Deploying on Docker with Portainer

Learn how you can use Portainer to deploy your applications on Docker and Docker Swarm environments, including an introduction to containerization and real world example deployments.

SWARM AND SERVICES

SUMMARY

Introduction

Welcome!

Thank you for starting the Deploying on Docker with Portainer training course!

This course is intended as an introduction to the concepts of deploying on Docker environments, and will cover the following topics:

1. Containers - what they are, how to create them, and how to edit them once created.
2. Volumes - what a volume is, the different types of volumes, how to create new volumes and how to attach them to containers.
3. Stacks - the concepts behind stacks and how they can be used to build powerful, repeatable and reliable deployments.
4. Docker Swarm - what Swarm is, the differences between Docker Swarm and Docker Standalone, and use cases for Swarm.
5. Swarm services - how a service differs from a container, how to spin one up and how to scale one after creation.

By the end of this course you should have a good understanding of deploying on Docker environments. We won't cover every piece of functionality here, but this should get you confident with the important deployment concepts and prepare you for more advanced topics.

So, let's get started!

Anatomy of a container

Anatomy of a container

Before we start working with containers, we first need to understand what containers are and how they fit into the Docker ecosystem.

What is a container?

In its simplest form, a container is a unit of software designed to perform a task. That task might be simple (serve a static web page) or complex (machine learning) or somewhere in between. Containers run independently, but can communicate with other containers and non-container resources if required.

When we talk about containers, we're generally talking about the containerized format popularized by Docker, which has since become the industry standard used by a multitude of environments.

Think of an assembly line manufacturing cars. In this scenario, the container is an assembly robot responsible for one part of the car-assembly process, but it works with other robots (containers) to build the car.

Containers can be thought of like robots in an assembly line: dedicated to one task.

Each robot in the assembly line has a set of instructions for the task it needs to perform. Robots might be physically identical but they perform different jobs based on the instructions they're given. When it comes to containers, those instructions take the form of a container image. It is the image that tells the container to be a web server, a database or something else.

Is a container like a virtual machine?

In some respects, containers and VMs have a lot in common. But they have one key difference: virtual machines virtualize the hardware of the underlying system, while containers virtualize the operating system.

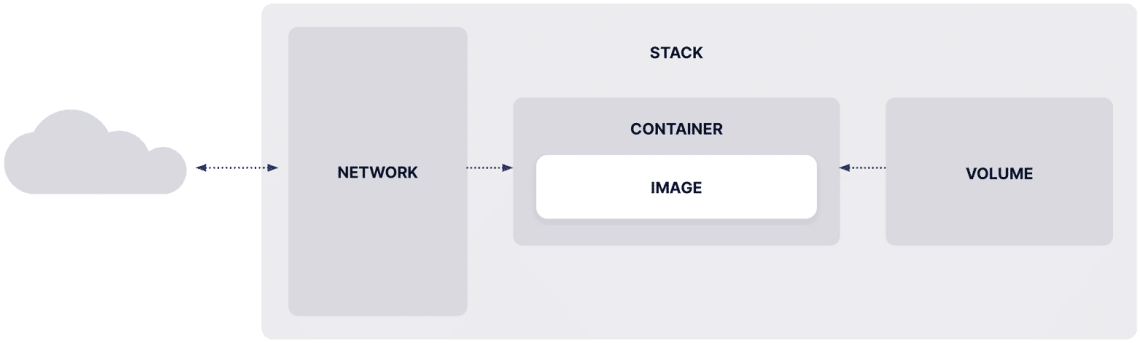
Containers are designed to be idempotent. You should be able to destroy a container and create a replacement without any data loss. This is achieved by providing the basics of

the configuration via the image, and any customizations through deployment options such as environment variables. If a container needs to store something persistently, for example a database, a volume should be attached to the container to provide this persistency without sacrificing the idempotency of the container itself.

Idempotency makes containers more flexible and portable than virtual machines. If we go back to the assembly line metaphor, imagine moving your robot to a new assembly line that builds a different model of car. The physical robot (container) can still do the job, but it will need a new set of instructions (container image). A virtual machine would need to destroy and rebuild the entire factory to change the model of car you're building.

Container components

As described above, a **container** is the running result of an **image** deployed onto an orchestrator (such as Docker). Containers by themselves are idempotent, so if they need persistent storage we would attach a **volume**. Access to the container (both externally and internally from other containers) is provided by a **network** configured on the orchestrator. And all of this can be grouped together, deployed and managed as one in a **stack**.



The components that can make up a containerized deployment.

We will cover each of these concepts and more in further detail in future lessons. For now though, let's dive right in and create our first container.

Deploy your first container

Now that we understand the concepts behind a container, let's deploy one on our environment using Portainer. For this example, we will:

1. Create a container running nginx
2. Configure the container to listen on a specific port
3. Make a directory on our host accessible within the container

START

Preparation

On your host environment, create a directory called `/mnt/nginx` (or use an alternative path that suits your environment). Inside that directory, create an `index.html` file with the following content:

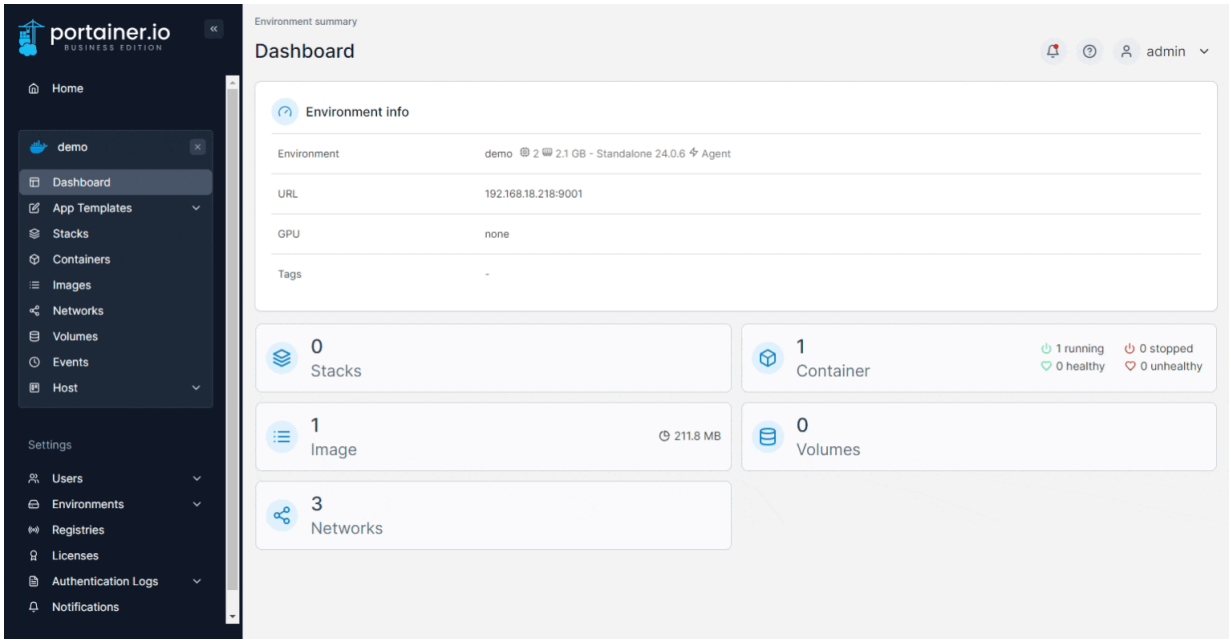
```
<html>
    <head>
        <title>Welcome to nginx!</title>
    </head>
    <body>
        <h1>Welcome to your nginx container!</h1>
        <p>If you can read this, your nginx container was created successfu
    </body>
</html>
```

Make a note of the directory name you used here (if you changed it) as you'll need it later.

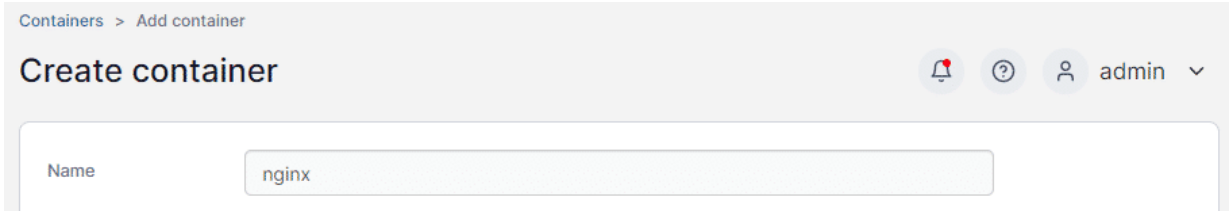
CONTINUE

Create your container

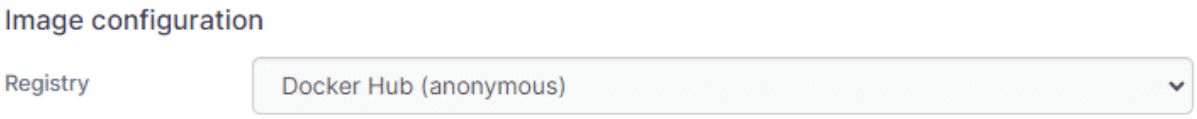
In Portainer select your environment then choose **Containers** from the left hand menu, then click **Add container**.



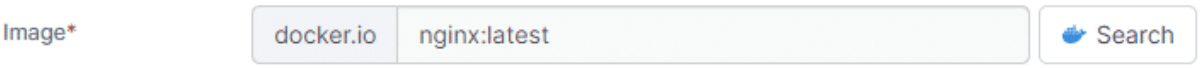
The first thing you'll need is a **name** for your container. This can be whatever you want it to be, as long as it's unique on the environment. For this example, we'll call our container `nginx`.



As we've discussed, a container is built from an image, so we now need to tell Portainer which image we want to use and which registry that image is coming from. Since there are official nginx images on Docker Hub, we can use **Docker Hub** as the registry.

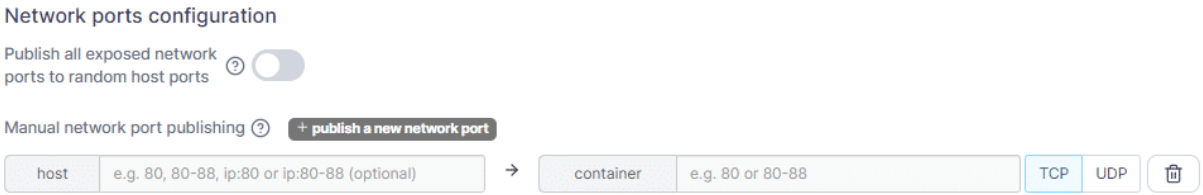


For the image, we'll use nginx with the latest tag, which you define using the `image:tag` format - so for this example, `nginx:latest`:



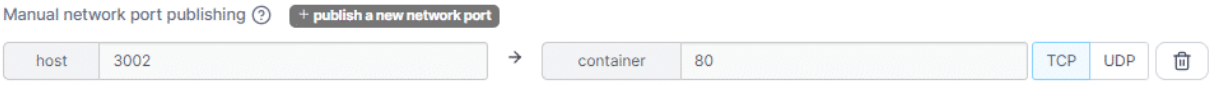
At this point you could deploy the container, but you wouldn't be able to access the nginx server with your web browser. To allow access, we need to publish a port that will be available externally and route to the internal port of the nginx server running in the container we're creating.

In the **Network ports configuration** section, click the **publish a new network port** button next to the **Manual network port publishing** label. A couple of new fields will appear, where we define the host port and container port as well as the protocol.



The **host** port is the port that will be available externally, and the **container** port is the internal port within the container that the host port will be routed to. The host port must be unused by other services on your host. The container port will be defined by the particular services configured in your image - in the case of nginx, the nginx service listens on port `80` by default.

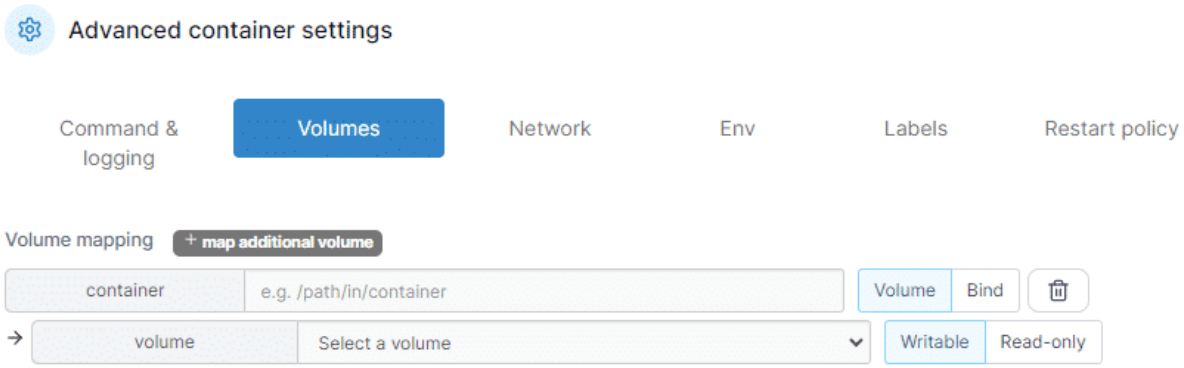
For this example, let's publish this container on port `3002` by setting the **host** field to `3002`. If this port is already used on your environment, feel free to change it to suit. The **container** port, as mentioned above, should be `80`. The **protocol** should be kept at TCP.



