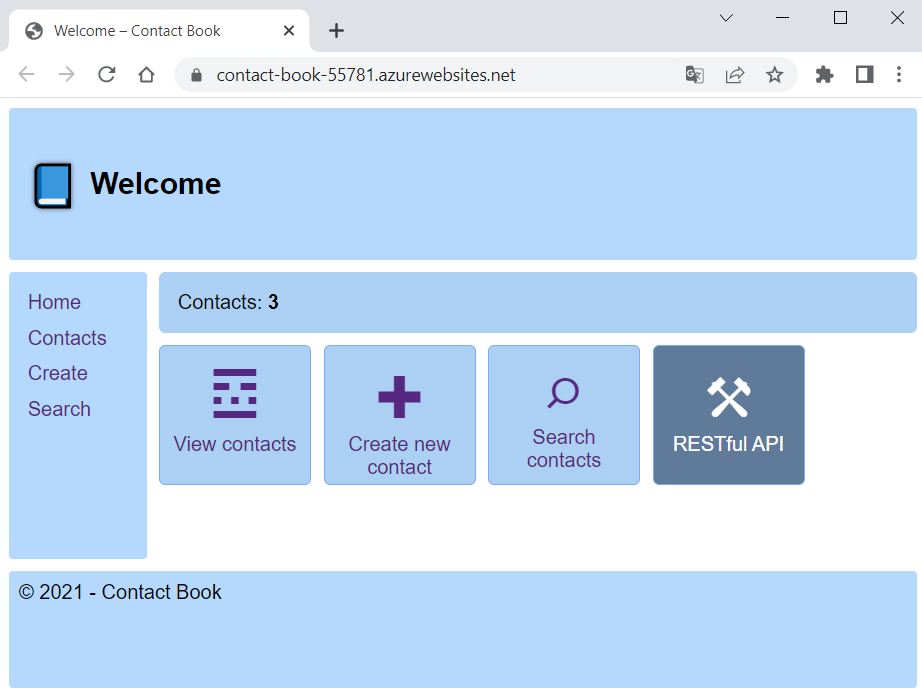


When **done**, make sure that you **have a resource group**, an **app service plan** and a **Web app** in Azure:



Also, make sure that the "Contact Book" **app** **is up and working** on the **provided domain URL** in Azure. First, however, you should **wait a bit** and make sure that the **deployment is successful**:



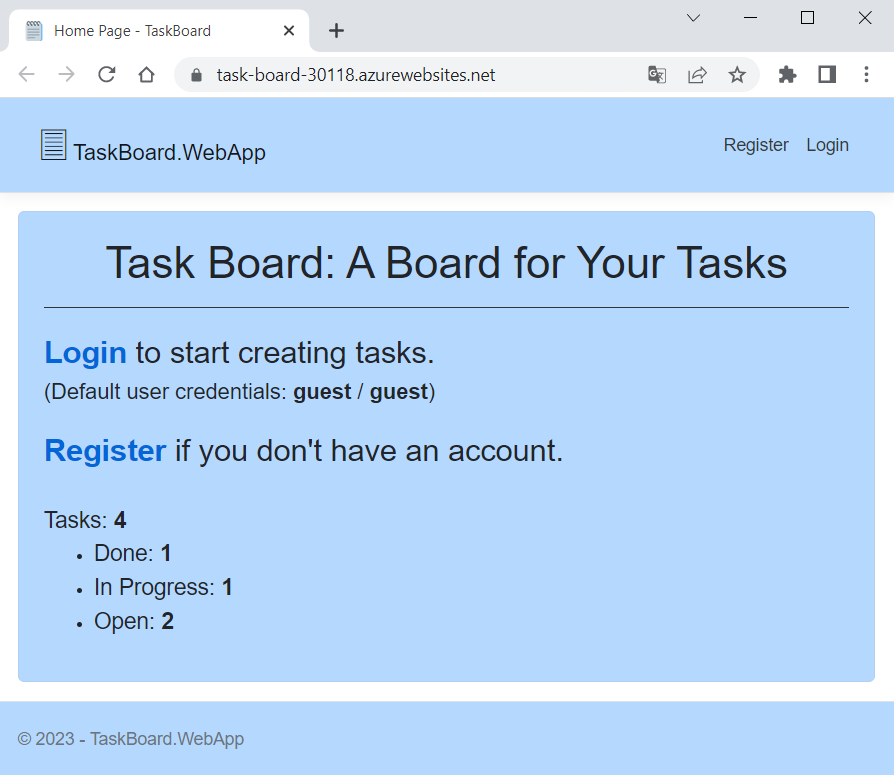
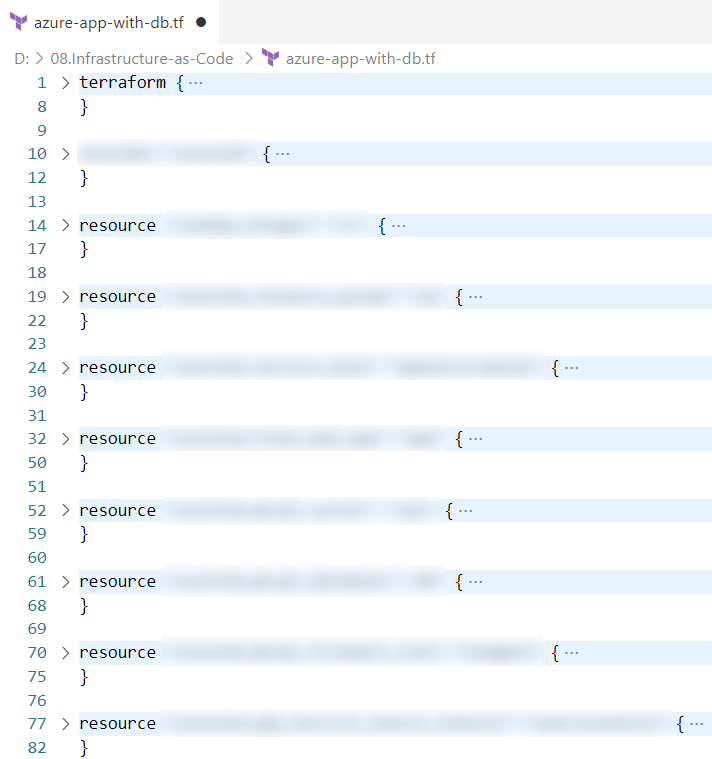


Finally, you can **destroy the created Azure resources** using the **well-known** Terraform **command**.

And this is how you can **deploy an app to** Azure with some easy steps, using Terraform.

### Azure Web App with Database

Create a Terraform **configuration** to **create and deploy the** "TaskBoard" **Web app** from the **resources** to Azure Web Apps. It is an **ASP.NET Core Web app** with a **SQL Server database**, which you should **upload to a GitHub repo** before you start.



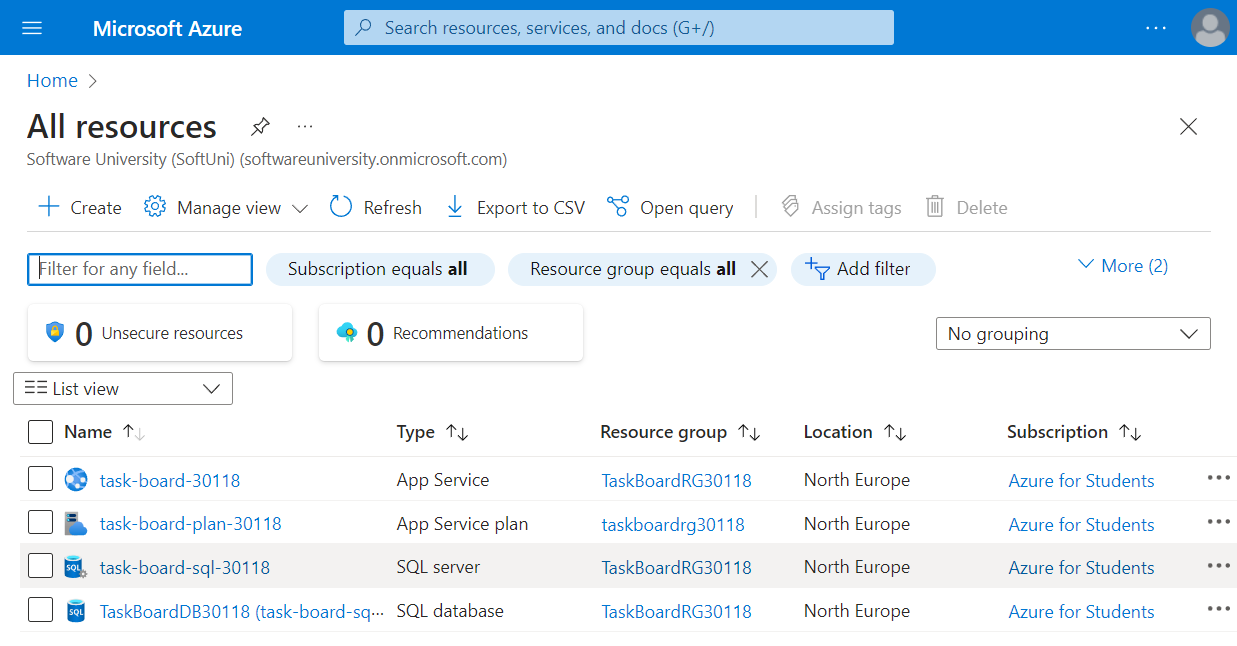
#### Write and Apply a Terraform Configuration

