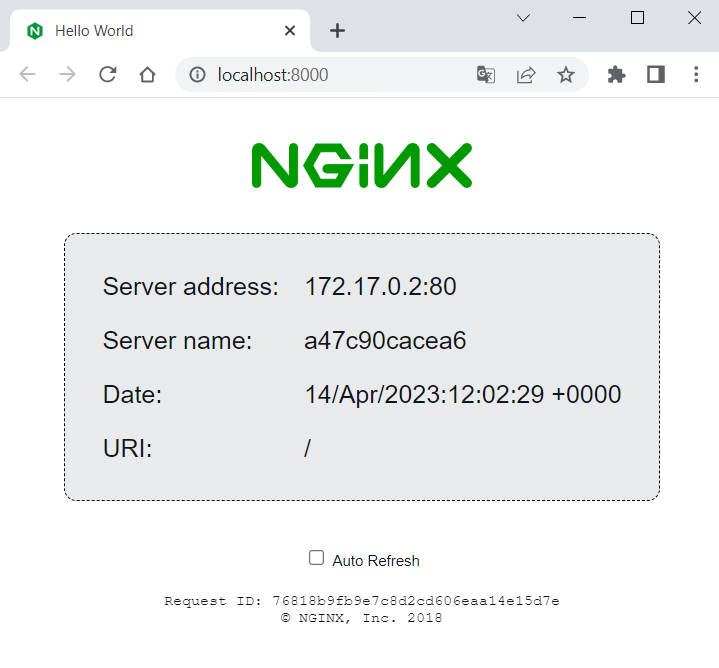
# Lab: Infrastructure as Code

Lab for the ["Containers and Clouds"](https://softuni.bg/trainings/4332/containers-and-cloud-september-2023) course @ SoftUni

## NGINX Docker Container

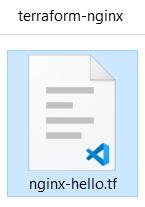
It is time to **create our first infrastructure**. We will provision an NGINX Docker **container** using Terraform.



For this, you should have **Docker running**.

### Step 1: Write the Configuration

First, **create a directory** that will keep your Terraform **configuration files**. Do it in a way you like – using File Explorer or a **terminal**. Then, **create a file** with the .tf **extension** in the **folder**, where we will **define the infrastructure** that we want:



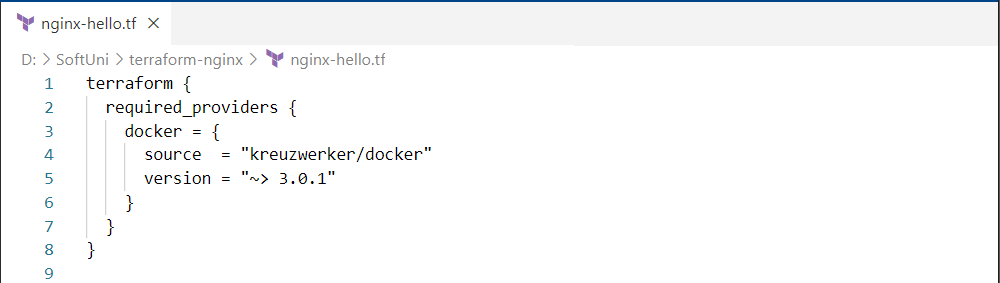
**Open the file** **in an** **editor** of your choice and **let's write the configuration** for the NGINX **container**.

Start by adding a terraform {} **block**, which **contains** Terraform **settings**, including the **required providers** Terraform will use to **provision your infrastructure**.



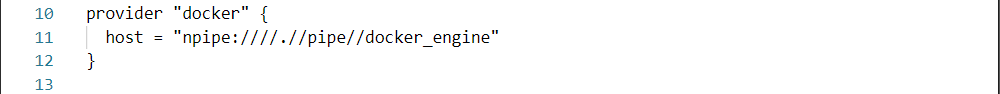
Each Terraform **module** must **declare which providers it requires**, so that Terraform **can** **install** **and use them**. **Provider requirements** are declared in a required\_providers {} block in the terraform one and consist of a **local name**, a **source location**, and a **version constraint**.