Finally, we want some content for our nginx server. We can use the directory we created earlier to contain the website files we want to make available through our nginx container. To make this directory available within the container, we need to create what's called a **bind mount**.

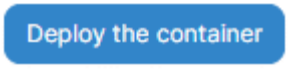
A bind mount is one of the two primary volume types (the other being **named volumes**, which we'll cover in a later lesson) and is essentially a way of making a directory on the host environment available within a container at a specified path. Containers have their own self-contained file systems which only exist while they are running, so anything you want available in the container persistently must be mounted in via a bind mount or named volume.

To add a bind mount to this container's configuration, scroll down to the **Advanced container settings** section and select the **Volumes** tab. From this tab, click **map additional volume**. Some new fields will appear which we'll now complete.

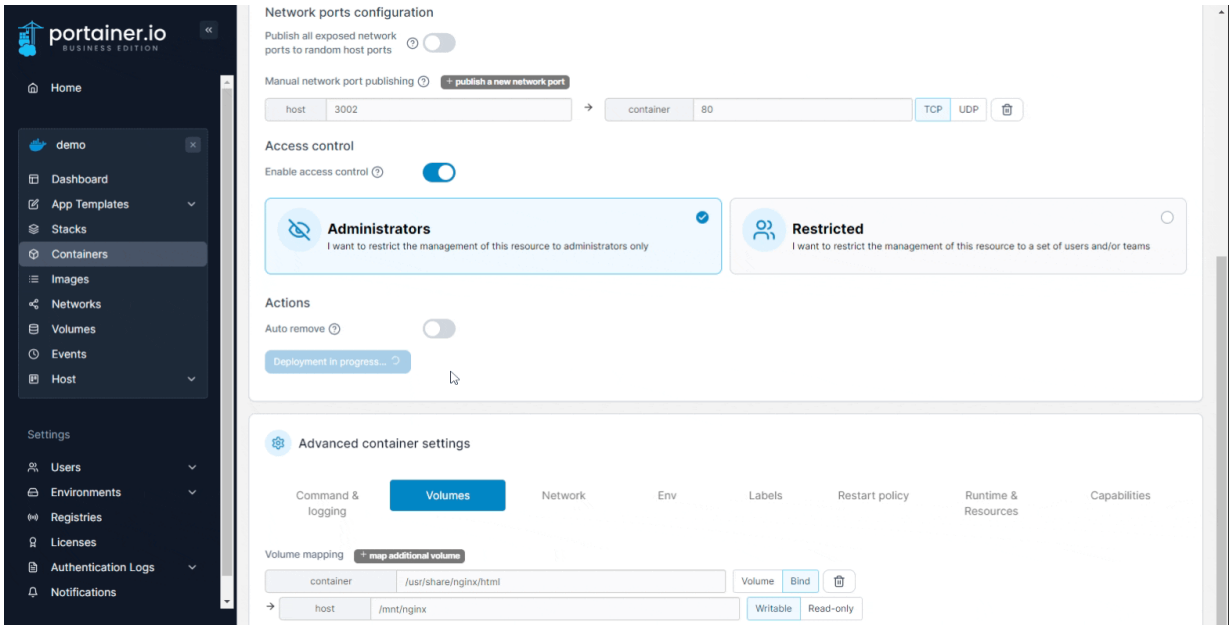


Since we're adding a bind mount, we first need to click the **Bind** button to change the **volume** field into the **host** field. The **container** field is the path within the container where you'd like your mounted directory to appear, and the **host** field is the path on your host that you'd like to mount in your container. Since the default webroot path for the nginx image is `/usr/share/nginx/html`, we enter that in the **container** field. For the **host** field, enter `/mnt/nginx` (or the path you used above) as that's the directory we want mounted in our container. For this example, we'll leave the write permissions on the default of **Writable**.

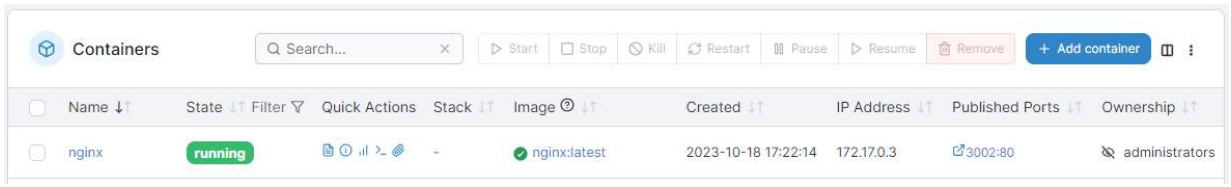
This should be everything we need to get our nginx container up and running. Click the **Deploy the container** button to create the container.



Portainer will now take your settings and deploy the container on your environment. Once it completes you'll see a notification in the top right of the window and be taken back to the list of containers.



You should see the `nginx` container in the list, with a **running** state to indicate it is active. You'll also see the image name and tag we used and the port we published.



Let's test our deployment. In a new browser tab, go to the IP address of your environment with the port we published. For example, if your environment's IP was `192.168.1.1` and you used the default port of `3002`, go to:

```
http://192.168.1.1:3002
```

If you have configured your environment's public IP address, you can also click on the published port in the container list to open it in a browser.

If all has gone well, you should see the contents of the `index.html` page we created earlier.

CONTINUE

Summary

In this lesson we've covered the creation of a container with Portainer, including:

- [ ] Naming the container
- [ ] Selecting an image repository and entering an image name and tag
- [ ] Publishing a port
- [ ] Bind mounting a directory from the host

We've learned how to deploy a container, so now let's learn how to edit a deployed container with Portainer.

Editing your container

Now that you have deployed your container, you should be up and running. But what if you want to make changes to the container's configuration?

Let's say you need to change the image tag you're using on the container. In our previous lesson we created a container using the `nginx:latest` image and tag, which will have pulled the most recent version of the image to use. Let's imagine however that we need to ensure the container runs on a specific version of nginx.

Let's also imagine that we need to change the port the container is published on (perhaps you want to run something else on the port it currently uses). We can make both of these changes to the container at the same time and redeploy it with the updates, all through the Portainer interface.

START

1

Preparation

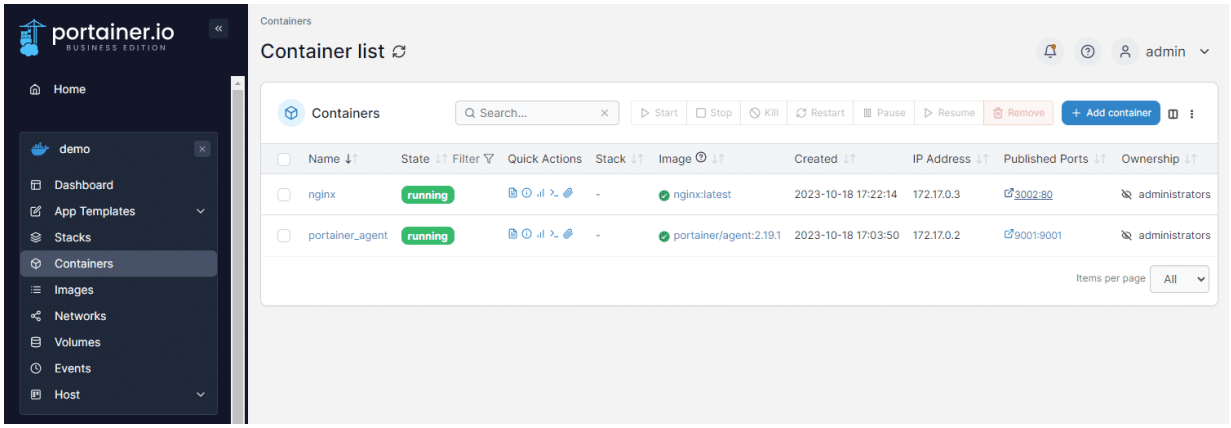
This lesson assumes you have a nginx container deployed as per the previous lesson. If you don't have this container, you can adjust these instructions to suit your setup.

CONTINUE

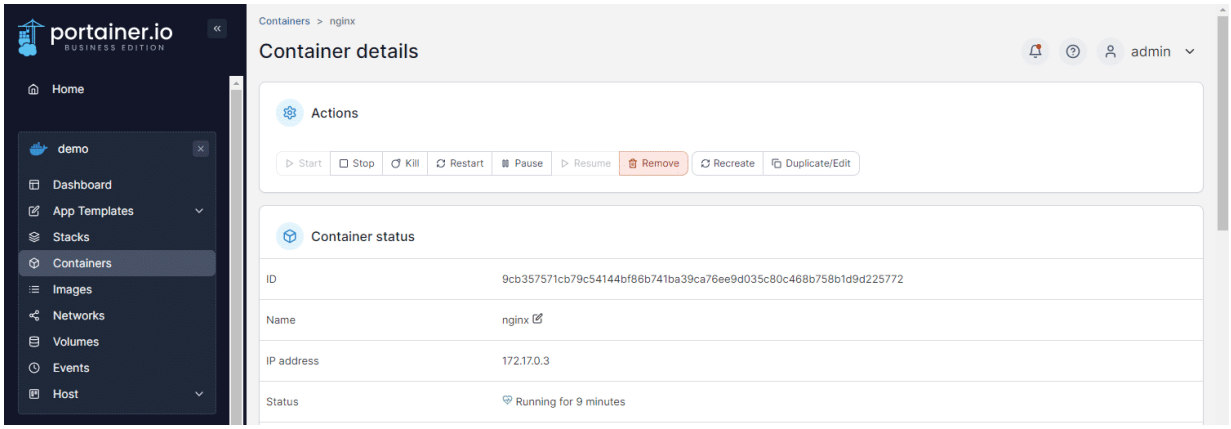
2

Edit your container

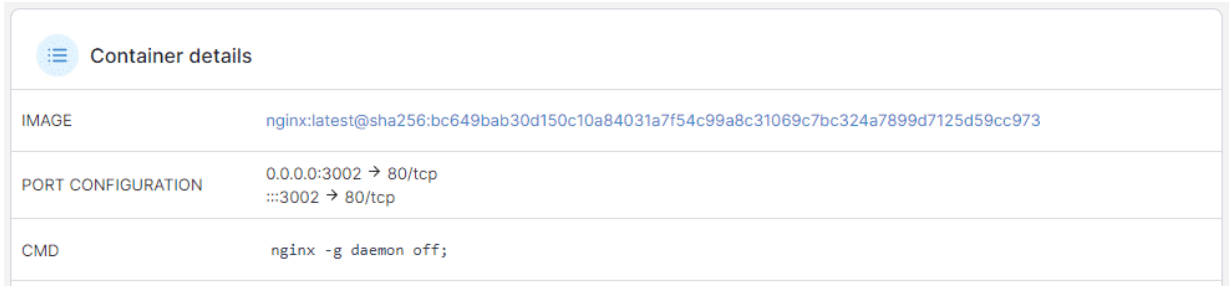
First, go to the list of containers on your environment by clicking **Containers** in the left menu. You should see a list of the containers deployed on the environment.



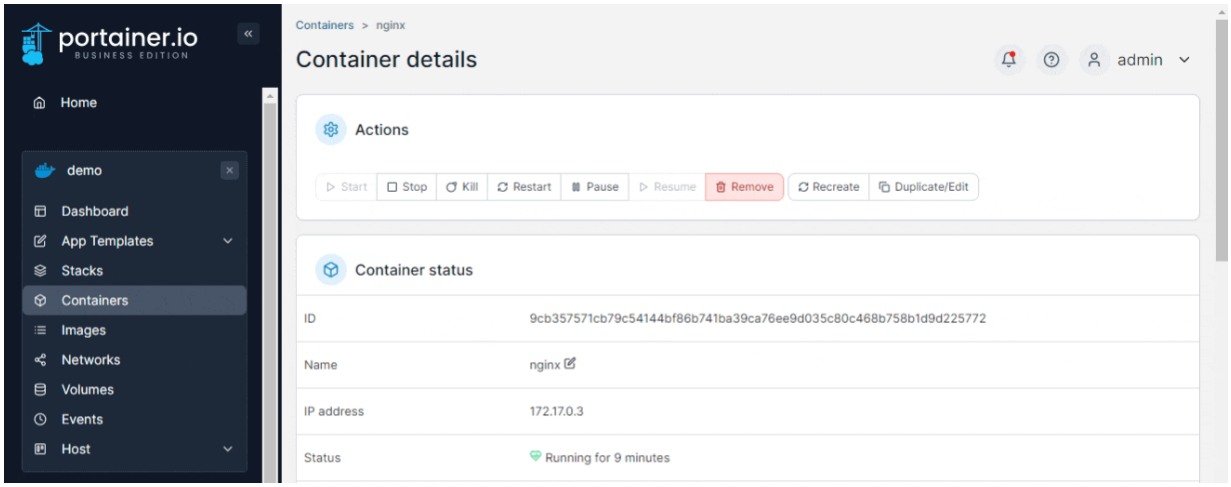
From here, click the `nginx` container we created earlier. This will take you to the details page for the container.



If you scroll down to the **Container details** section, you can see information about the container including the image you're using as well as the port configuration, both of which we'll be adjusting now.