In this task, you can use the **Terraform configuration** **from the previous task** but you should make the following **modifications and additions**:

* Create a **server resource** **in Azure** with **name**, **resource group name**, **location**, **version**, **administrator** **username** and **administrator password** arguments
* Create a **database resource in Azure** with **name**, **server ID**, **collation**, **license type**, **SKU name** and **zone** **redundancy** arguments
* Create a **firewall rule for the Azure server**, which has a **name** and **server ID** and sets "0.0.0.0" as **start and end IP addresses** (this means that it allows other **Azure resources to access the server**)
* **Application stack** should be set to dotnet\_version = "6.0"
* The **Linux Web app** should contain a connection\_string **block** with:
  + Name: "DefaultConnection"
  + Type: "SQLAzure"
  + Value: "Data Source=tcp:${*fully qualified domain name of the MSSQL server*},1433;Initial Catalog=${*name of the SQL database*};User ID=${*username of the MSSQL server administrator*};Password=${*password of the MSSQL server administrator*};Trusted\_Connection=False; MultipleActiveResultSets=True;"
* The **GitHub repo URL** should be changed to point out a **repo with the source code** of the "TaskBoard" **app**

Find the **Terraform resources** you need and **how to configure them** by yourself. Also, use the **random integer** you have created as a resource to **generate unique names**, as well as **resource references** where possible.

When your **configuration is written**, use the well-known Terraform **commands** to **apply it**. After a while, your **declared resources should be provisioned** **in** Azure:



And then, when the **app is deployed from the GitHub repo**, your **app should be up and working**.

#### Separate Configuration to Multiple Files

What we should do now is **separate our Terraform configuration to multiple files**, as it is **good practice** that allows **configuration modularity**, **reusability**, etc.

When done, we will have the **following files** (not necessary with the same file names):

* main.tf – the main Terraform configuration file
* variables.tf – contains variable declarations
* values.tfvars – contains values for the variables
* outputs.tf – contains outputs declarations

Let's see how to **separate our configuration**.

##### Step 1: Define Input Variables

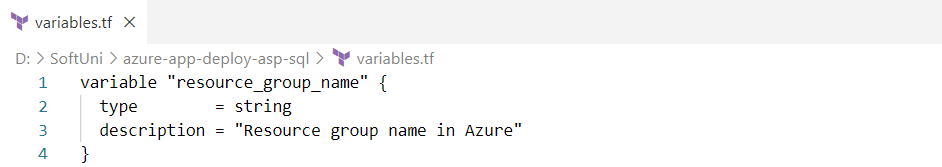
You have the **configuration for provisioning and deploying a Web app with database** but it is all in one .tf **file** – including resource names, administrator credentials, etc. There are quite a **few hard coded values** that would make sense to have as **input parameters instead**, as this would allow us to **re-use the same template** to create multiple environments with a slightly different configuration.

In our **configuration**, we have the following **values** that can be turned into **input parameters**:

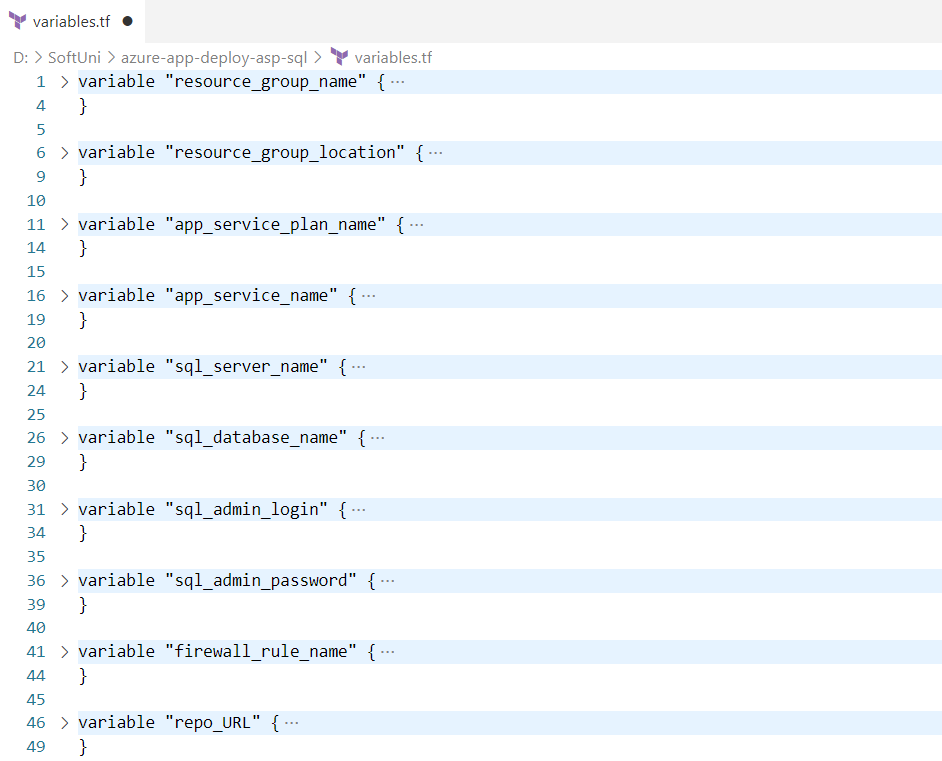
* Resource group name
* Resource group location
* App service plan name
* App service name
* SQL server name
* SQL database name
* SQL administrator login username
* SQL administrator password
* Firewall rule name
* GitHub repo URL

Create a **new** .tf **file** in the **Terraform configuration directory** and let's **define the input variables**. Each **variable** will have a **name**, **type** and **description**. In addition, it can have a **default value** that you can add if you want.

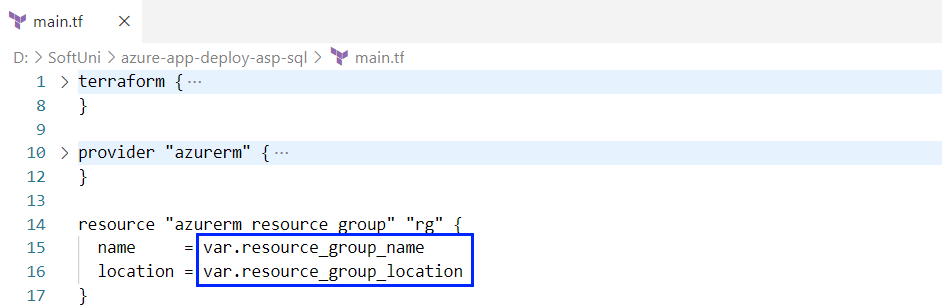
**Define each variable** from the above list in this way:



You can go on with the **rest of the variables' definition by yourself**, following the **syntax** shown. At the end, you should have **10 variables**:

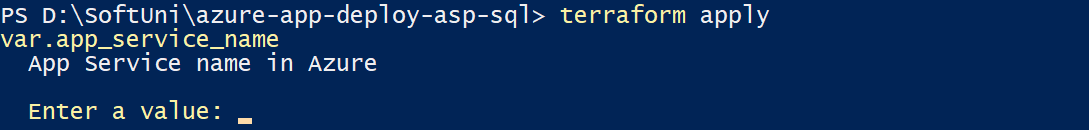


Now let's **use these variables** in the **main Terraform configuration file** we have. To do this, use the following **syntax**: var.{*variable name*}. Do it like this for **all input variables** you defined:

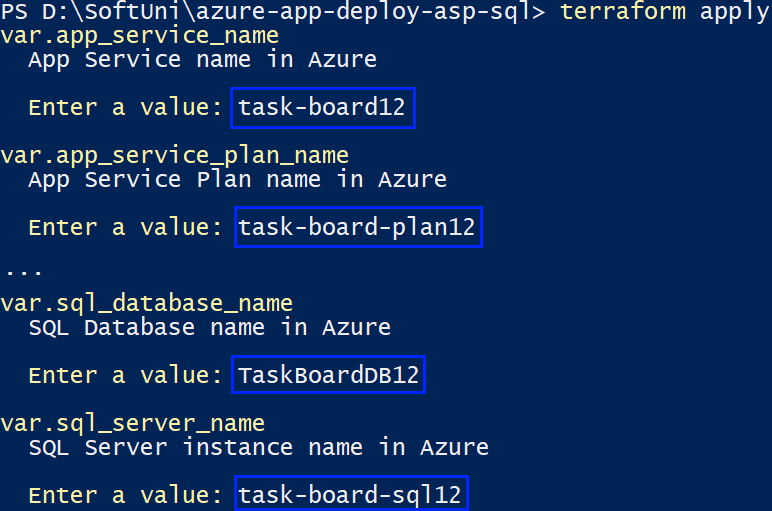
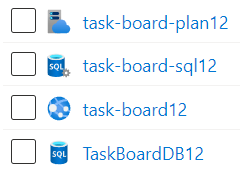


In addition, you can still use the **randomly generated integer value** as **part of the resource names** or you can **remove this resource** if you don't need it. However, make sure that your **resource names are unique enough** or **errors may appear**.