In our case, we need Docker, so we should **install a provider** **for it**. **Providers are installed** from the [Terraform Registry](https://registry.terraform.io/) by default and the one we need is called "kreuzwerker/docker". (<https://registry.terraform.io/providers/kreuzwerker/docker/latest>). Use it like this:



**Provider's** local name is its **unique identifier** **within this module** (in our case "docker"), the source defines the **global source address for the provider** you intend to use and the version **constraint** **specifies which subset of available provider versions the module is compatible with**.

Next, the provider {} **block** **configures the specified provider**, in this case "docker". **Configure Docker** to **connect to Docker daemon** and **interact with Docker containers** **and images** on a Windows **host** using a **named pipe**: "npipe:////./pipe/docker\_engine":



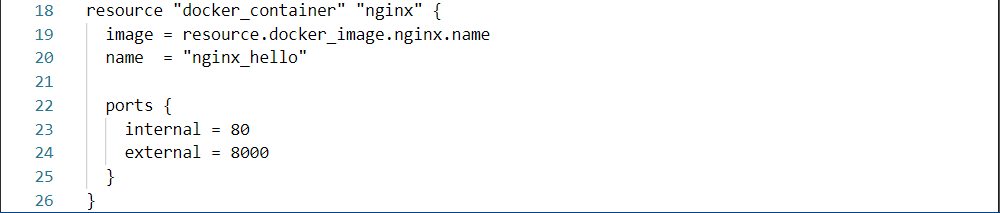
Now we need to **pull the** "nginxdemos/hello" **image** from Docker Hub to later create a container with it. To do this, you need to **create a** Terraform **resource**.

**Resource blocks** are used to **define components of your infrastructure**. They have **two strings before the block**: the **resource** **type** and the **resource name**. Together, the resource type and resource name form a **unique ID for the resource**.

Use the "docker\_image" Terraform **resource** and "nginx" **for a resource name**. Also, **provide the name of the** DockerHub **image** you want to use:



At last, **create a resource for the Docker container**, using the **image** **we defined as a resource** (with its **unique resource ID** as **image name**). Also, you should give a **name to the container** and **map ports**:

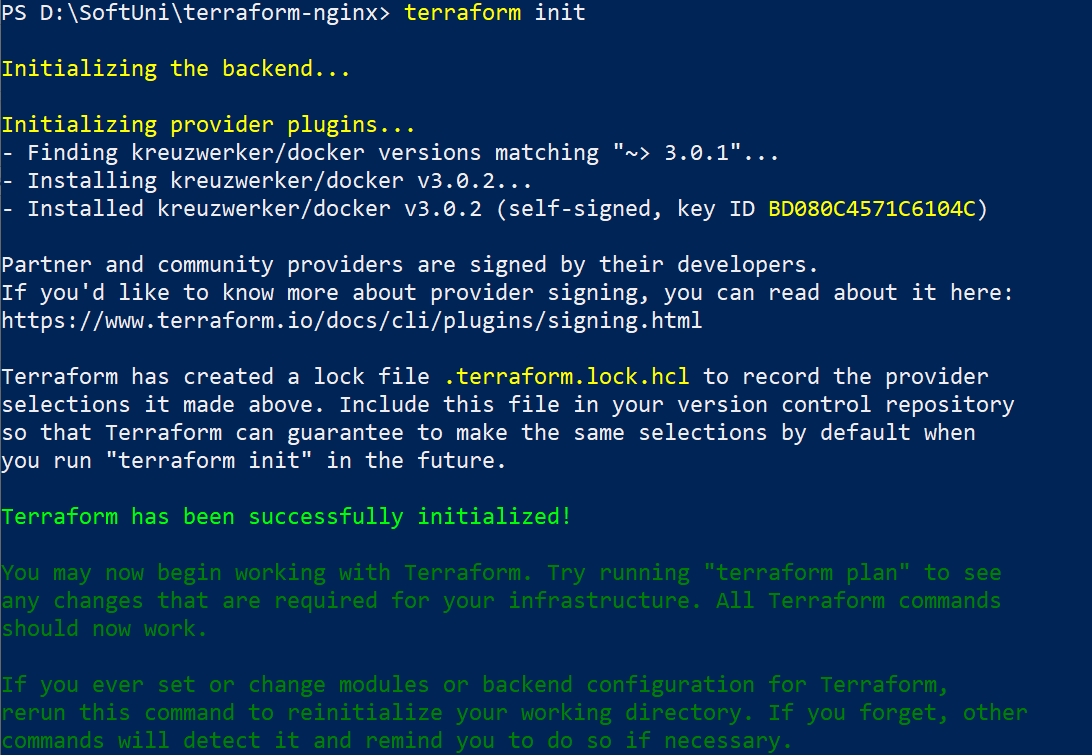


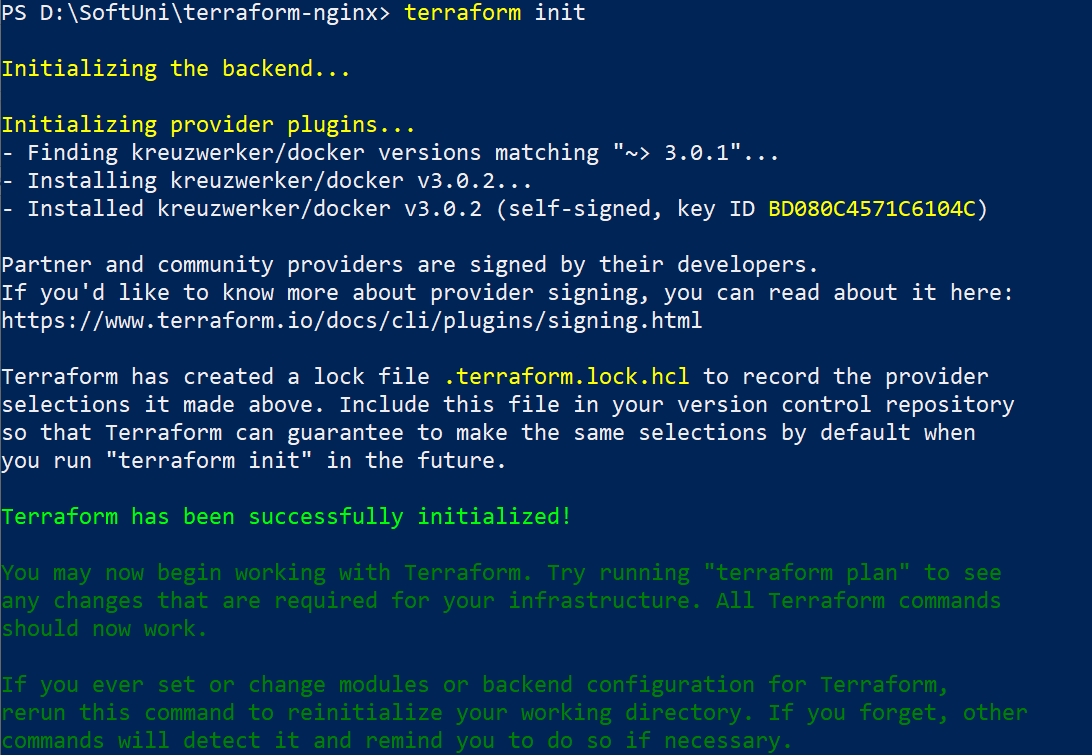
As you can see, the **resource blocks can also contain arguments** which you use to **configure the resource**.

Now your Terraform **configuration is ready**. **Save the file** and see how to use it in the next steps.