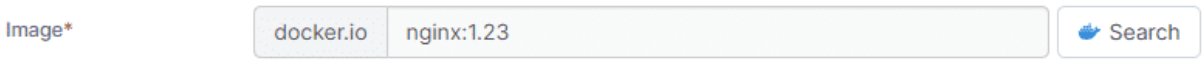
Scroll back up to the top of the page and click the **Duplicate/Edit** button to start editing.



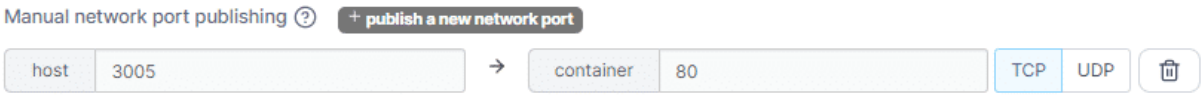
This page should look familiar, as it is essentially the same as the container creation page.

First, let's adjust the image tag we're using. For this example, instead of `nginx:latest` we'll set this to `nginx:1.23` to specify we want to use version `1.23` of nginx specifically, instead of what's tagged as `latest`.

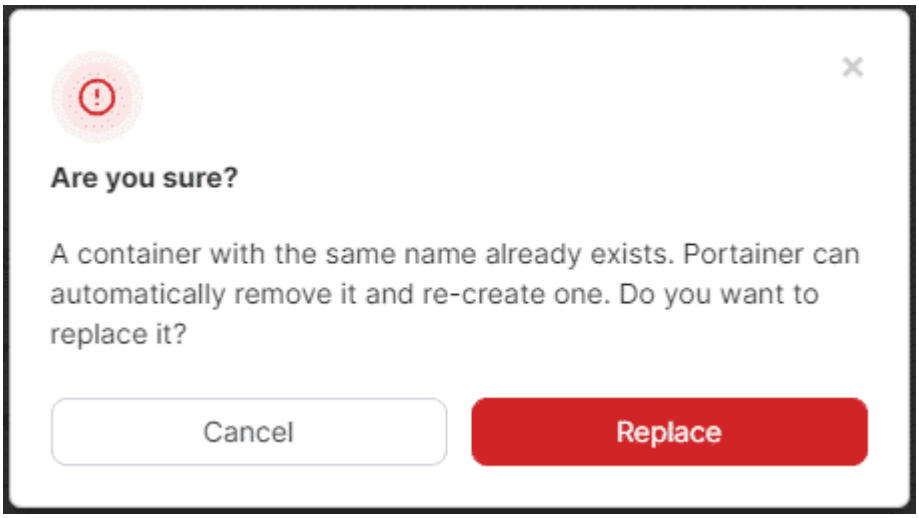
In the **Image** box, change the entry to read `nginx:1.23`.



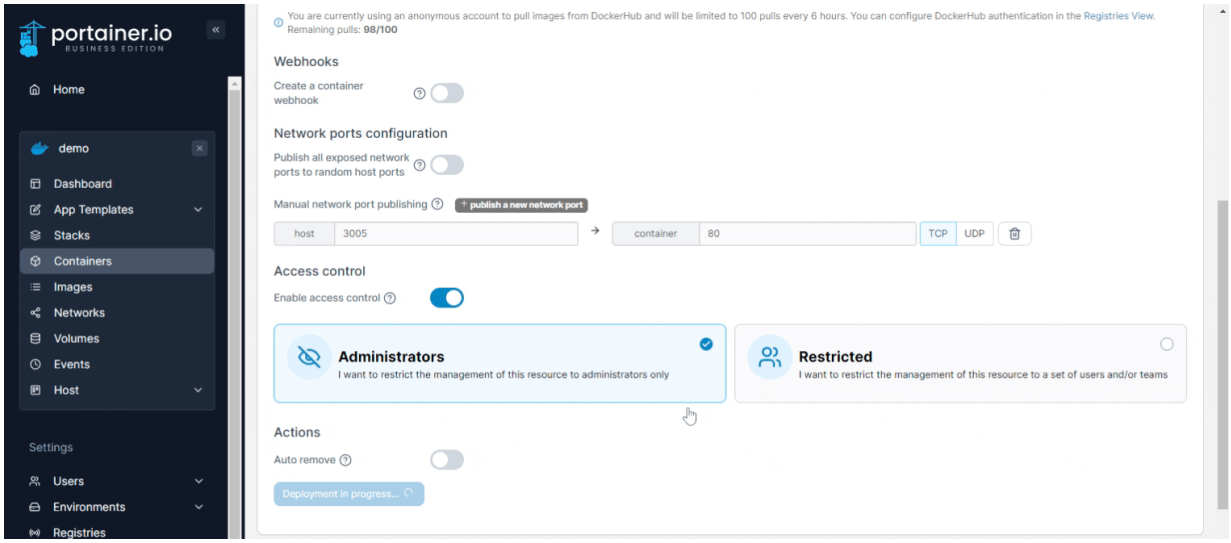
Now let's adjust the published port. In the **Network ports configuration** section you should see the port we published earlier - `3002` - listed as routing to port `80`. For this example, let's change the host port to `3005` instead of `3002`. Enter `3005` in the host box, replacing the current entry. We don't need to change the container port, as nginx is still listening internally on port `80`.



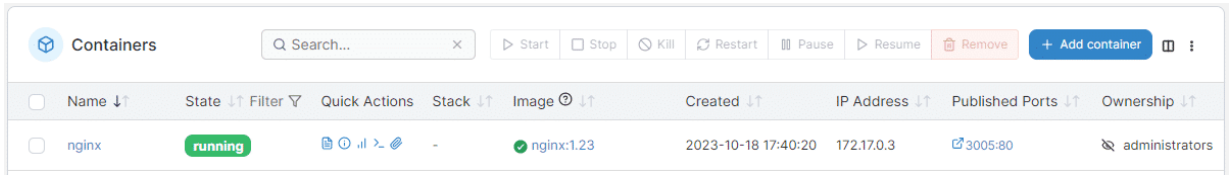
Check that your settings are right then when you're ready click **Deploy the container**. You'll be asked to confirm your action as the container already exists, which we expect, so click **Replace**.



The deployment will then begin, removing the old container and creating a new one with the new settings to replace it. Once this process completes you'll be taken back to the container list.



From here we can see that the image for the nginx container is now set to `nginx:1.23` and the published port is now `3005` instead of `3002`.



You can confirm the new published port is working by accessing it in a browser:

```
http://192.168.1.1:3005
```

Replace `192.168.1.1` with the IP address of your environment.

CONTINUE

3

Summary

In this lesson we have learned how to:

- [ ] Edit an existing container
- [ ] Change the image and/or tag that a container uses
- [ ] Modify the published port of a container

In the next lesson we'll talk about the concept of volumes, which we briefly touched on earlier.

Volume concepts

In this lesson we'll learn about volumes in Docker. We'll cover:

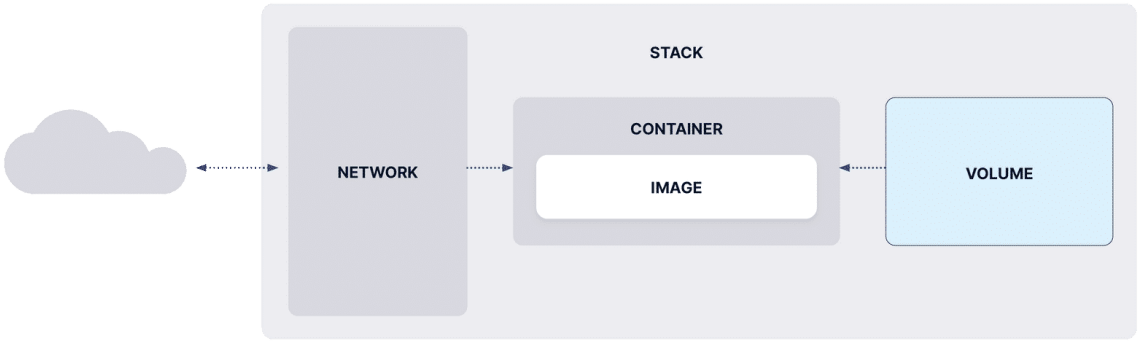
1. What a volume is in Docker terms
2. The main two types of volumes: named volumes and bind mounts.

START

1

What is a volume?

A volume is a way for a container to be given persistent storage for files that are outside of the image itself but need to remain through restarts of the container.



Our application component diagram, with the volume component highlighted.

One of the best examples of where you'd need a volume is if you were hosting a website with your container. The container would provide the web server software but not your specific HTML / CSS / JS for your particular website. These site files would potentially change often, but would need to be kept around if you needed to restart or upgrade the container (for example, to deploy an updated version of the web server software). To achieve this persistency, you would attach a volume to your web server container and put your website files in that volume. The web server container would see this volume as part of it's filesystem at the path you provide, and would be able to use it accordingly.

CONTINUE

2

Named volumes vs Bind mounts

There are two primary types of volumes available in Docker: **named volumes** and **bind mounts**. Each type has it's own benefits and drawbacks, functioning in different ways externally but both appearing in the same way to the container's internal file system.

Bind mounts

A bind mount is fairly straightforward - it is a way to mount a directory from the host machine into your container. You simply define the directory on your host system (for example, `/mnt/nginx`) and the path where you'd like it to appear within your container's filesystem (for example, `/user/share/nginx/html`). That's it. If this example looks familiar, that's because if you followed our deploying a container lesson you would have already created a bind mount.

While simple, there are certain considerations you should keep in mind when using bind mounts. Because you are mounting your host filesystem into your container, your container has access to that filesystem. This could be a security issue if you're running an insecure or malicious container image. A bind mount is also managed outside of the Docker orchestrator, so you would need to ensure that the directory stays at that path, has the right permissions, and isn't modified unexpectedly by other processes on the host system.

Named volumes

Named volumes, often simply referred to as "volumes", are Docker's answer to persistent storage. When you create a volume, Docker builds a virtual file system that can then be attached to a container (or multiple containers) to serve as persistent storage. This virtual

file system, while existing on the host, is managed by Docker itself. As such, you're not mounting the host filesystem directly into your container at all, unlike with a bind mount.

Once a volume has been created, you can mount it to a container by selecting the volume and the path where you'd like it to appear within your container's filesystem (much like a bind mount above). Because the volume is referenced by a name rather than a path, this makes it more portable especially when you consider how different paths may work on different OSes and deployments. You can even create and mount volumes as part of a stack deployment alongside your containers to ensure portability.

There are of course cases where a volume might not make sense - for example, if you had a directory with a large amount of files that you wanted to make available to other services outside of your container, but in many situations using a named volume can provide you with benefits that a bind mount cannot.

What about network volumes?

Docker also supports the mounting of network volumes (such as NFS or CIFS shares) within a container, and these essentially work in a similar way to a bind mount in that you provide the source path (and often in the case of a network mount the necessary access credentials) and the path to mount within the container file system. Also much like bind mounts, network volumes are managed outside of Docker so don't have the benefits of named volumes.

CONTINUE

Summary

In this lesson we've learned:

- [ ] What a volume is
- [ ] The two main types of volume

Next, let's take that knowledge and use it to create a new named volume.

Creating a named volume

In the previous lesson we introduced the concepts of volumes in Docker. In this lesson we'll take that knowledge and use it to create a named volume. We will:

1. Learn the concepts of named volumes, and how they work on Docker and Swarm environments.
2. Create a new named volume in our Docker environment.
3. Attach our new volume to a container.

Docker refers to named volumes as simply "volumes". We'll use both terms interchangeably below.

START

1

Named volume concepts

Bind mounts, which we covered in the previous lesson when creating our first container, are relatively straightforward to understand. They are simply making a directory on the host server available within a container at a specified path. As such, a bind mount requires a container to exist.

Named volumes however can and do exist without a container. You would generally create a named volume independently from a container, or as part of a stack definition (which we'll cover in a later lesson). Named volumes are managed by Docker itself, are less dependent on the underlying host directory structure, and through the use of drivers provide the ability to attach directly to storage on remote hosts, for example via NFS or CIFS. And from a security perspective, having your persistent storage in a separately managed system from your host environment helps to protect against malicious or misconfigured containers from accessing the host file system.

A note for Docker Swarm users

One important point to note for Swarm environments is that Swarm itself does not contain any functionality to replicate volumes and the data within across nodes. When creating a volume on Swarm, you will need to specify the node on which that volume resides. If you need persistent storage across Swarm nodes for your application, a third-party storage system such as Ceph or GlusterFS is needed, which is outside the scope of this course.

Now that we have a basic understanding of the idea behind named volumes, let's go ahead and create one.