Now let's try to **apply the Terraform configuration** we have and see what will happen:



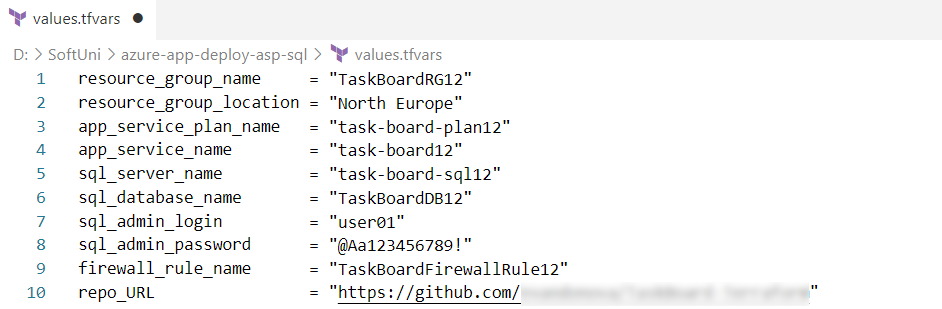
As you can see, you are **prompted to enter an app service name** for the app\_service\_name **input variable**. You should **add values for all variables** and then they will be **used in your configuration**. All of them are **required** as we didn't put default values.

 🡪 

Now we have **input variables for our configuration**, which is nice. However, if we run terraform destroy, we should **enter the same values again**, which is not pleasant.

##### Step 2: Create File with Variable Values

If we **don't want to enter values for the input variables**, we can **create a file** for them. Create a **file** with the .tfvars **extension** and **add value for each variable** using this syntax: {*variable* *name*} = "{*variable* *value*}".



Now we can **apply our configuration** again, using the .tfvars **file** we created:

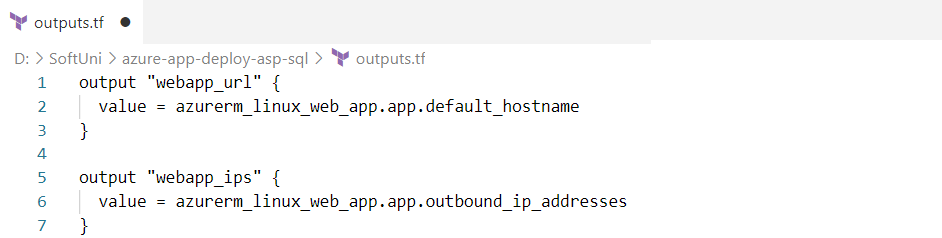


The **file should be found** and **values used** – you should **not be prompted** to add any value manually.

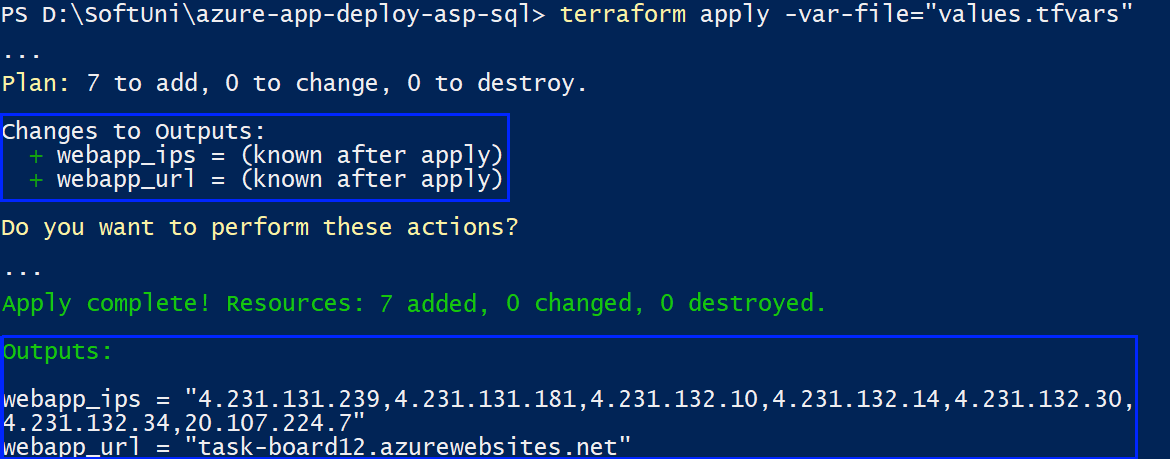
##### Step 3: Define Outputs

At the end, we can **add outputs** that will **print us the URL of the Azure Web app** that will be created and its **outbound IP addresses**. **Outputs** are basically just pieces **state information** that you want to have available for different purposes.

You should create **a new** .tf **file** and **define the outputs** with **name** and **value** using the following syntax:



When you **apply the configuration**, the **values of the outputs** should be **printed** **in the terminal**:



After all this separation of the **Terraform configuration** **to** **files**, it should still be **working** and **provision the resources in Azure** successfully.

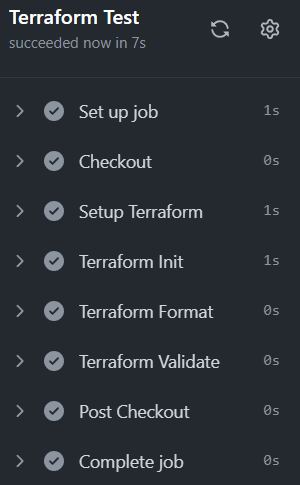
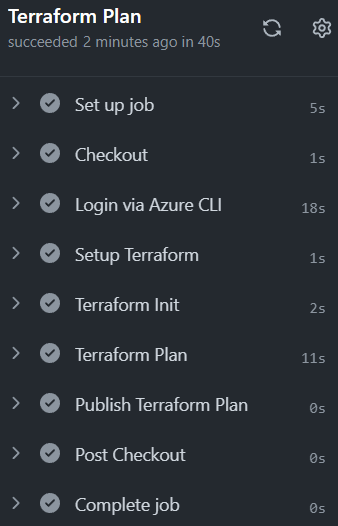
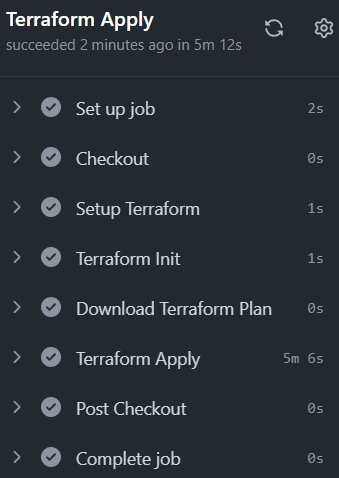
Now your **configuration follows good practices**. However, in the **next task** we will see how to **improve it** even more.

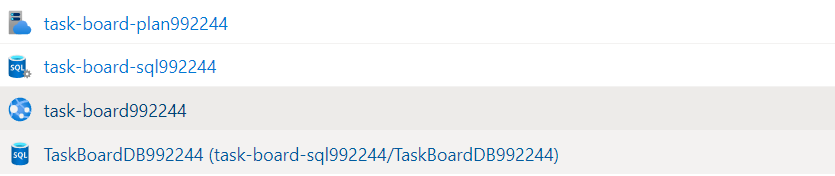
### Terraform with CI/CD

Now we will **upload the Terraform configuration from** the **previous task** (for provisioning **Azure resources** and **deploying the "**TaskBoard**" Web app** to Azure Web Apps) **to GitHub** and will use GitHub Actions **workflows** to **test and run the configuration**.