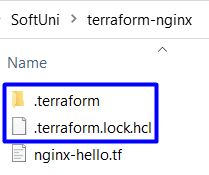
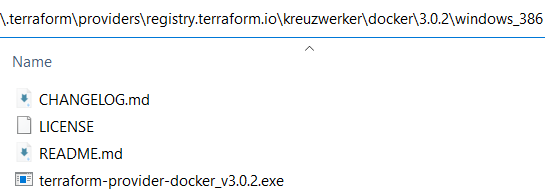
### Step 2: Initialize the Directory

When you have a **configuration**, you need to **initialize a configuration directory**, which **downloads and installs the providers** defined in the configuration. Do it with the terraform init **command** in the Terraform **configuration directory**:





Terraform **downloads the** docker **provider** and **installs it in a hidden subdirectory** of your current working directory, named .terraform. It also **creates a lock file** named .terraform.lock.hcl which **specifies the exact provider versions used**, so that you can control when you want to update the providers used for your project:

Then, before we **create the container**, let’s **format and validate the configuration** we have written.

### Step 3: Format and Validate the Configuration

The terraform fmt **command** **automatically updates configurations** in the current directory for readability and consistency:



Terraform **will** **print out the names of the files it modified**, if any. In our case, our configuration was formatted well.

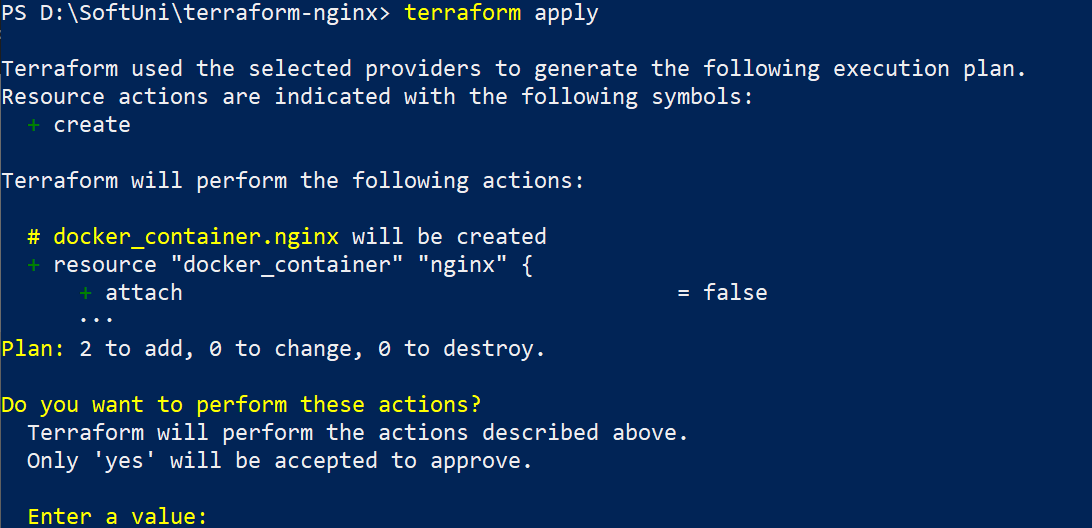
You can also make sure your **configuration is syntactically valid and internally consistent** by using the terraform validate command:

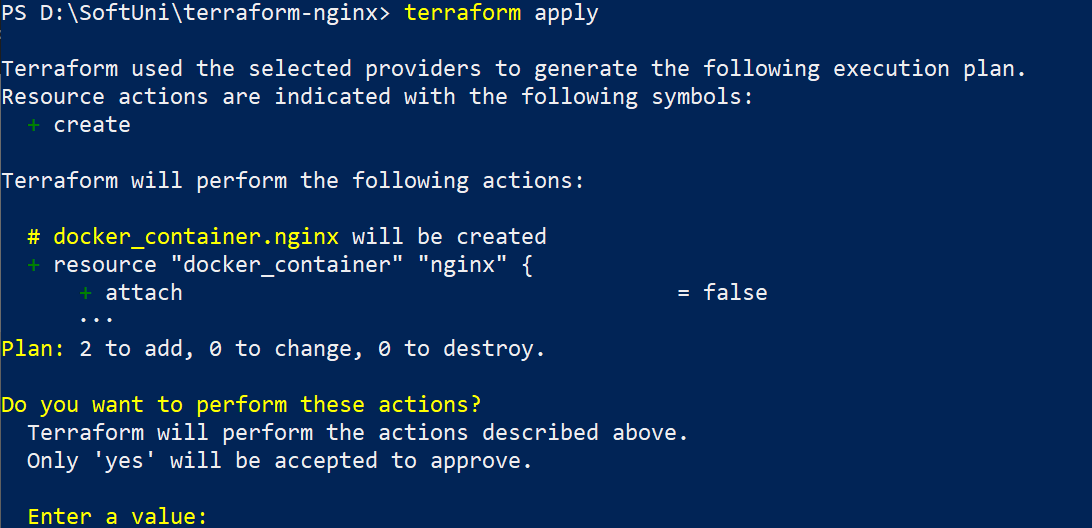


Validation was successful and our **configuration is valid**. Now it is time to apply it.