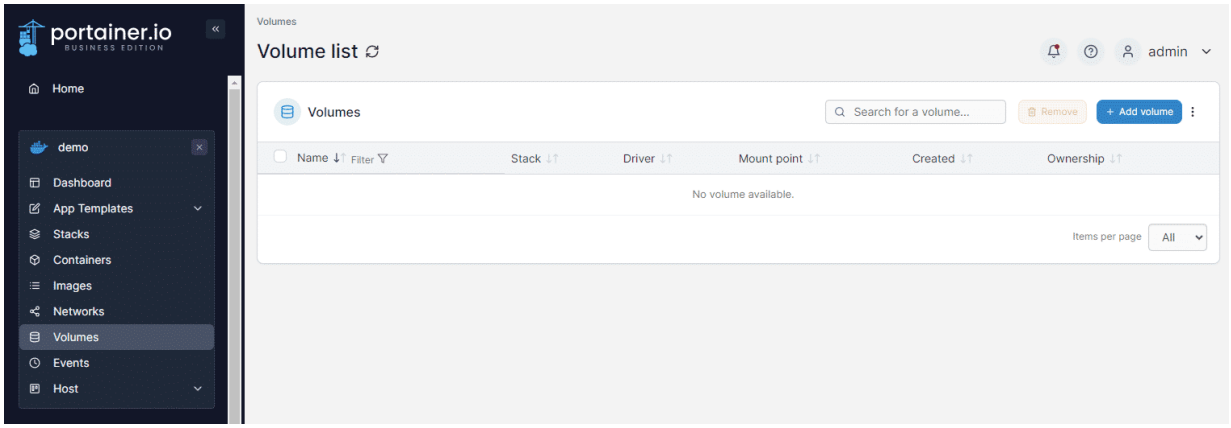
CONTINUE

2

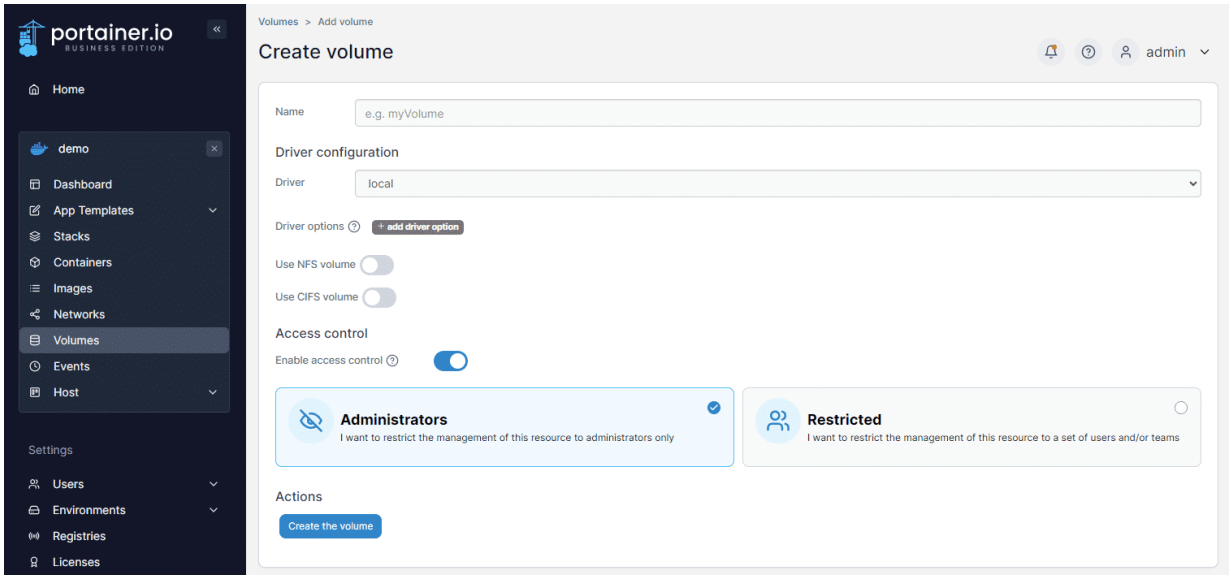
Create a named volume

First, we'll need to select our Docker environment in Portainer. This can be either Docker Standalone or Docker Swarm, and either a local environment or an Agent-managed environment - either will work.

Once you have selected your environment, click **Volumes** in the left hand menu to list the current volumes on your selected environment.

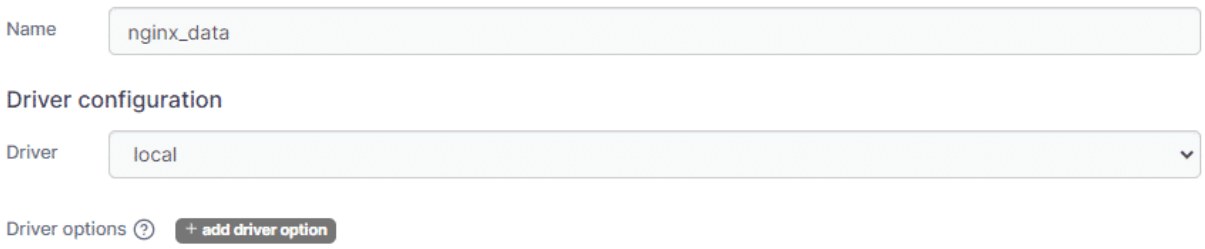


In my environment I don't have any volumes yet, so let's make one. To create a new volume, click the **Add volume** button in the top right. This will take you to the Create volume page.



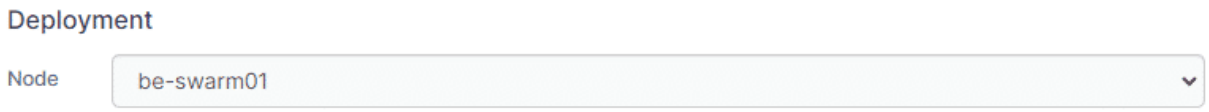
Every named volume needs, well, a **name** - so let's provide that first. Volume names must be unique to the host machine they are created on, and can only consist of alphanumeric characters, upper or lower case, the dash, underscore, and period. For this example, let's call ours `nginx_data`.

For the **Driver**, keep this on local for this example. It's likely that this will be the only option, in any case. Additional drivers are an advanced feature and outside this lesson's scope. We also don't need any driver options for this example.



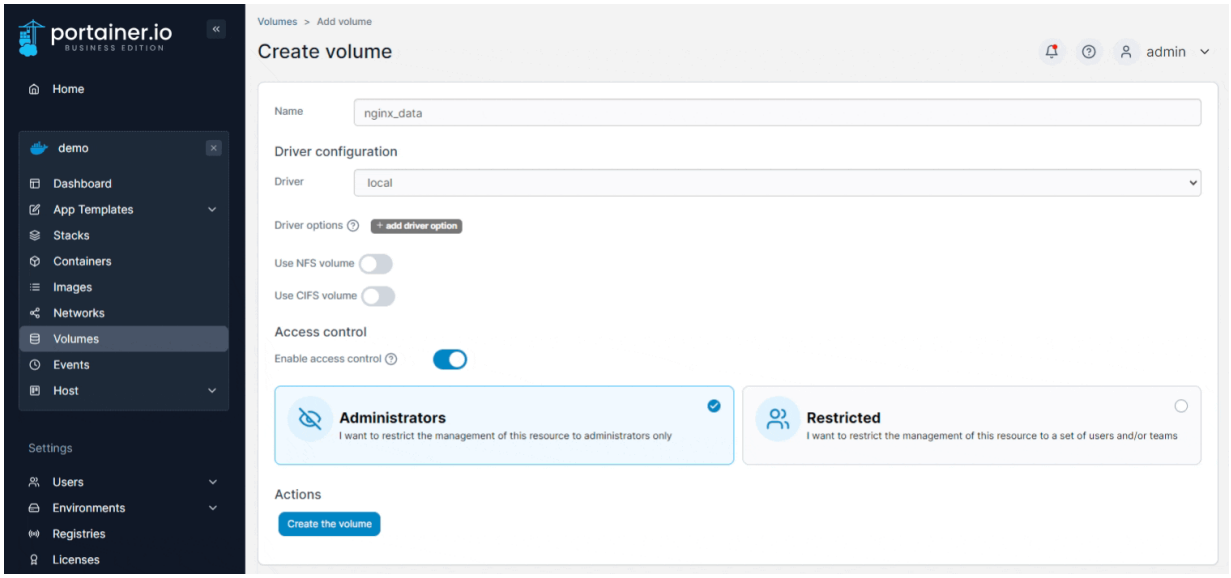
Next you'll see options for either a NFS volume or CIFS volume. Again, for this example we are going to skip these options, but if you were creating a volume from a NFS or CIFS location, you would enable one of these options and fill in the required configuration.

If you're on a Swarm environment, you'll also see a **Deployment** section here and a Node selector. This lets you specify the node that you want to create the volume on. As described above, Swarm doesn't do anything special with volumes as compared to Docker Standalone, so you must specify the node that the volume will reside on and take that into account when provisioning your containers.



Only shown for Docker Swarm environments.

With everything configured, we can now click **Create the volume**. The volume will be provisioned and you'll be returned to the volume list page, where you should now see your new volume.



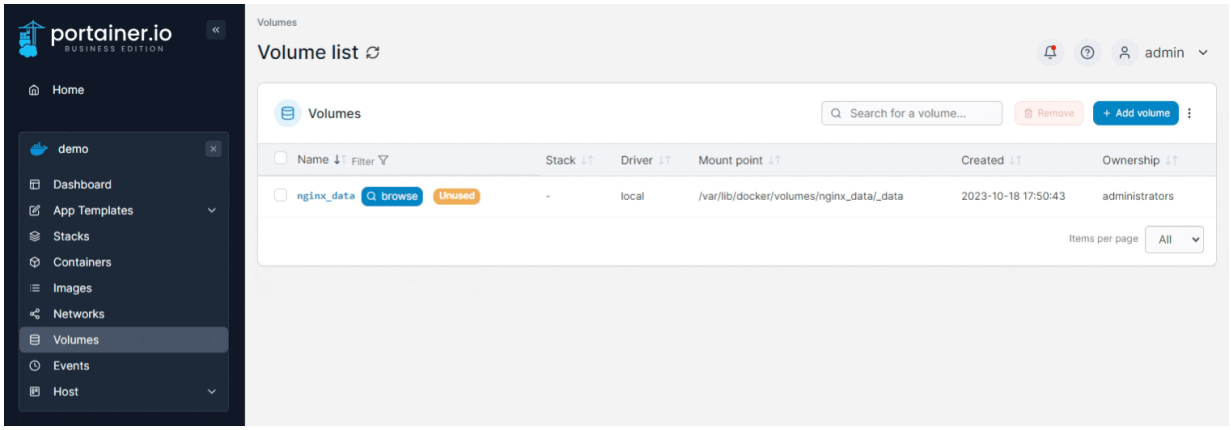
CONTINUE

3

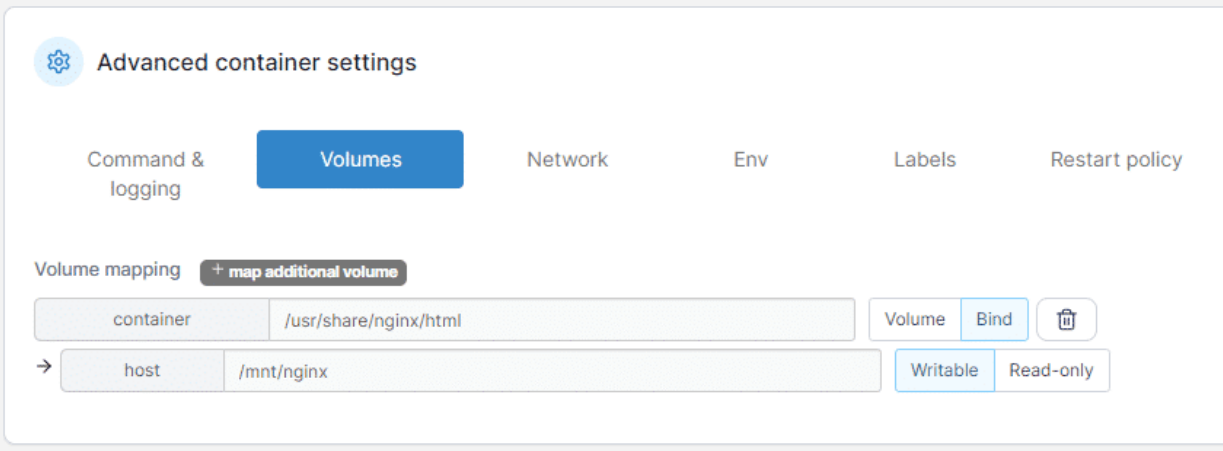
Attach a volume to a container

Now that we've created our new named volume, we want to attach it to a container. For this, we can use the nginx container we created in the previous lesson.

From Portainer, select your Docker environment then select **Containers** from the left menu, Then, click the name of the nginx container we created earlier. This will take you to the container details page. We want to make changes to this container's configuration, so click the **Duplicate/Edit** button.

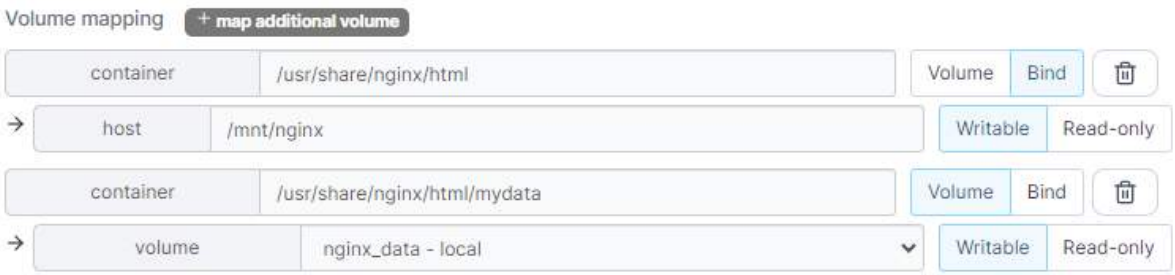


Now that we're editing our container, scroll down to the **Advanced container settings** section and select the **Volumes** tab. You should see the bind mount we added when creating the container here.