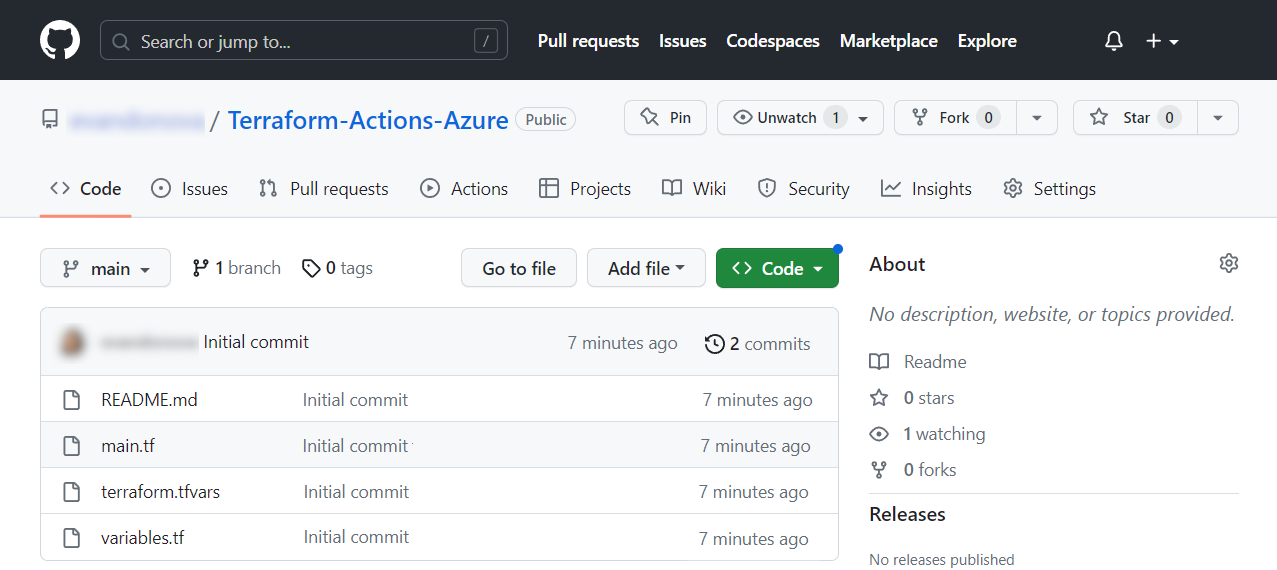
By combining **Terraform with GitHub Actions**, we can **automate the infrastructure provisioning process**, **ensure** **consistency**, and **integrate it into your CI/CD workflows**, promoting **efficient software delivery** and **reducing manual tasks**. It provides a streamlined and efficient **workflow** **for managing infrastructure as code**, making it easier to **maintain**, **test**, and **deploy your infrastructure** **resources**.

We will have **GitHub Actions workflows** that will provision the **Azure resources** we want:



Start by creating a **GitHub repository**, which should contain your main.tf **Terraform configuration file** and your additional **Terraform files** – terraform.tfvars and variables.tf:



**Note**: when the .tfvars **file** with **variable values** is **named** "terraform", **Terraform finds it on its own** and you **should not point to it specifically** in the **Terraform commands you run**.

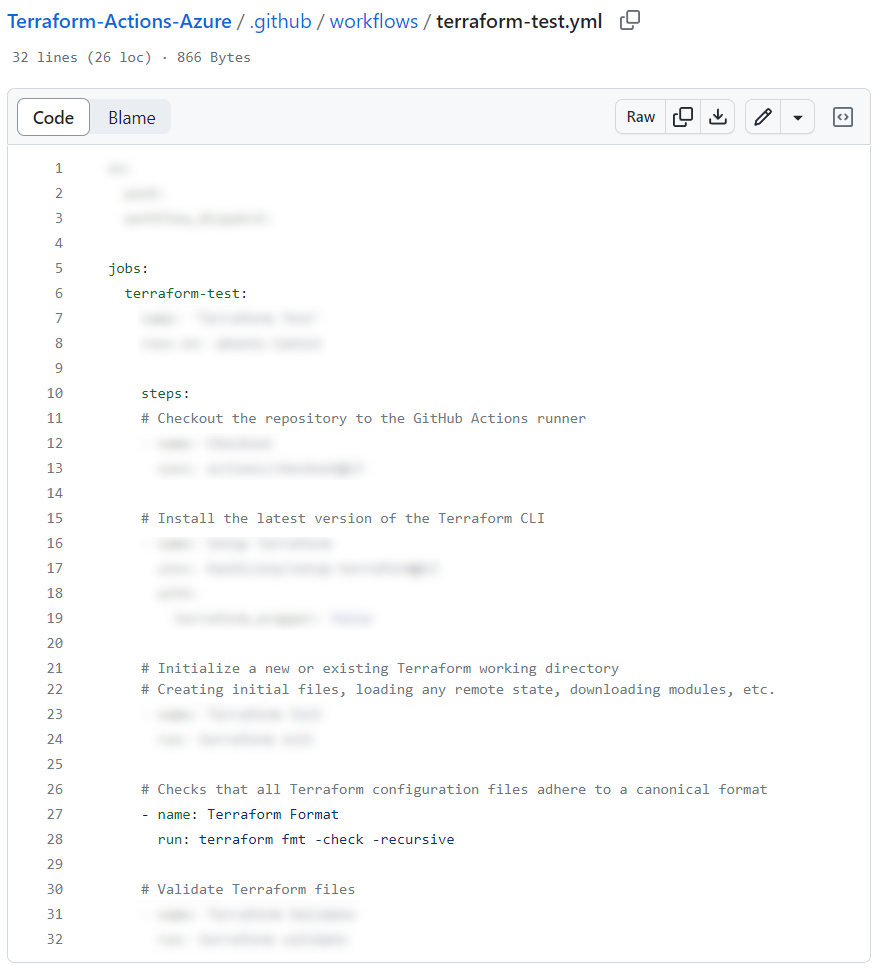
Also, you **don't need the** outputs.tf **file**, as you can use GitHub Actions to show you what you need when a workflow is run.

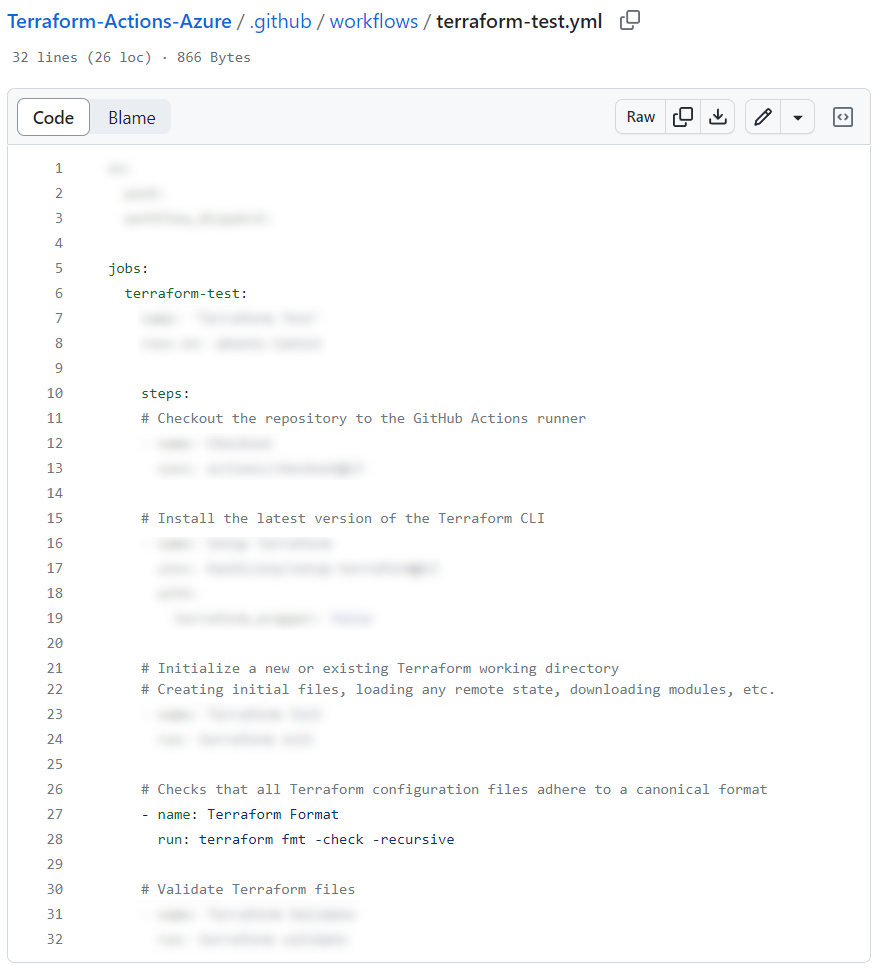
Now let's see how to write the **GitHub Actions workflows** we need.

#### Test Workflow

We will first write a **test workflow in GitHub Actions** that will try to **initialize the working directory**, **check if the configuration files are correctly formatted** and **validate the configuration**.