### Step 4: Create Infrastructure

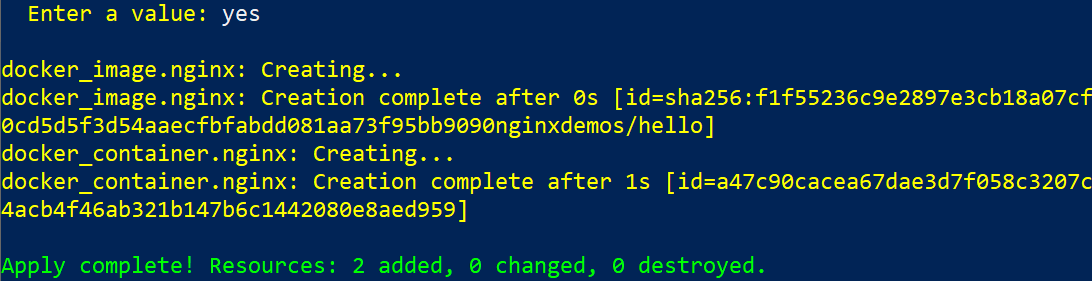
Use the terraform apply command to **execute the configuration file and apply it** to **create the wanted infrastructure**. Terraform will **print a similar output**:





Before it applies any changes, Terraform **prints out the execution plan** which **describes the actions Terraform will take** in order to **change your infrastructure to match the configuration**. Terraform will now pause and **wait for your approval** before proceeding.

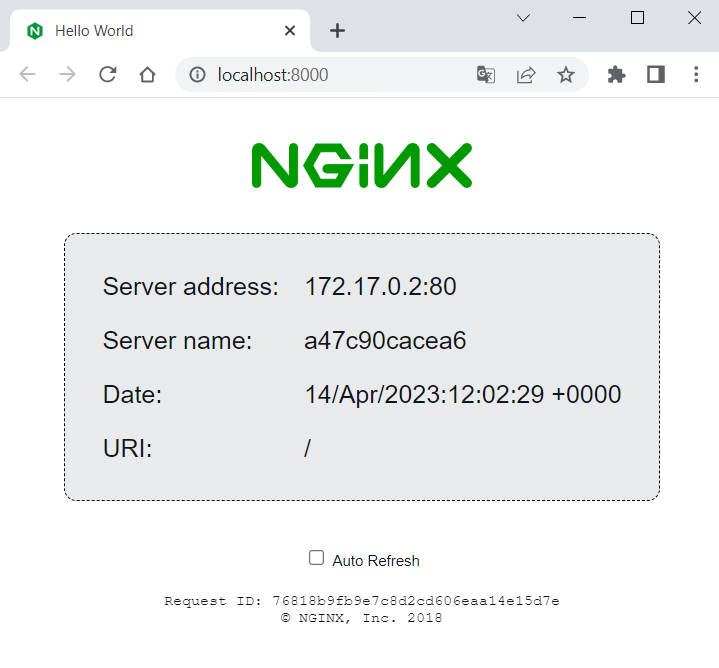
In this case the **plan is acceptable**, so **type** "yes" at the **confirmation prompt to proceed**:



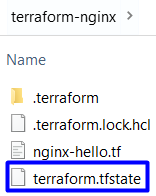
Terraform **has created the infrastructure you want**, which includes **2 resources** – the **container image** and the **container** itself. You can **look at them** in Docker Desktop:



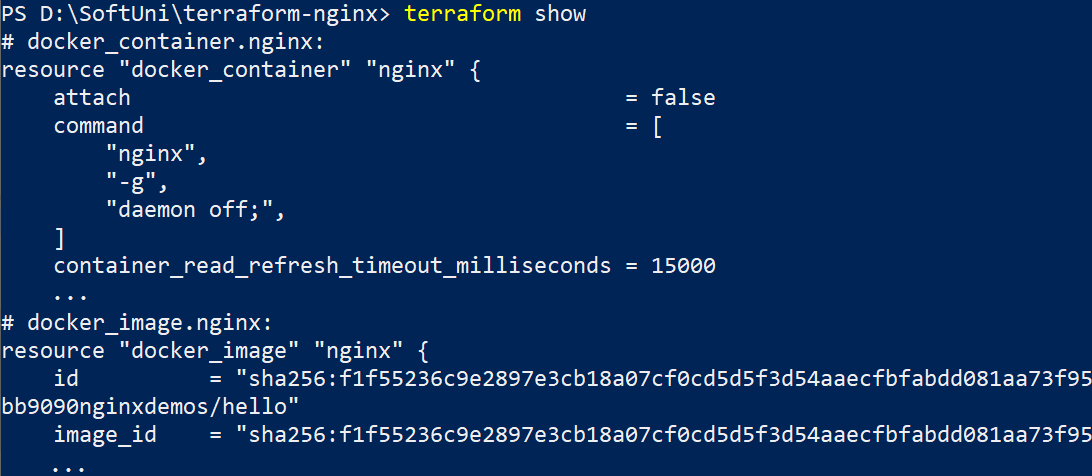
You can also **visit** <http://localhost:8080/> to **validate that the NGINX container** **is started and working**:



You can also **inspect the state of the resources that** Terraform **manages**, as it wrote data into the terraform.tfstate **state file** when you **applied your configuration**:



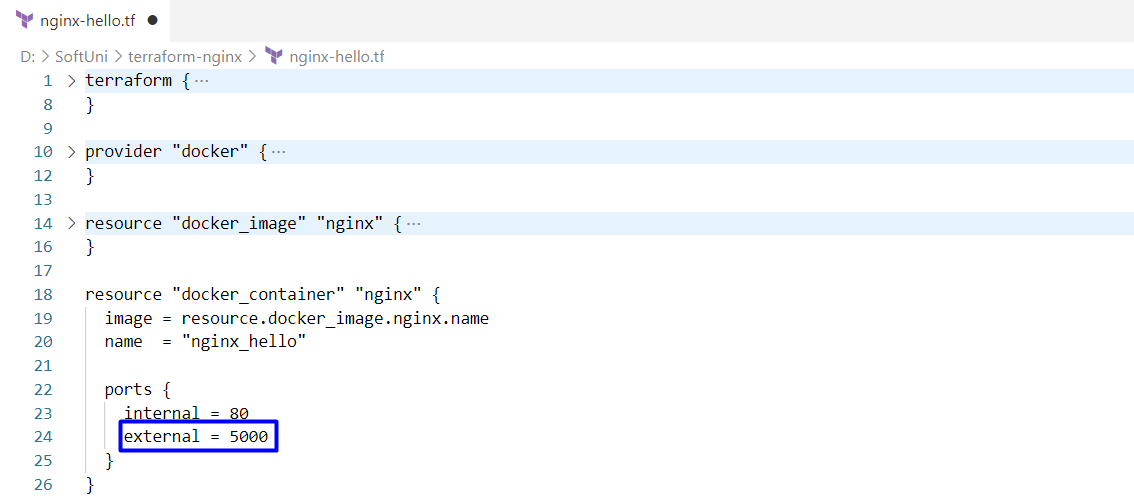
**Inspect the current state** using the terraform show **command**:



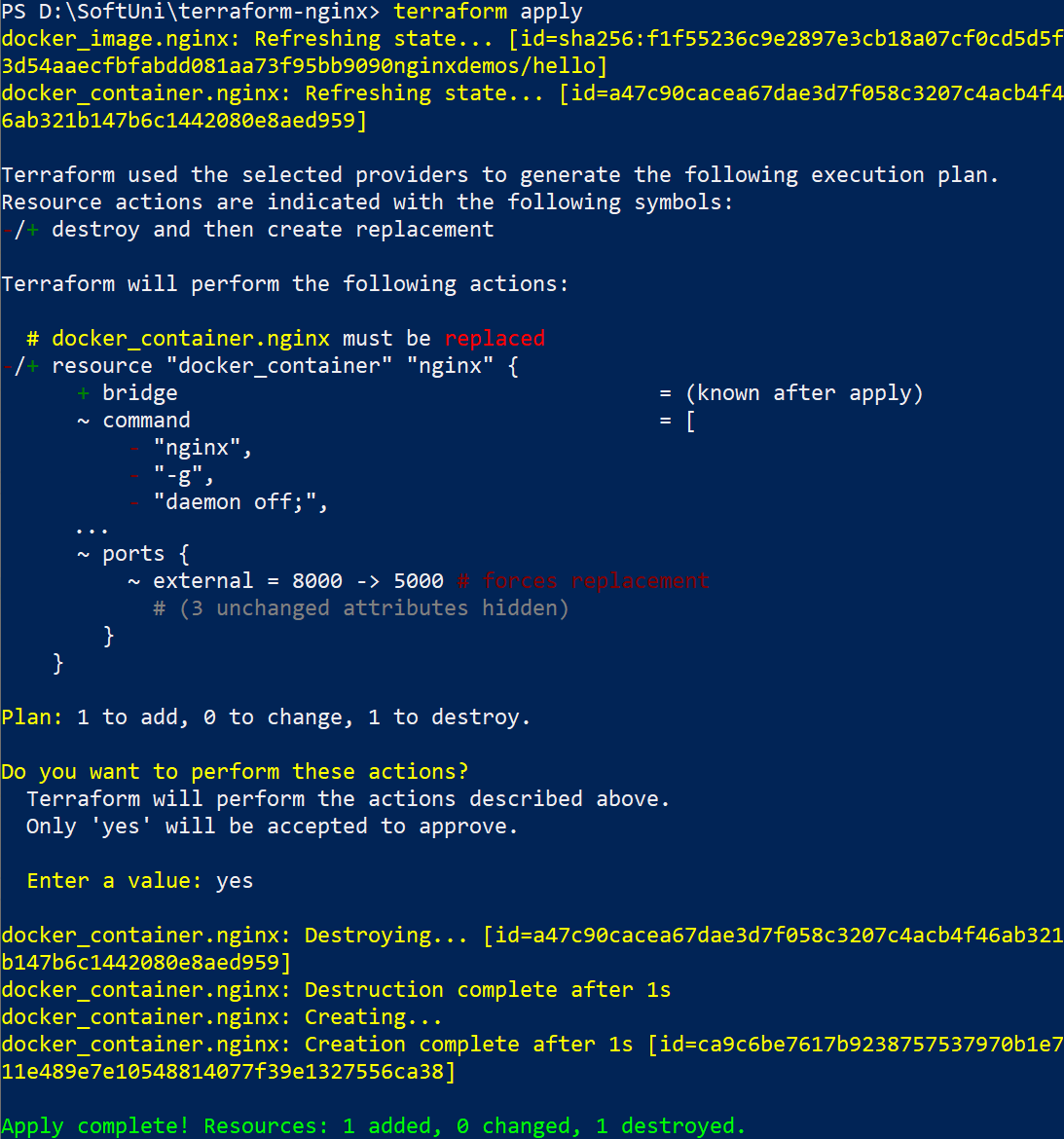
You have your Terraform **configuration and resources**. Now let's see how to **modify them**.