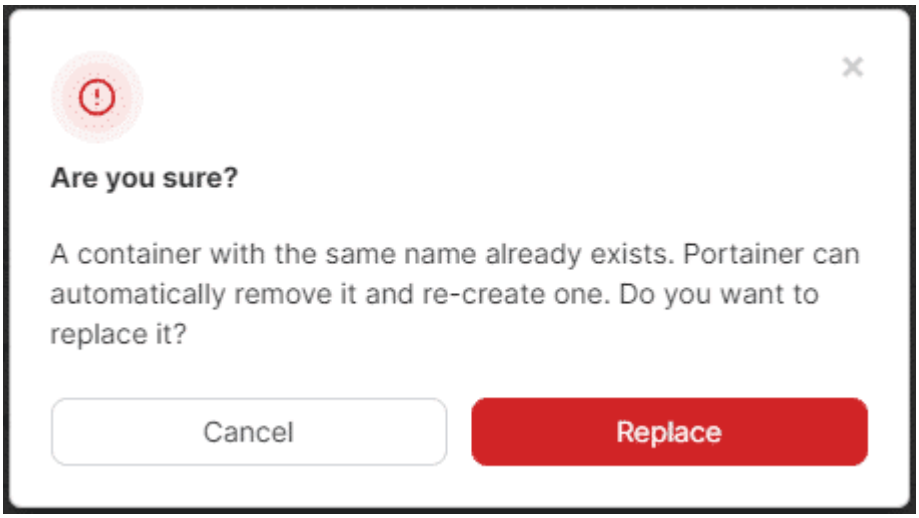
We can either remove our existing bind mount and replace it with the named volume mount, or alternatively we can add the named volume alongside the bind mount at a different path within the container. For this example, let's take the second option. Click the **map additional volume** button to add a new set of fields. In the new container field, enter our path within the container - for this example, let's mount it at `/usr/share/nginx/html/mydata`. Ensure the **Volume** option is selected, as that's what we're adding. Then from the volume dropdown, select the `nginx_data` volume we created previously.

Your volume mappings should end up looking like this:

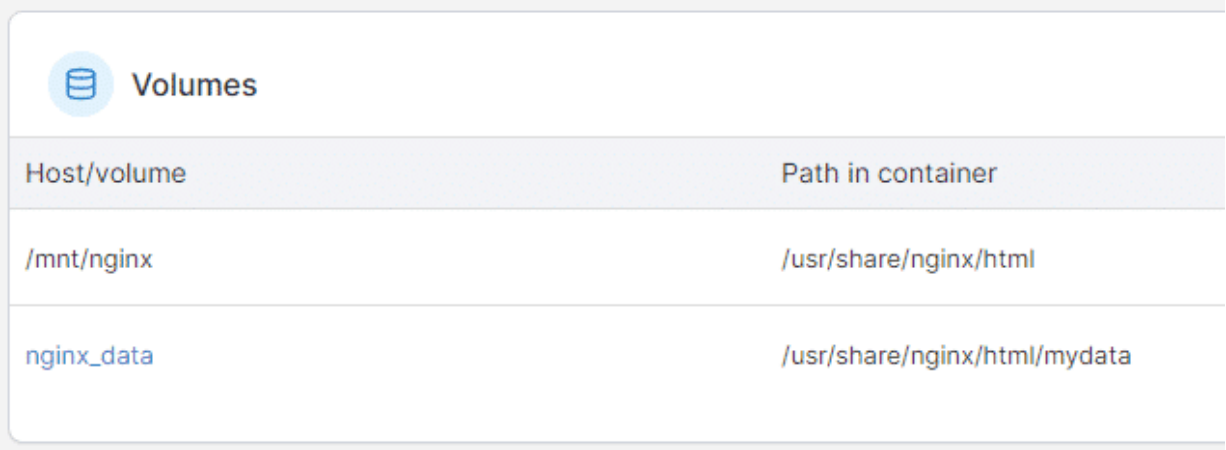


Once you're happy, click the **Deploy the container** button just above the Advanced container settings section. You'll be asked to confirm your action as the container already

exists - click **Replace** to proceed.



Portainer will now remove the old container and spin up a new version of it with the new configuration. You can confirm this has happened by clicking the name of the container to go to the details page, then scroll down to the **Volumes** section where you will see both the bind mount and named volume attached.



You can even browse to the IP of your server and the port we chose to view the container contents as before, with the contents of the `nginx_data` volume available at

the `/mydata` subpath.

Note you will get a `403 Forbidden` error when trying to browse to this path - this is because there is no data in our new volume. Adding data to your named volume is outside of the scope of this lesson, though if you are managing your Docker environment with the Portainer Agent and have volume browsing enabled, you can browse the contents of your volume and upload files directly from within the Portainer interface under Volumes in the left menu.

CONTINUE

4

Summary

In this lesson we've covered the following:

- [ ] The concepts behind a named volume, including specifics to note about volumes on Swarm environments.
- [ ] Creating a basic named volume through Portainer.
- [ ] Attaching our new named volume to the nginx container we previously created.

In the next lesson we'll look at how we can combine all we've learned so far about containers and volumes into one deployment through the use of Stacks.

Anatomy of a stack

In previous lessons in this course, we've talked about containers and volumes (both bind mounts and named volumes) - both elements that can help to make up an application deployment. But rather than creating those elements individually, what if we could provision everything we need to run our application all at once?

In this lesson, we'll talk about how we can do that using stacks. We'll cover:

1. What a stack is
2. The basic structure of a stack file
3. Some of the elements that can be used when constructing a stack

A quick note about Compose and Stacks

When we talk about stacks in Portainer, we can be referring to one of two things, depending on the type of environment:

- For Docker Standalone environments, Stacks refers to Docker Compose file deployments - for example, what you would use a "docker compose up" command to start via the CLI.
- For Docker Swarm environments, Stacks refers to Docker Stack file deployments - for example, what you would use a "docker stack deploy" command to start via the CLI.

Functionally there's not very much difference between Compose stacks and Swarm stacks - they're both created from YAML files that use the same format. There are some definitions that are specific to either Compose or Swarm, but from an introductory concept perspective we can talk about them interchangeably. We'll cover some of the Swarm-specific functionality in a later lesson.

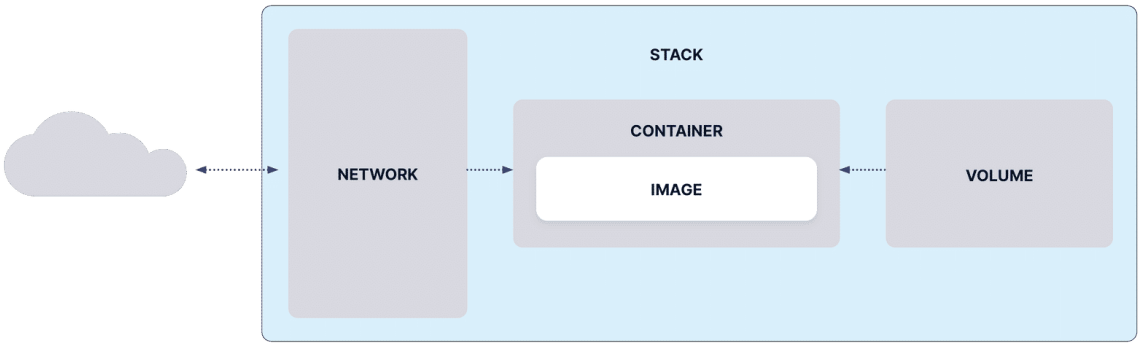
Because of these similarities, we opted to use the term Stacks across the Portainer application for both Compose stacks and Swarm stacks.

START

1

What is a stack?

A stack is a way of providing a single definition that creates multiple elements, usually (but not always) related to a single application deployment. A stack is defined in a YAML-format file.



Our application component diagram, with the stack component highlighted.

To understand this further, let's look at an example application - WordPress. If you're not familiar with WordPress, it is a blogging platform you can host and customize yourself. As a web application, it has a few requirements:

- A web server (such as Apache or nginx) with PHP installed and configured.
- A database server (such as MySQL or MariaDB) within which it can store the blog posts and other data.
- A place to store the WordPress code, plugins and themes.

On a traditional setup, you might have a VM that runs Apache with PHP and MySQL to host this. In the container world though, best practice is to have a container for each

individual service, and volumes for persistent storage (the WordPress code, plugins and themes, as well as the database files). For example:

- A container named `wordpress` that contains the web server (Apache with PHP)
- A volume named `wp_data` for the WordPress code, plugins and themes
- A container named `db` for the MySQL application
- A volume named `db_data` for MySQL to store the database files

Rather than creating (and managing) each of these elements individually, we can put them into a single stack file definition. As well as being more organized, this improves the portability of the application, as we could take that stack file, deploy it on a different Docker installation and be confident that our application will come up with everything it needs, first time.

CONTINUE

The stack file structure

Now that we have an example of how we might use a stack, let's create one. As above, we'll create a stack for our WordPress deployment with:

- A container named `wordpress` that contains the web server (Apache with PHP)
- A volume named `wp_data` for the WordPress code, plugins and themes
- A container named `db` for the MySQL application
- A volume named `db_data` for MySQL to store the database files

Turning this into a stack file, we might end up with:

```
version: '3'

services:
  wordpress:
    image: wordpress:php8.1
    volumes:
      - wp_data:/var/www/html
    ports:
      - 8088:80
    restart: always
    environment:
      WORDPRESS_DB_HOST: db:3306
      WORDPRESS_DB_USER: wordpress
      WORDPRESS_DB_PASSWORD: wordpress

  db:
```

```
    image: mysql:8.1
    volumes:
      - db_data:/var/lib/mysql
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: secure_root_password
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: wordpress

volumes:
  wp_data:
  db_data:
```

This might look confusing, but once you understand the concept it is pretty straightforward. Let's start at the top.

Stack files are YAML files, and as such adhere to the YAML specification, which has strict formatting requirements, in particular around indentation. This is outside the scope of this course.

CONTINUE

Version

```
version: '3'
```

This specifies the `version` that your stack configuration adheres to, so that Docker knows which specification to reference when parsing it. This is mostly for informational purposes these days and isn't strictly necessary for Compose stacks, but we recommend including it anyway.

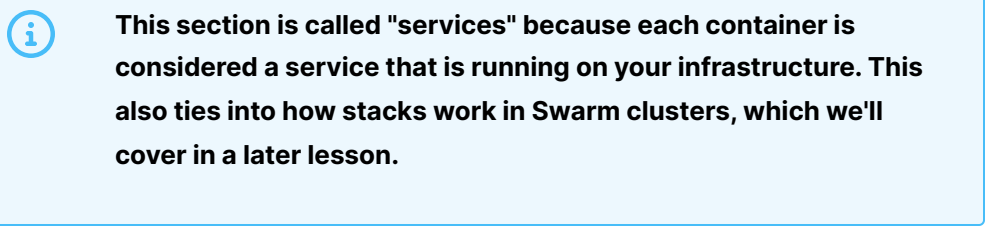
The rest of the configuration file is split into sections based on the type of resource, starting with Services.

Services

```
services:
  wordpress:
    image: wordpress:php8.1
    volumes:
      - wp_data:/var/www/html
    ports:
      - 8088:80
    restart: always
    environment:
      WORDPRESS_DB_HOST: db:3306
      WORDPRESS_DB_USER: wordpress
      WORDPRESS_DB_PASSWORD: wordpress

  db:
    image: mysql:8.1
```

```
    volumes:
      - db_data:/var/lib/mysql
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: secure_root_password
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: wordpress
```

Within the `services` section, we define our containers. Each container is defined by their name (for example `wordpress` or db) and underneath that definition we specify the necessary options for the container.



Let's now look at the options for each service.

Image

```
  wordpress:
    image: wordpress:php8.1
```

First, we specify the `image` that the container uses. You'll remember from an earlier lesson that the image is the "script" for your container, telling it what to do. In this case, we're using the `wordpress` image with the `php8.1` tag. This image includes Apache, PHP 8.1, and WordPress itself. For the database service, we're using `mysql:8.1`.

Volumes

```
    volumes:
      - wp_data:/var/www/html
```

Next we define the volumes this container will have access to. These are provided in a list format, with hyphens denoting each list item. You can specify multiple volume attachments within this section, but for this example we only need the one.

On the left of the colon, we're specifying a named volume, in this case `wp_data`. If you were using a bind mount, you'd include the path on your host here. On the right of the colon, we're specifying where within the container's file system we want to mount this volume, in this case `/var/www/html` (the document root for the Apache installation in the `wordpress` image).

If this looks familiar, it's the same way we ordered things when attaching our named volume to our nginx container in a previous lesson. It's also the same order as you would provide when creating a container with a `docker run` command.

For the database container, we're doing the same thing, but with a volume named `db_data` mounted in `/var/lib/mysql`.

Ports

```
    ports:
      - 8088:80
```

The `ports` option specifies the ports that the service should expose. Like volumes, this is presented in a list format. The value to the left of the colon is the port to expose on the host, and must not be already used by a service. The value to the right of the colon is the port within the container that your service (Apache) is running on, and you would like to be accessible from the host port.

In our example, we've chosen to use `8088` as the host port. If that port is already in use on your host, adjust as needed. Apache uses port `80` by default, so our internal port is `80`.