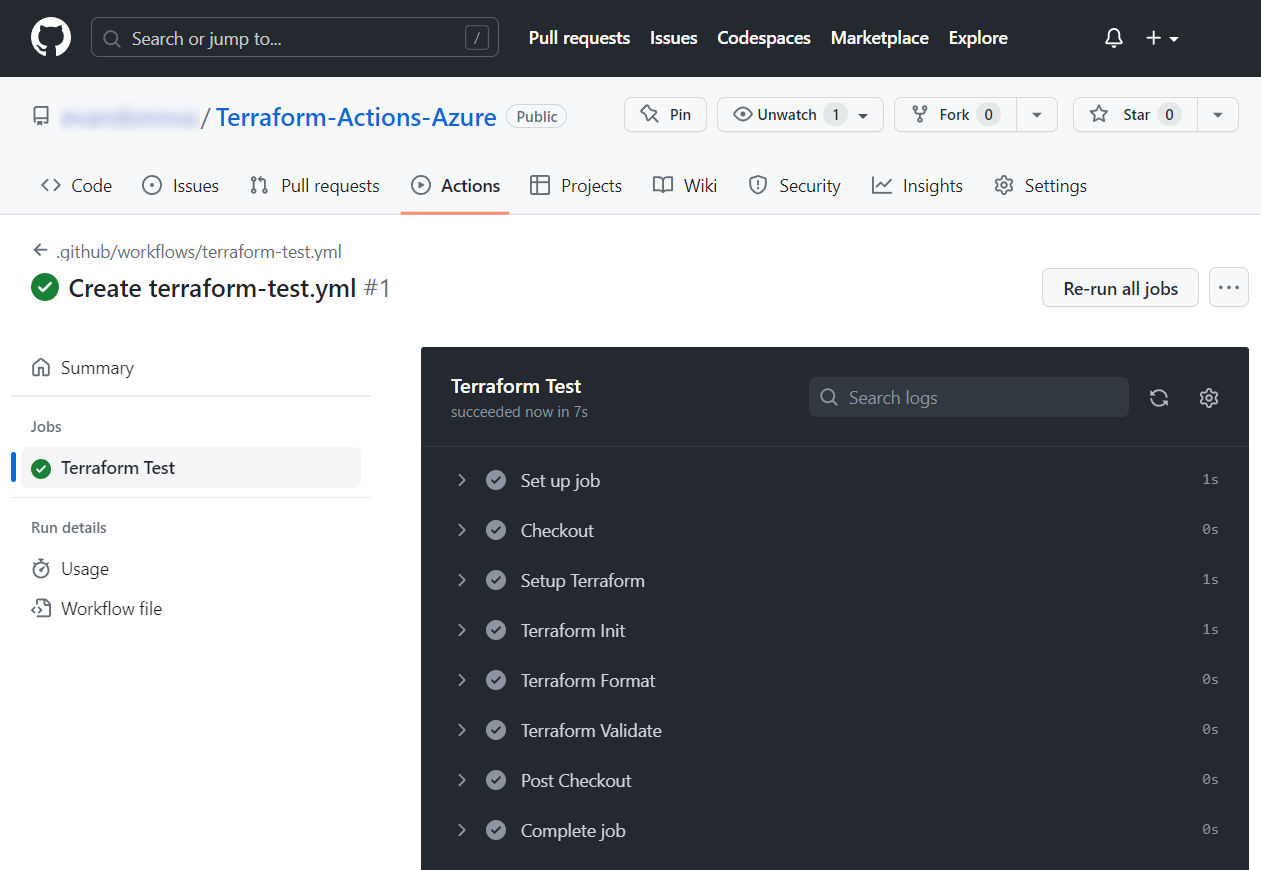
Create a YAML **file** in **GitHub Actions**. The **workflow** should look like this:





Look at the **comments in the above workflow** – they **describe the steps** for **testing the Terraform configuration**.

**Write the workflow** and **run it**. It should be **successful**:



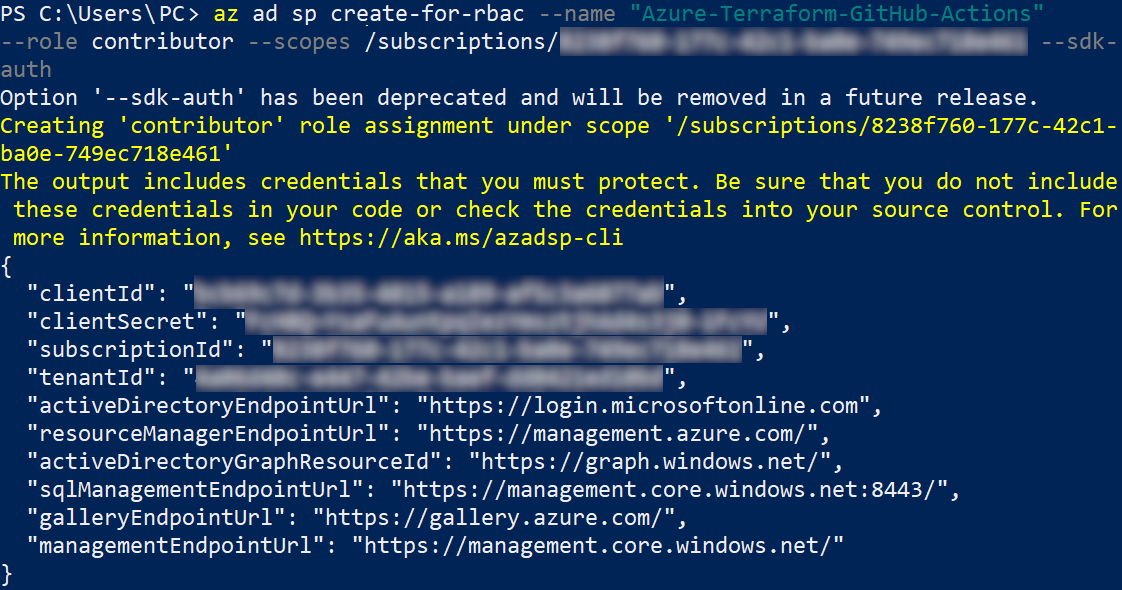
If you **receive any error**, **fix it** – you may have problems with your **Terraform configuration files** or the **workflow file** youhave just created.

#### Apply Configuration Workflow

When we have a **valid configuration** with **working tests** in **GitHub Actions**, let's use a **workflow to provision resources** and **deploy the "**TaskBoard**" Web app** to **Azure**. You should **authenticate in Azure** using a **service** **principal** and then **write the workflow**.

##### Step 1: Create Service Principal

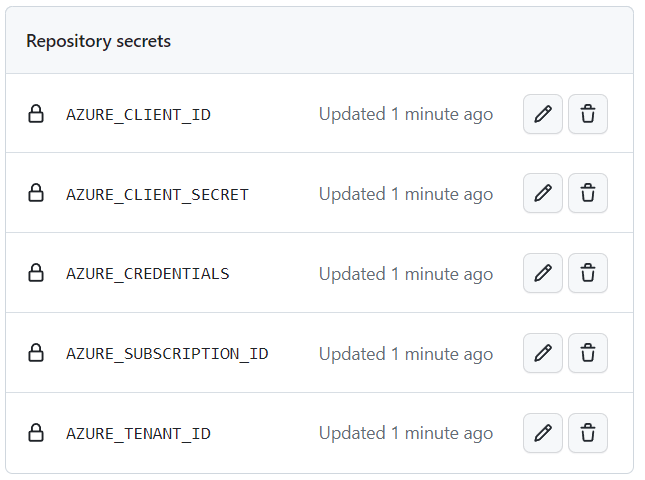
We should **create a service principal** with a "Contributor" **role** in **Azure** that we will use to **authenticate GitHub Actions**. Do it with the **following command locally** or **manually through Azure Portal**:



**Copy the credentials JSON** as you will need it for the next step.

##### Step 2: Create GitHub Secrets

As you know, it is **good practice** to **store your credentials** as **secrets in GitHub**. You need the following secrets:



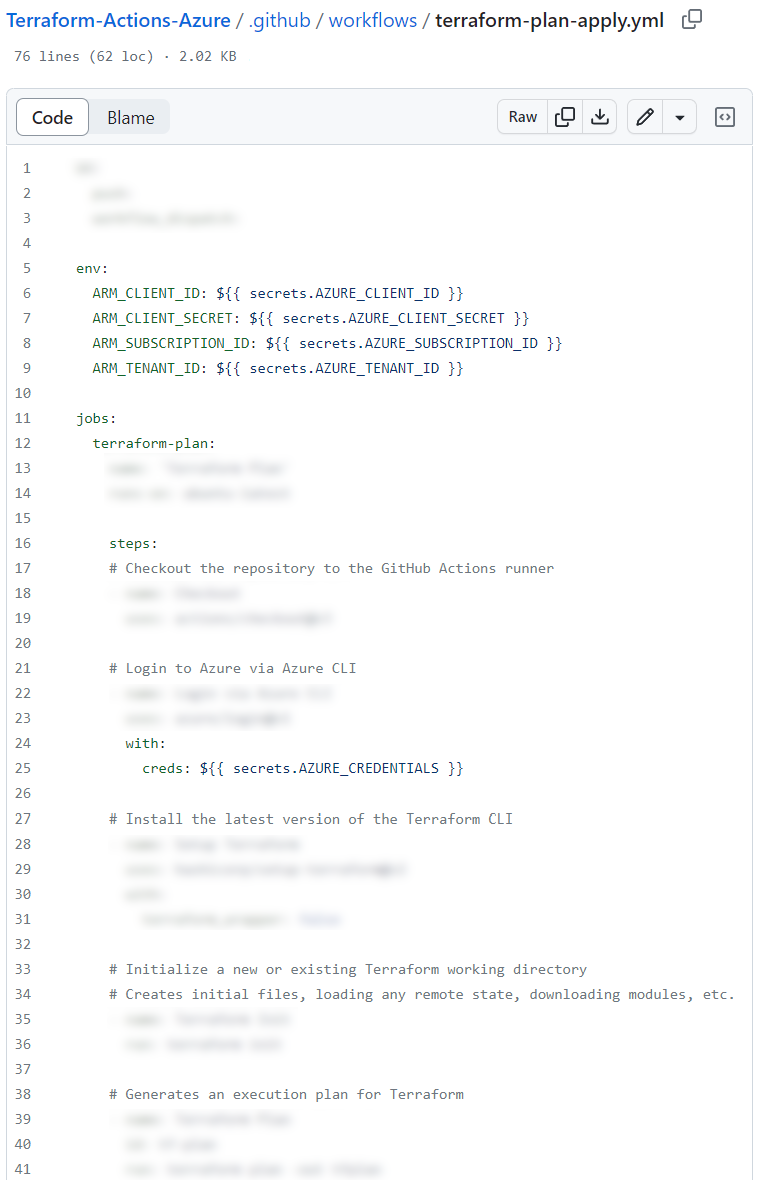
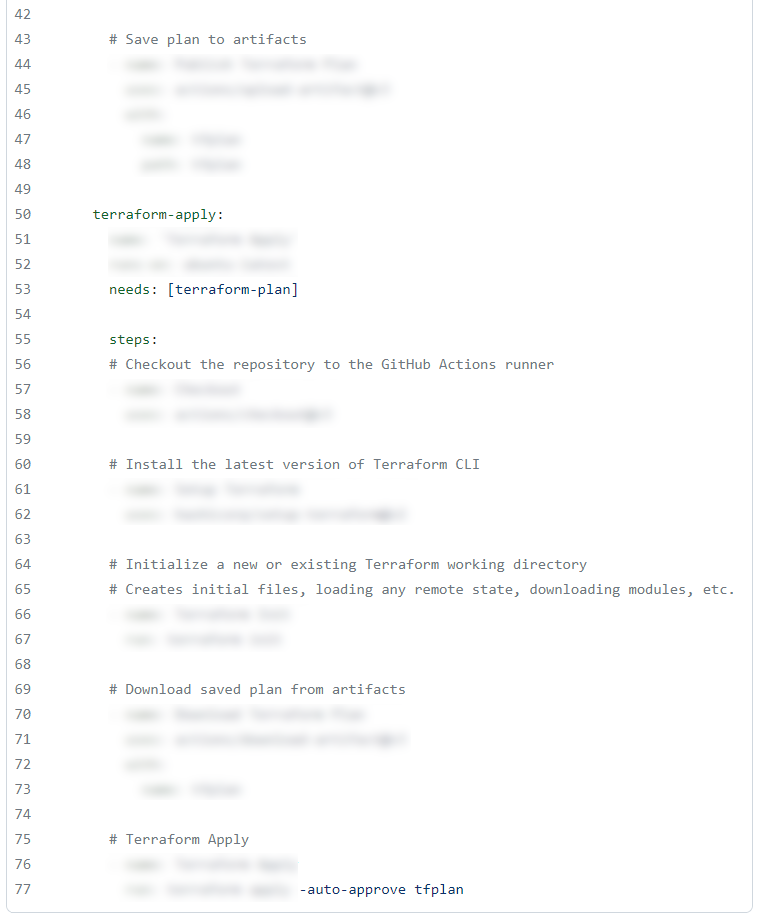
"AZURE\_CREDENTIALS" should **contain the whole JSON** **that we copied earlier** and the **rest of the variables** should contain **only the corresponding parts** of it (only the **value**, without quotes "").

Now we are ready to write the **GitHub workflow** that uses these secrets.

##### Step 3: Write the Workflow

Finally, let's **write the workflow** that will consist of **2 jobs** – the first one will **create the Terraform plan** and the **second one will apply it**.