### Step 5: Update Configuration

To **change your configuration**, make changes to the .tf **configuration file**. For the example, let's **change the external port** of the container **from** 8000 **to** 5000:



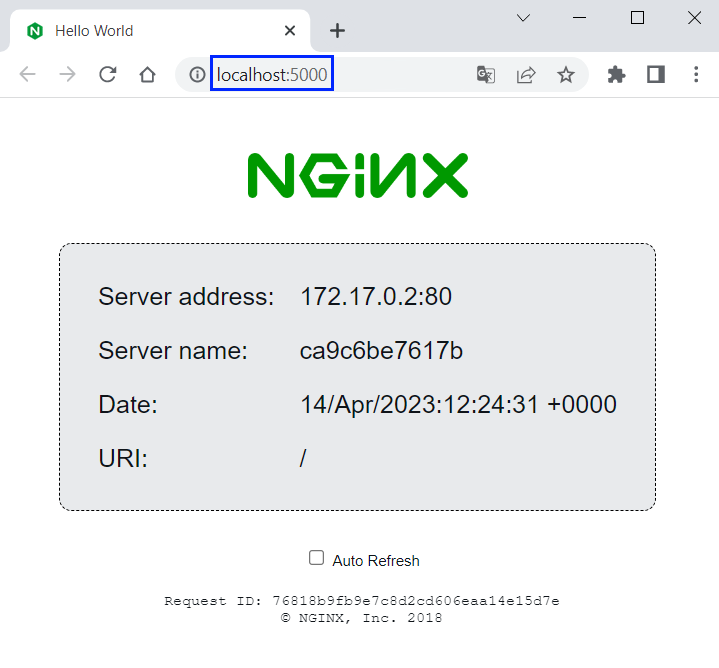
**Save the file** and **apply the changed configuration**:



The Docker **provider** knows that it **cannot change the port of a container** after it has been created, so Terraform **destroys the old container** and **creates a new one**.

And you **have your new container**, accessible on <http://localhost:5000/>:



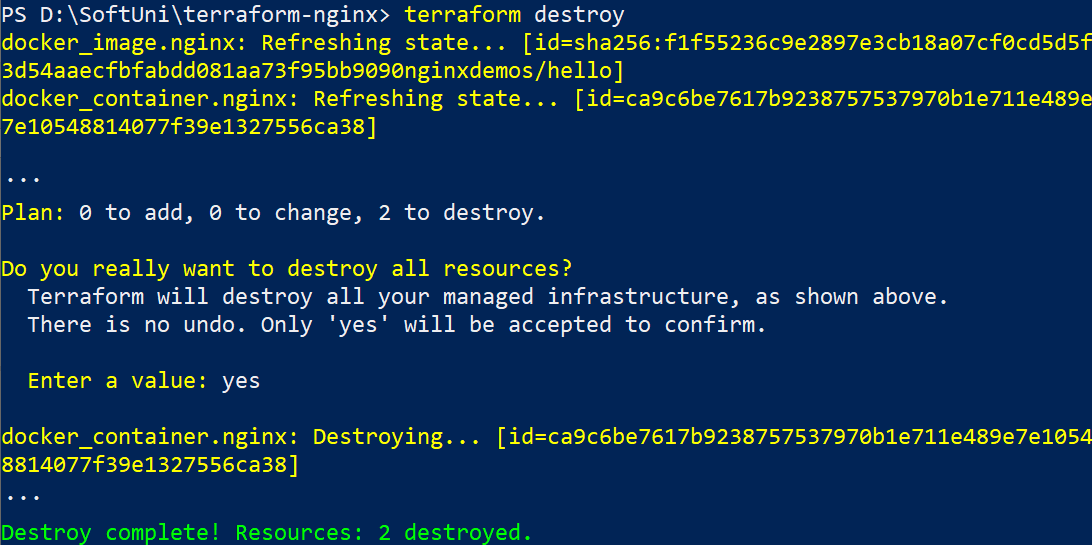


As you saw, you can **change the** Terraform **configuration** **and resources** easily in a way you want. However, **don't forget to examine** Terraform **execution plans** **carefully before applying them**, so that you don't perform incorrect or dangerous actions.

### Step 6: Destroy Configuration

Once you **no longer need infrastructure**, you may want to **destroy it**. Use the terraform destroy **command** to **terminate resources** managed by your **Terraform project**.

**Destroy the resources** you created:



Terraform **destroys resources** in a **suitable order** to **respect dependencies**. You can **verify that the NGINX image and container don't exist anymore** in Docker Desktop.