If you like, you can omit the host port entirely, and Docker will choose a random port to assign on the host. Bear in mind this random port might change if the container is restarted. You can also specify the host IP address as well as the host port, which is useful if you have multiple network interfaces on your host and only want to expose the port on one of them.

For the database service, we don't want to expose any ports at all, so we haven't included this section. Docker's internal networking will allow the WordPress container to talk to the database container without requiring us to expose the port externally.

Restart

```
    restart: always
```

The `restart` setting specifies how the service acts when it is shut down. This also covers how it behaves when the host itself is restarted. The `always` option we use here means the service will always restart itself. Other options include `on-failure`, `unless-stopped`, and no (the default). We've set the same restart option on both services.

Environment

```
    environment:
      WORDPRESS_DB_HOST: db:3306
      WORDPRESS_DB_USER: wordpress
      WORDPRESS_DB_PASSWORD: wordpress
```

The `environment` section allows you to set environment variables that will be configured within the service. This can be formatted as an array (`- key="value"`) or a dictionary (`key: value`), and we've chosen to use the dictionary format in this example. The key is the name of the environment variable, and the value is the value to apply to that environment variable.

In this example, there is functionality built into the WordPress image we're using that can take these specific environment variables in order to configure the database it will use (which we'll be running as a separate service). Note that in the `WORDPRESS_DB_HOST` environment variable we're referencing the name of the database service (db) - Docker provides internal DNS resolutions for containers in the same stack.

We're using similar functionality from the MySQL image we're using for the database service:

```
    environment:
      MYSQL_ROOT_PASSWORD: secure_root_password
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: wordpress
```

Here we're telling MySQL what to set as the root password, and we're also creating a new database, user and password to be used by the WordPress container. Note the username and password values here match those in the WordPress service's environment variables.

This covers the services we need for our stack - WordPress and the associated database. We've also mounted two named volumes - one for each service - at paths within the container's file system. But we haven't yet created those named volumes. That's what the next section is for.

Volumes

Just as the services section lets you create services, the `volumes` section lets you create volumes. This is notably separate from the volumes options within each service, as those

must always reference existing volumes (or bind mounts).

```
volumes:
  wp_data:
  db_data:
```

This may look like nothing, but it is important. Here we define two new named volumes, `wp_data` and `db_data`. However, we don't specify any settings for each volume. This is because we don't actually need to in this instance - the defaults that Docker uses when creating volumes are what we want, so simply defining them as needing to exist is enough to create them when the stack is provisioned.

If we did not have this section, the stack provision would fail, as the services are referencing volumes that do not yet exist. If you had pre-created those named volumes, you can tell Docker that with the `external: true` option in this section. For example:

```
volumes:
  wp_data:
    external: true
  db_data:
    external: true
```

If you're using bind mounts instead of named volumes, you don't need to define them in this volumes section. This section is only for named volumes.

Other sections

This covers everything that makes up our example stack file, but there are other sections that can exist, including networks, configs and secrets (for Swarm clusters), as well as many other options within each section. These are out of scope for this lesson, however. You can find a comprehensive list in the Docker documentation.

CONTINUE

4

Summary

In this lesson we've learned about how we can combine multiple components into a stack file. We've covered:

1. The basics of what a stack is (for both Docker Standalone and Docker Swarm)
2. The structure of stack files
3. Some of the sections and options that can be used in a stack file through an example deployment of WordPress.

Next, we'll use what we've just learned to spin up a WordPress stack in our environment.

Deploy a stack

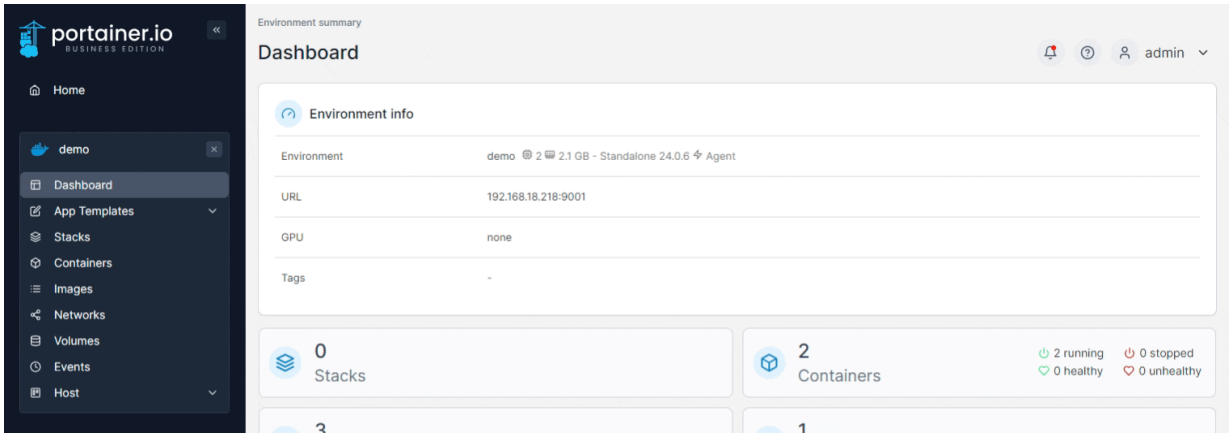
In the previous lesson we introduced the concept of stacks in Docker, and looked at an example stack to deploy WordPress. In this lesson, we'll put that learning into practice. We will:

1. Examine the options for deploying a stack with Portainer
2. Deploy our example WordPress stack to our environment

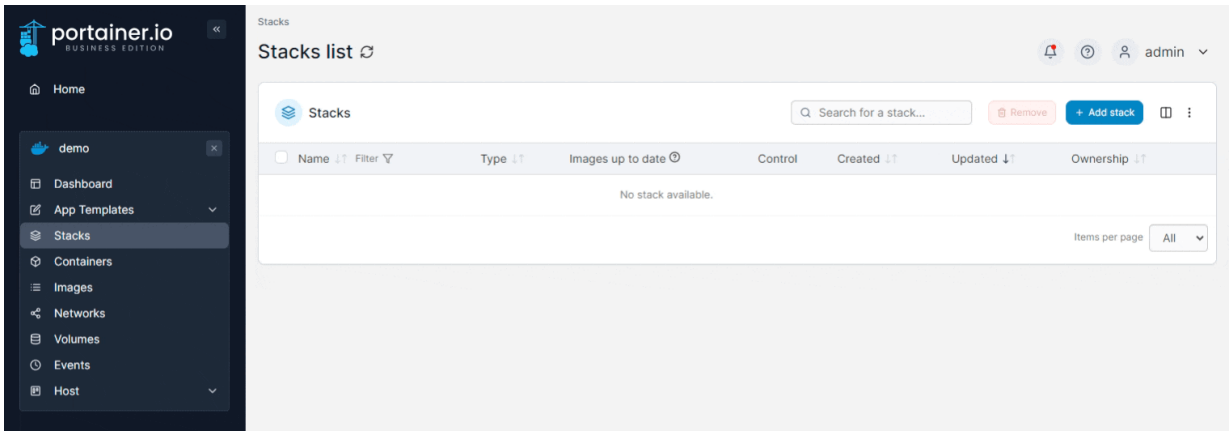
START

Deploying a stack with Portainer

Portainer provides multiple methods for how you can deploy a stack, depending on how you wish to supply your stack file and how you want to manage it. Regardless of which option you choose, the first step is to select your Docker environment, then in the left menu select **Stacks**.



Here you'll see a list of stacks (if any) deployed to your environment. We're going to create a new one, so click the **Add stack** button in the top right.



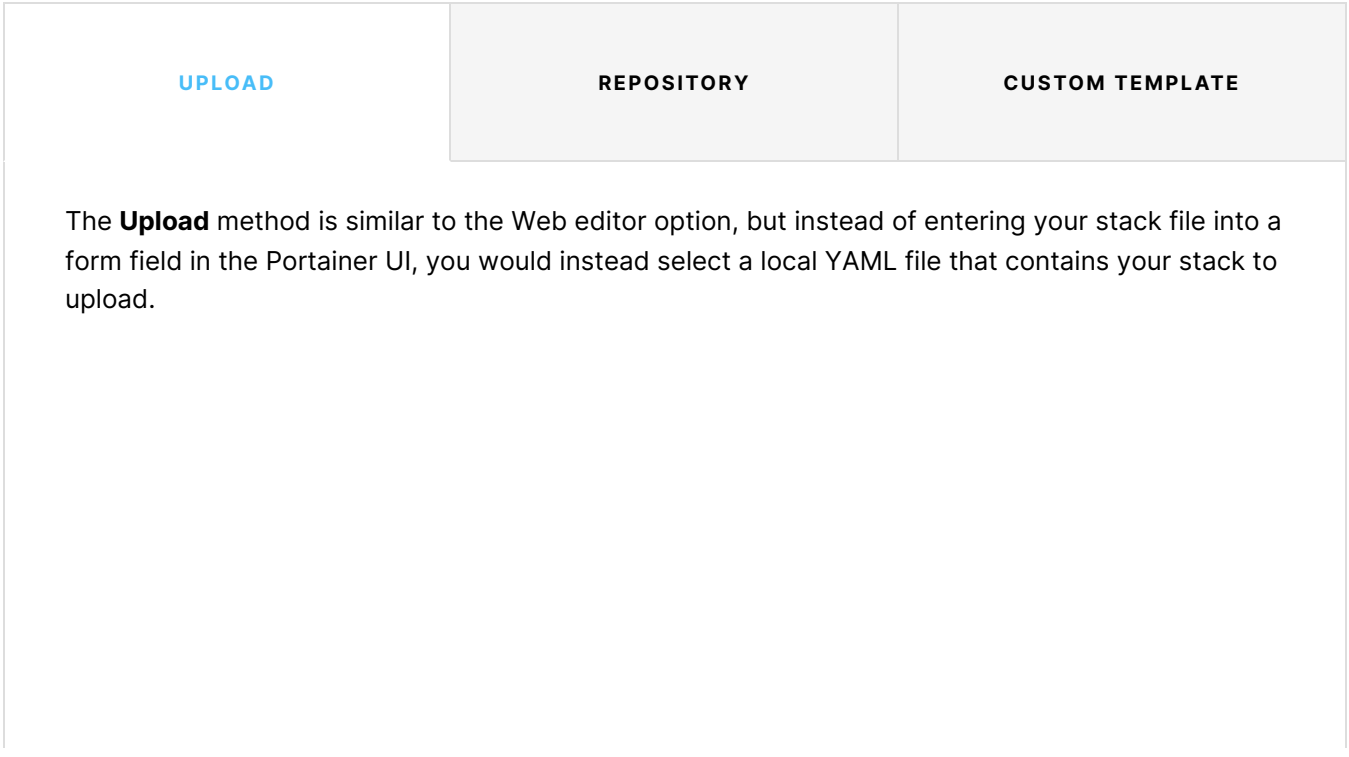
Every stack needs a **name**, so let's start by entering one. Stack names must be unique across your environment. For our example, since we're deploying WordPress, let's call

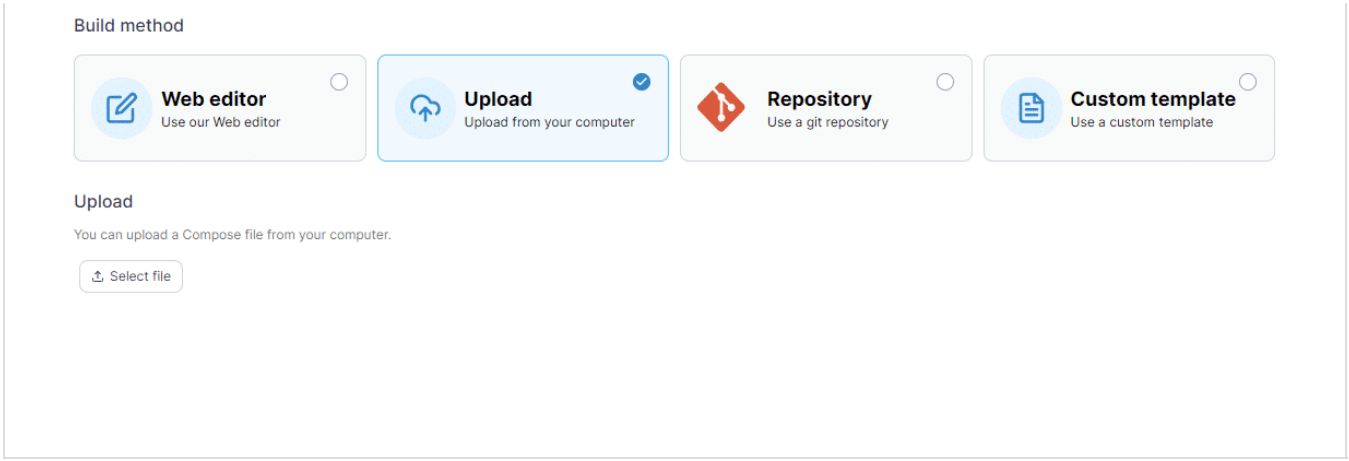
ours `wordpress`.

Next, we choose the build method we want to use for our stack. Portainer provides four options for this:

- Web editor
- Upload
- Repository
- Custom template

For this example we're going to use the Web editor option, but let's first briefly discuss the other options that are available.