**Write the workflow** in this way:

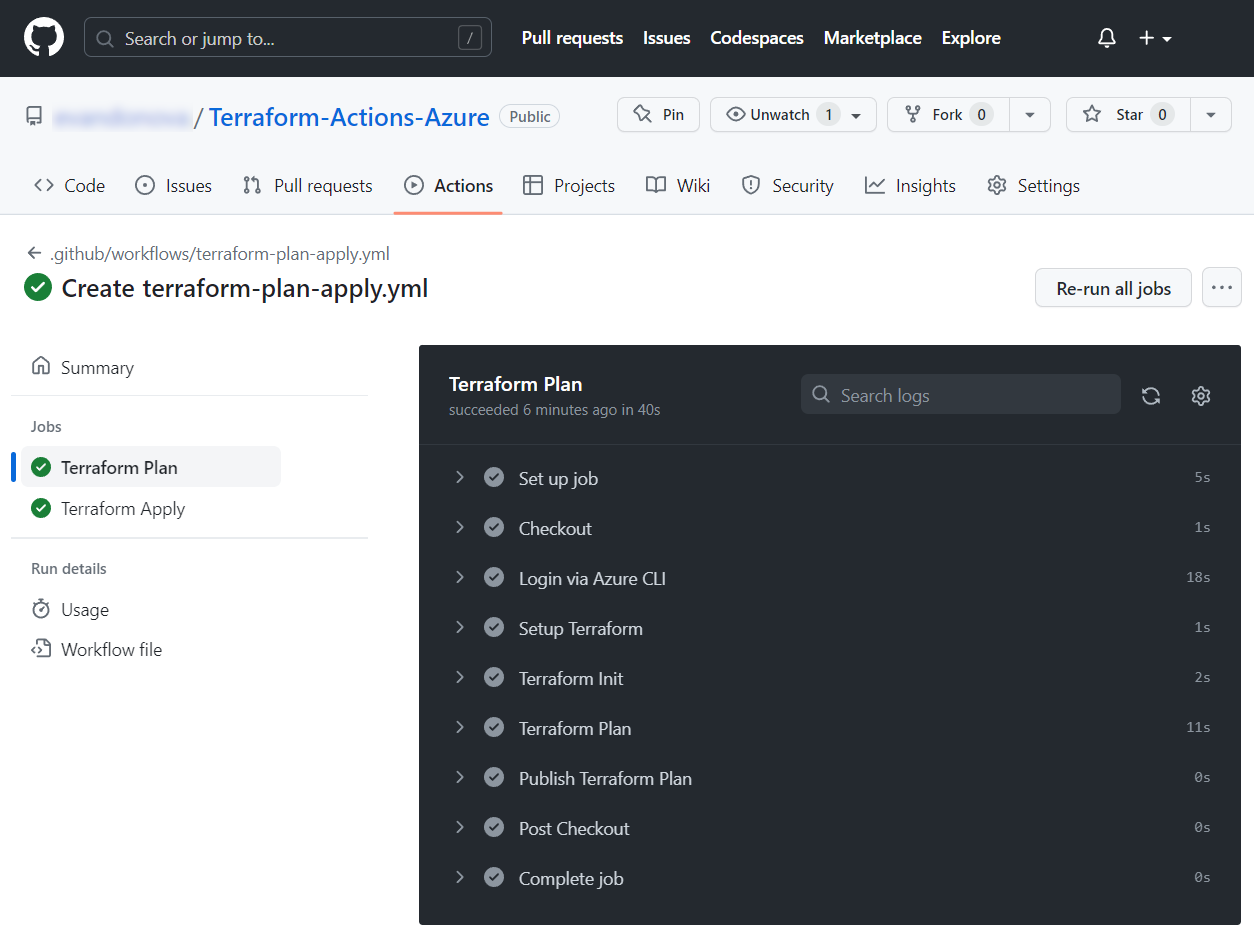
 

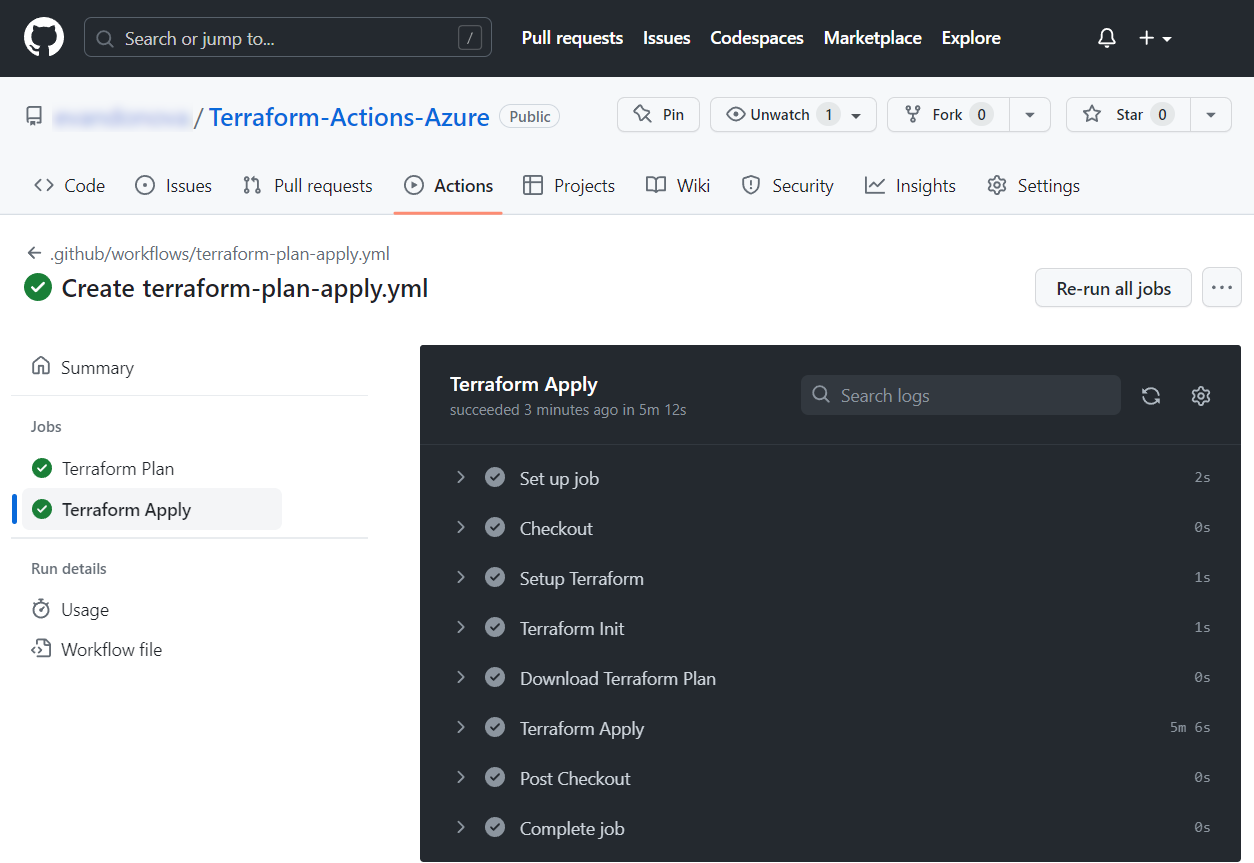
You can use the **steps from the test workflow** we created earlier as part of **this YAML file**.

Note some **specific things** about this **workflow**:

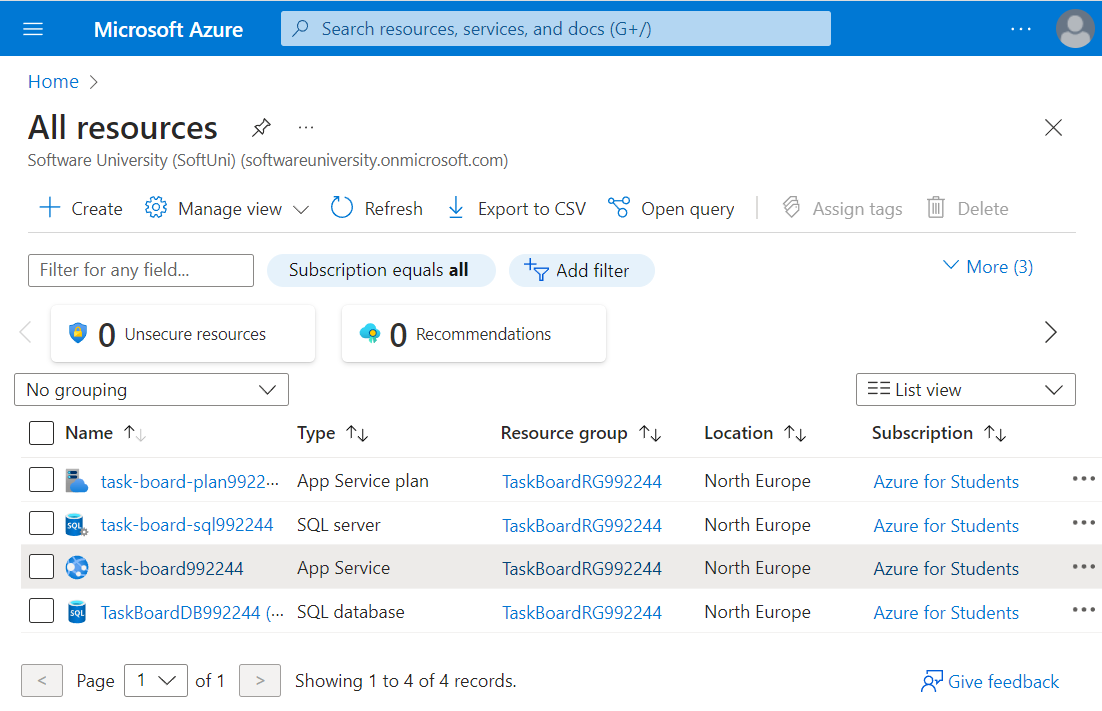
* You need some **environment variables** so that **Terraform** **can authenticate in Azure**.
* You should use the "AZURE\_CREDENTIALS" **GitHub secret** to **authenticate GitHub Actions in Azure**.
* The **second job** should **depend on the execution** of the **first one**.
* You should **add the** "-auto-approve tfplan" **flag** to **automatically approve the changes** in the "tfplan" without requiring manual confirmation during the workflow run.

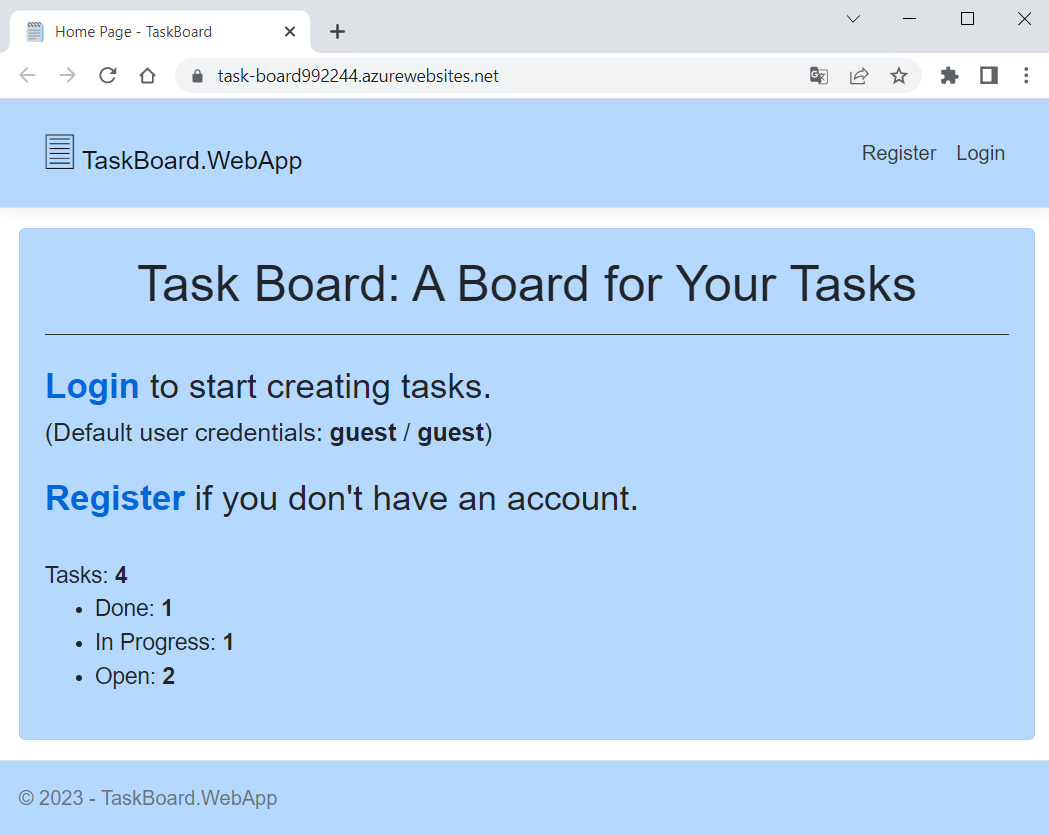
The **workflow** **should** **run successfully**:





Also, the **Azure resources** you defined in the **Terraform configuration** should be **provisioned** and the "TaskBoard" **app deployed and working**:



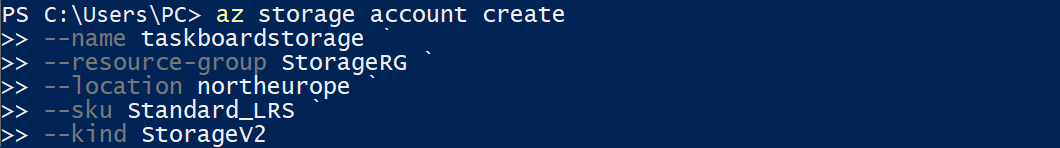
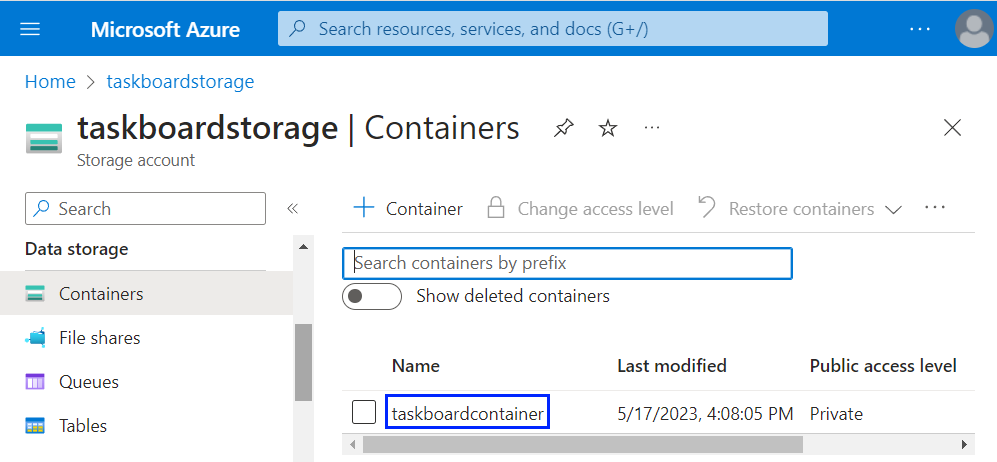


We successfully used **GitHub Actions** to **run a Terraform configuration** that **provisions resources in Azure**. However, if we **change the configuration** and **run the workflow again**, an **error will occur**. This happens because we **don't save the Terraform configuration state file**.

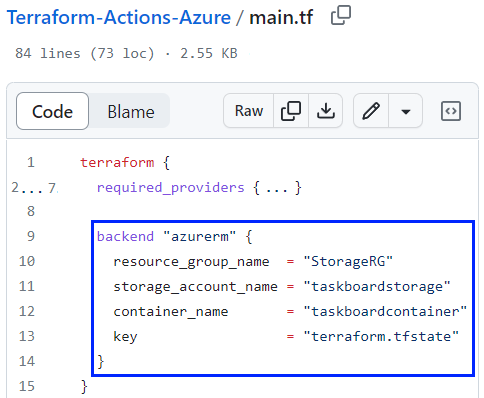
#### Store State File in Azure Storage Account

**Terraform** utilizes a **state file** to **store information** about the **current state of your managed infrastructure** and associated configuration. This file will need to be **persisted between different runs of the workflow**.

The recommended approach is to **store this file** within an AzureStorageAccount and this is what we will do now. First, you should **create an Azure storage account** with a **container** to **store the state file**:

Then, to **use this storage** **in** **Terraform**, you should **add a** backend **block** in the main.tf **configuration file**:

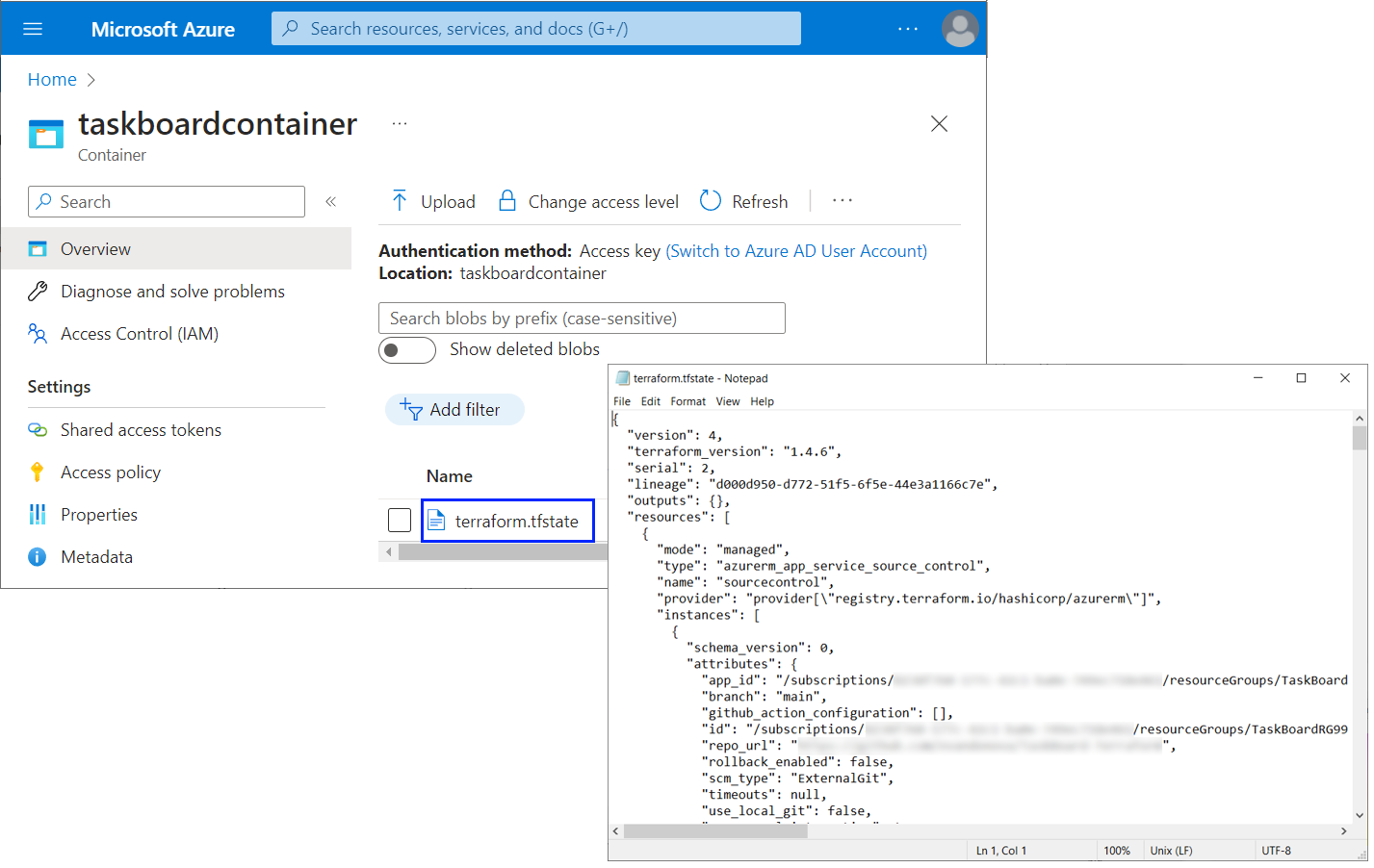


A backend **block** defines where **Terraform stores its state data files**. You should provide the **names of your resource group**, **storage account** and **container**, as well as to set a **name of the state file** that will be created.

**Commit the changed file** **to GitHub** and wait for **GitHub Actions to run the workflow**.

**Note**: you **should not have your resources in Azure now** or the **GitHub Actions workflow will still give you an error** when **trying to create** **them**, as they are **not defined in the state file**. **Delete the resources** you created previously with your Terraform configuration from **Azure**.

The **workflow should be** **successful** and you should see that a terraform.tfstate **file** was created in your **Azure storage container**:



Go and **make a change in your Terraform configuration** **in GitHub** and **run the workflow again** – the **modified resources** should be **updated successfully** **in Azure**.

Now you have a **fully working** GitHub Actions **+** Terraform **+** Azure **configuration** to create and manage resources.

#### \* More Configuration Improvements

We have a **good Terraform configuration** and **GitHub Actions workflows** created during the previous tasks but here are some **additional challenges for you** to overcome to **improve your Terraform skills** even more:

* You can **create a Terraform configuration file** to **provision an Azure storage account** **and container** for the **Terraform backend**, instead of doing it with commands like we did previously. Then, you can use a **GitHub Actions workflow** to **run that configuration** and **provision the resources in** **Azure**.
* You can **create a Terraform configuration file** to **create the service principal** and **assign the** "Contributor" **role to it** instead of doing it manually or with commands through Azure CLI. You can again try to **run the configuration in GitHub Actions**, not only locally.

By **fulfilling these additional tasks**, you would have **fully explored** and used the **integration between** Terraform, GitHubActions and Azure.