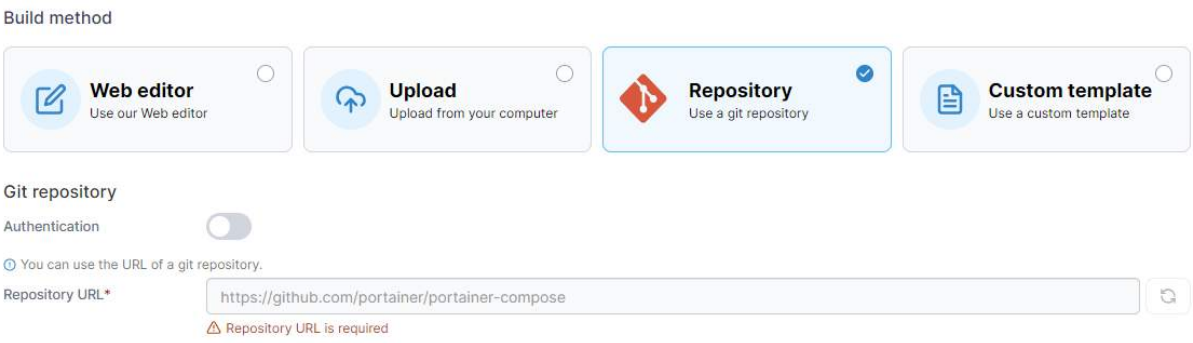


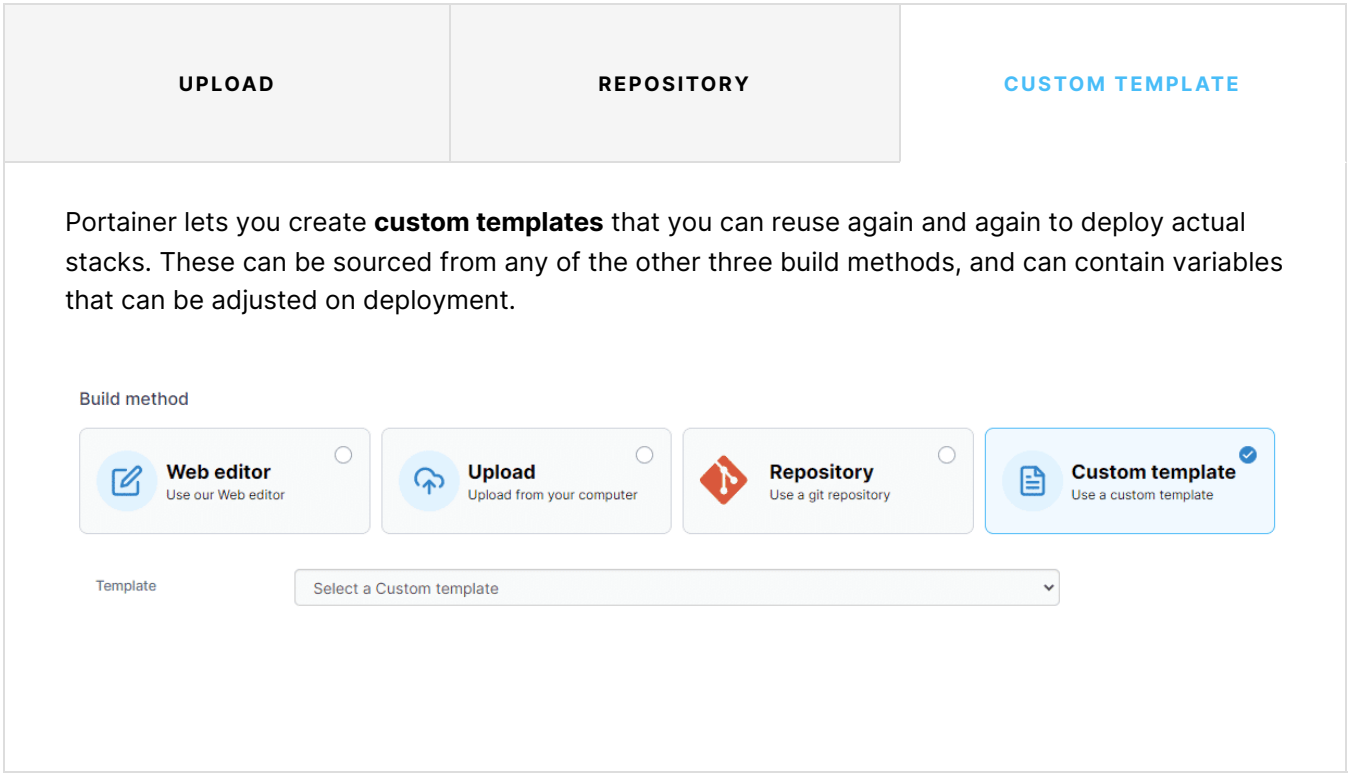


UPLOAD | REPOSITORY | CUSTOM TEMPLATE

If you have your stack file stored in a Git repository, you can use Portainer to deploy directly from there to your environment. This lets you use Git as the "source of truth" for your stack configuration, and with our GitOps update functionality you can automate stack updates based on changes in the Git repo. This is our recommended approach for stack deployment on production systems.

We'll go into much more detail on the **Repository** deployment option in a later course.





CONTINUE

2

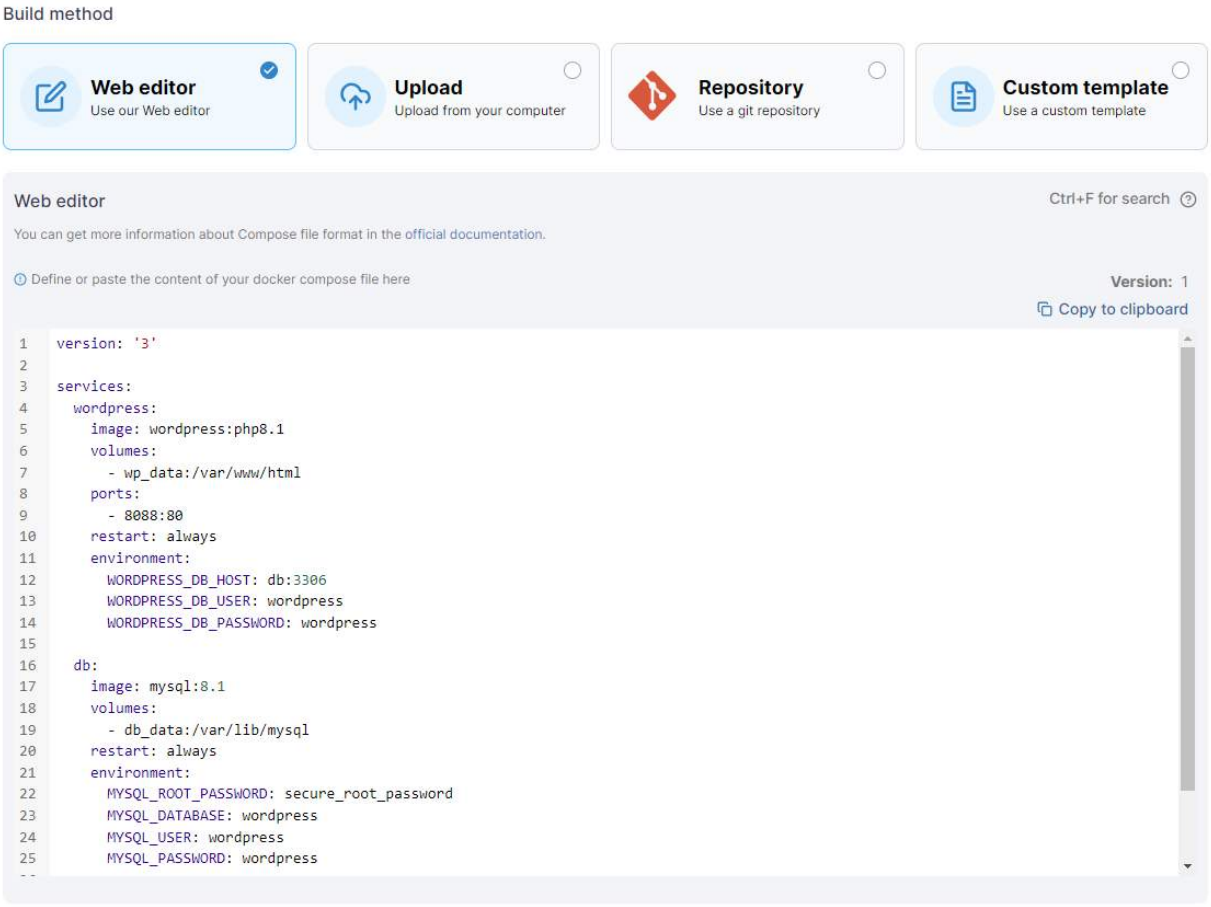
Deploy with the Web editor

In the **Build** method section, select the **Web editor** option. You'll now see a field for you to enter your stack file definition. We're going to use our WordPress stack from the last lesson here - here it is again:

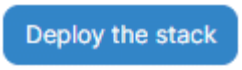
```
version: '3'

services:
  wordpress:
    image: wordpress:php8.1
    volumes:
      - wp_data:/var/www/html
    ports:
      - 8088:80
    restart: always
    environment:
      WORDPRESS_DB_HOST: db:3306
      WORDPRESS_DB_USER: wordpress
      WORDPRESS_DB_PASSWORD: wordpress

  db:
    image: mysql:8.1
    volumes:
      - db_data:/var/lib/mysql
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: secure_root_password
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: wordpress

volumes:
  wp_data:
  db_data:
```

To refresh, this stack file is going to create two containers - a WordPress container and a MySQL database container - and two volumes (`wp_data` and `db_data`) that will provide persistent storage for our two containers.

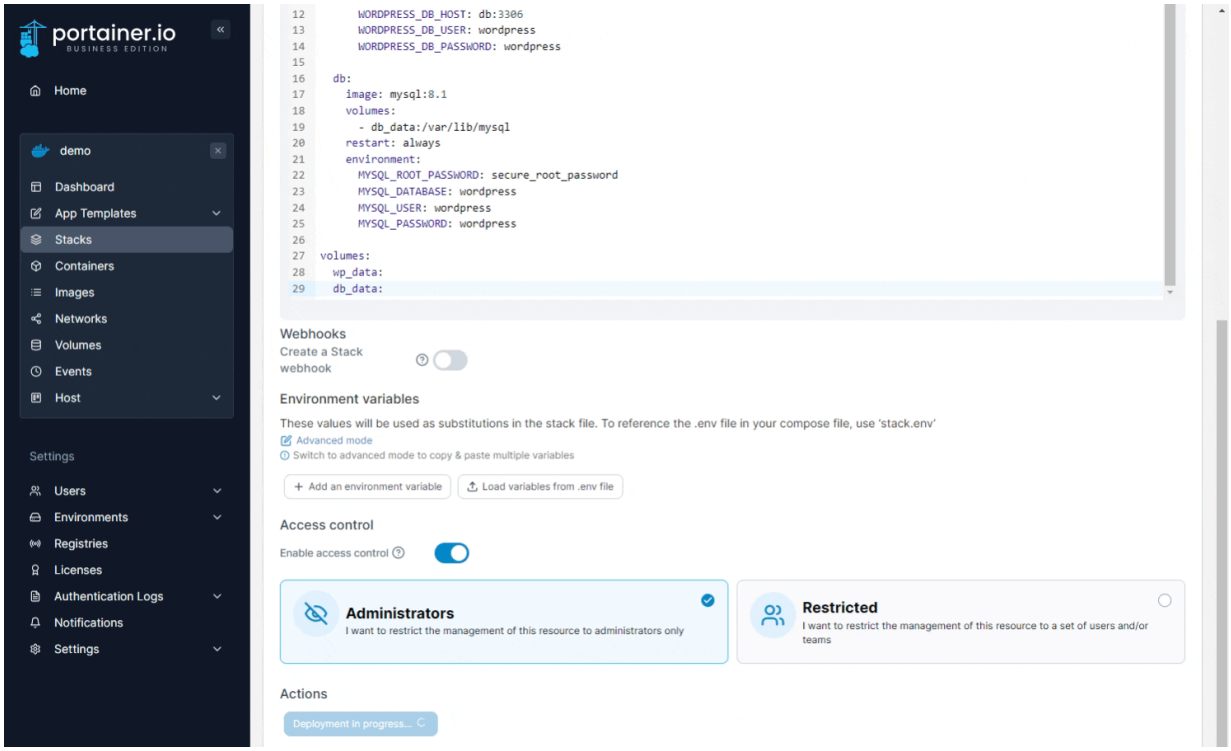
Paste the above code into the web editor form in Portainer. It should end up looking something like this:



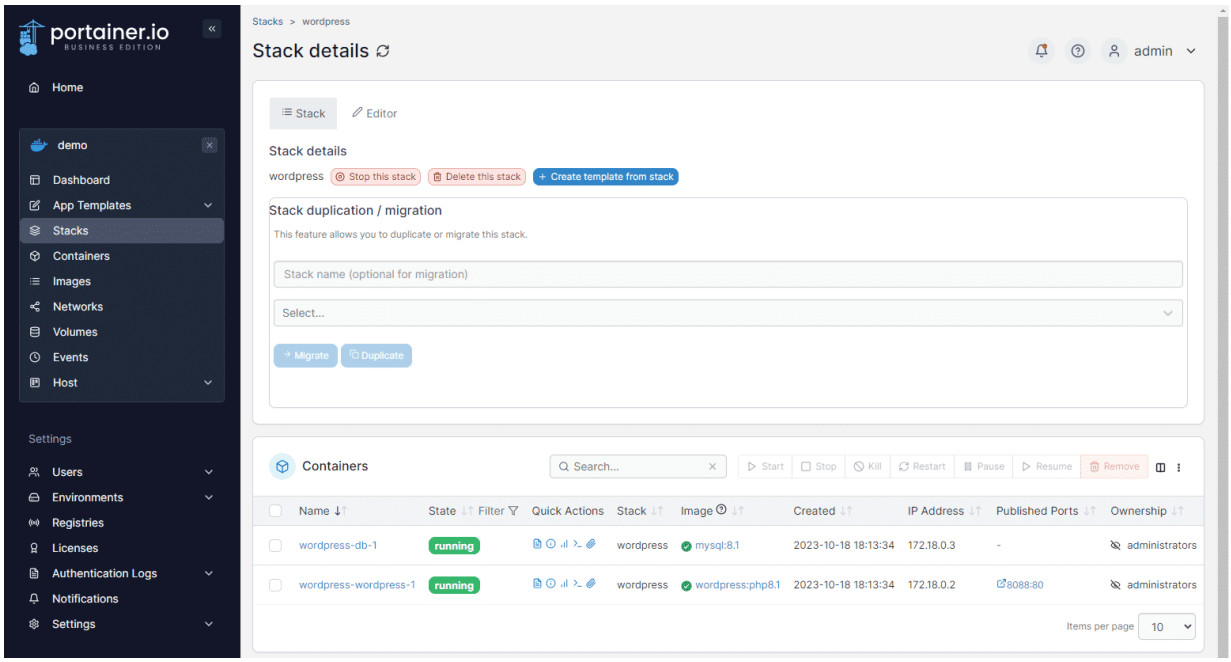
Below the editor you will see additional options for webhooks, environment variables and access control. For this example we won't be using any of those, so you can simply click the **Deploy the stack** button now.



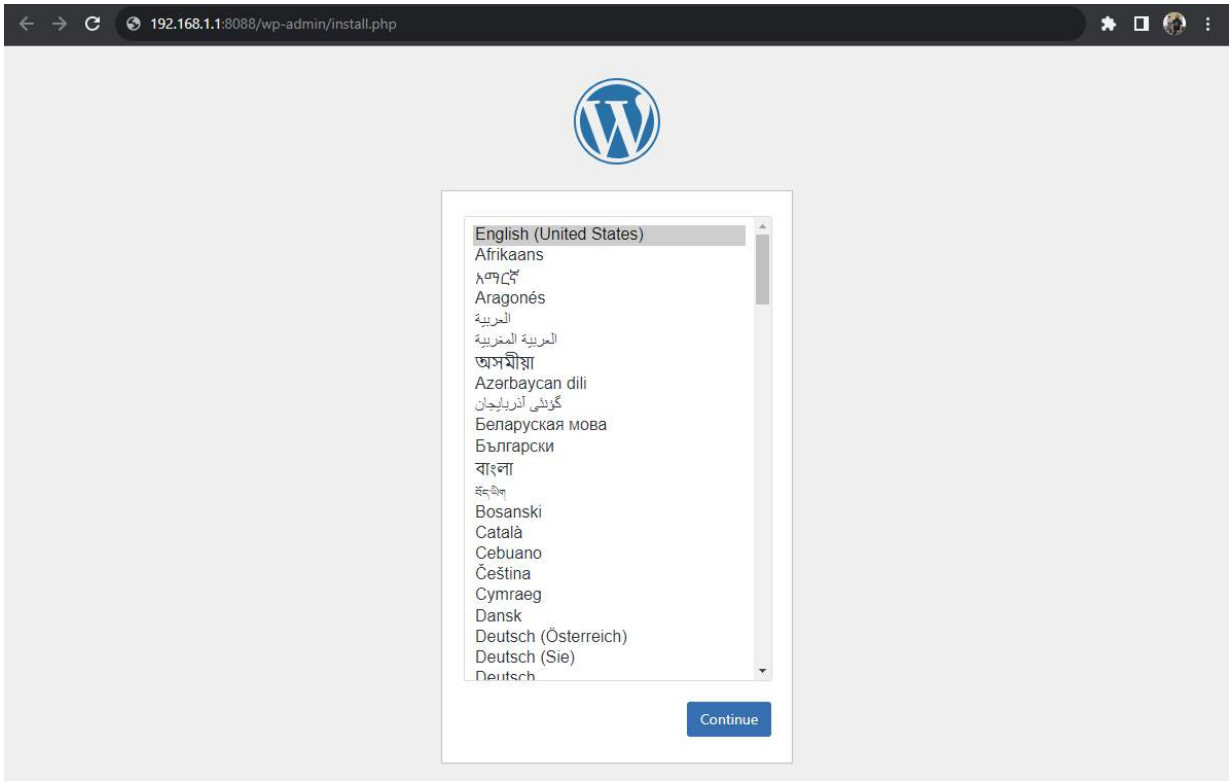
Your stack will now be deployed to your environment, and you'll be returned to your list of stacks.



You can click on the name of your stack to view the stack details, including the containers that make up the stack.



If you open a web browser to the IP address of your environment with the port we exposed for WordPress (8088) we will be presented with the WordPress installation wizard, showing that both the WordPress and MySQL containers have deployed successfully.



CONTINUE

3

Summary

In this lesson we took what we learned in the previous lesson about stacks and deployed a stack in our Docker environment. We covered:

- [ ] The methods you can use to deploy a stack from Portainer
- [ ] Deploying our WordPress stack using the Web editor method

In our next lesson we'll look at Docker Swarm in further detail, in particular around deployment considerations when running a Swarm environment.

What is a Swarm?

In previous lessons in this course, we've introduced the basic building blocks of a Docker deployment; namely containers, volumes and stacks. In this lesson we'll cover how those concepts apply to Docker Swarm. We'll learn:

1. What Docker Swarm is
2. How Docker Swarm differs from Docker Standalone
3. When you may want to use Docker Swarm instead of Docker Standalone

Let's get started then, with learning what Docker Swarm is.

START

Introducing Docker Swarm

Built on the underlying Docker technology, Docker Swarm can be thought of as an orchestration and management layer on top of Docker, specifically designed to allow for managing of and deployment over a cluster of Docker installations. This provides powerful features such as replication and load balancing, but does require multiple interconnected Docker installations across multiple servers, and as such has an additional management overhead to consider.

A Swarm cluster is made up of **nodes**, which are individual Docker installations running in swarm mode. Nodes can be either manager or worker nodes, or both. **Manager nodes** are in charge of allocating tasks to worker nodes based on the parent service's configuration, as well as handling the management of the cluster itself. **Worker nodes** are, as the name suggests, nodes that handle the workload distributed to them by the manager nodes.

A **service** is a definition of tasks that can be deployed on the Swarm cluster. Services contain the image and configuration for a container that will run on the cluster, along with a replication configuration - either the number of replicas to have running at any one time, or "global" mode in which the service is has a single task running on each node in the cluster.

A **task** is the actual container created on a node, defined by the service configuration, and placed according to the replication configuration of the service.

We'll talk about the technical configuration of services in the next lesson.

CONTINUE

2

Docker Swarm vs Docker Standalone

There are important differences to be aware of when it comes to Docker Standalone versus Docker Swarm.

With Docker Standalone, you're limited to deploying on a single Docker installation. There's no facility for replication or scaling across multiple nodes, for either redundancy or performance. With Docker Swarm, you have multiple nodes (servers running Docker) connected together into a cluster of nodes that act as one environment. In this arrangement, when deploying a service you can define the "optimal state" of that service, for example how many replicas of that service you want to be running at one time. Then, if one of the nodes in your cluster goes down, Docker Swarm will recognize the state is no longer optimal and schedules that service's tasks (ie, containers) that were on that missing node onto other nodes in the cluster, in order to re-establish the service's optimal state.

This distributed replica system can also allow you to make configuration changes to services without downtime. Let's say you had a service running 3 replicas across your cluster. You make a configuration change to the service, and Docker Swarm will shut down one of the 3 tasks and deploy the updated version of it. It will then do the same to the second task, and then the third. In this way, the service stays active the entire time it is being updated.

Docker Swarm also includes ingress load balancing in order to provide access to your service regardless of the node it currently resides on. When you publish a port for a

service, that port is available from any node in the cluster on that port. The swarm's internal networking routes the request to the correct node automatically.

On the flipside, to take advantage of these features, Docker Swarm does require you to have multiple Docker installations, which could increase the operating costs of your environment. The number of nodes will depend on your specific requirements, but you do need to consider redundancy of manager nodes as well as worker nodes in order to ensure your cluster can reach a consensus as to the configuration.

You will also need to ensure that each node in the swarm is able to reach and communicate with the others in order to act efficiently. This can be challenging in some networking configurations and with some providers, and adds another layer of complexity to manage.

CONTINUE

3

When would I use Swarm instead of Standalone?

Your specific needs will dictate whether Docker Swarm makes sense for you. If you don't need the replication and redundancy capabilities that swarm mode provides, then you could stick with Docker Standalone. If however these sound like something you do need, then Swarm can help to provide this.

Use Swarm when:	Use Standalone when:
You need to provide replication and/or redundancy to your services	Replication and redundancy are not a concern to you
You want to be able to do no-downtime service updates	You are okay with short downtimes when updating services
You are able to provision multiple servers to make up the nodes for your cluster	You want to keep server costs down or have limited resources
You are comfortable with the management overhead of maintaining multiple Docker servers	You don't want to manage multiple Docker servers

This all sounds like Kubernetes...

If you're familiar with Kubernetes, then yes, a lot of this functionality will sound the same. Kubernetes could be considered the next evolution of the concepts that Docker Swarm popularized. While Kubernetes is extremely powerful and fills a lot of the same needs as Docker Swarm, there is a steeper learning curve when it comes to switching to Kube from Docker, and in some cases moving to Docker Swarm is an easier transition for those used to the Docker way of doing things.

CONTINUE

Summary

This lesson has been about introducing you to the concept of Docker Swarm as compared to Docker Standalone. We've talked about:

- [ ] What Docker Swarm is
- [ ] What the differences are between Docker Standalone and Docker Swarm
- [ ] Some examples of where you might want to use Docker Swarm and where you might want to stick with Docker Standalone

In the next lesson, we'll look at how you would configure a service to take advantage of Docker Swarm's capabilities.

Anatomy of a service

To take advantage of Docker Swarm's orchestration capabilities, instead of deploying containers you should instead be deploying services. This lesson will introduce the concept of services as it pertains to Docker Swarm. We'll examine:

1. What a service is
2. Why you should use a service instead of containers
3. The specific options available when deploying a service

In the following lesson we'll put this knowledge into practice, but for now let's learn what a service is.

START

Introducing Services

As we briefly discussed in the previous lesson, a service in Swarm is a definition of a task to perform. The service defines the image and configuration, and is used to generate tasks (ie, containers) that actually handle the workload.

A key component of services on Swarm is the replication model. This determines how the tasks that the service defines are distributed across the nodes in your Swarm cluster. You can specify how many replicas a service must run as a minimum, and this can be adjusted on the fly as required. Alternatively, you can define a service as global, in which case the service will run a task on each available node in the cluster.

Mechanically, the configuration of a service is very similar to that of a container, in that you define the image to use, ports to open, volumes, networking, and much more. However, you also define the replication model as described above, and must also take into consideration how your application will behave in a replicated or distributed model, both from a processing standpoint and a persistent storage standpoint.

CONTINUE

2

Services versus Containers

The major difference to consider between services and containers is that services are multi-node aware, whereas containers can't be replicated or distributed in the same way. If you want to take advantage of Swarm's multi-node capabilities you will want to deploy your application as a service instead of a container, and define the replication model as

needed. If you just need to run a workload on a single node, you can spin up a container in the same way you would on a Docker Standalone environment.

Bear in mind that your application needs to be architected to work with your chosen replication model. Any persistent storage needs to be accessible from all the nodes you intend to run your service on, and in the same way, which can be complicated to achieve depending on your configuration. If you're looking at running your service on multiple nodes simultaneously (for example for load balancing or redundancy) you'll need to make sure your application is designed for this. For a static HTML website this might be fine, but if your application does any database writes for example, the database will need to be able to handle potential writes from multiple sources.

In most cases if you're looking at replication for your application you'll be aware of the technical considerations of doing so.

CONTINUE

3

Service specific options

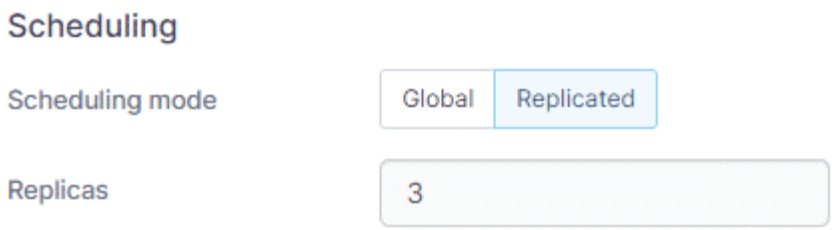
As we've already discussed, most of the options when adding a service will be familiar to you from adding a container, as they're very closely related. But let's look at the options that are specific to services.

Scheduling

This is the primary difference with services as compared to containers, and most other options we cover will stem from this. Your first choice is to choose the scheduling mode - **Global** or **Replicated**.

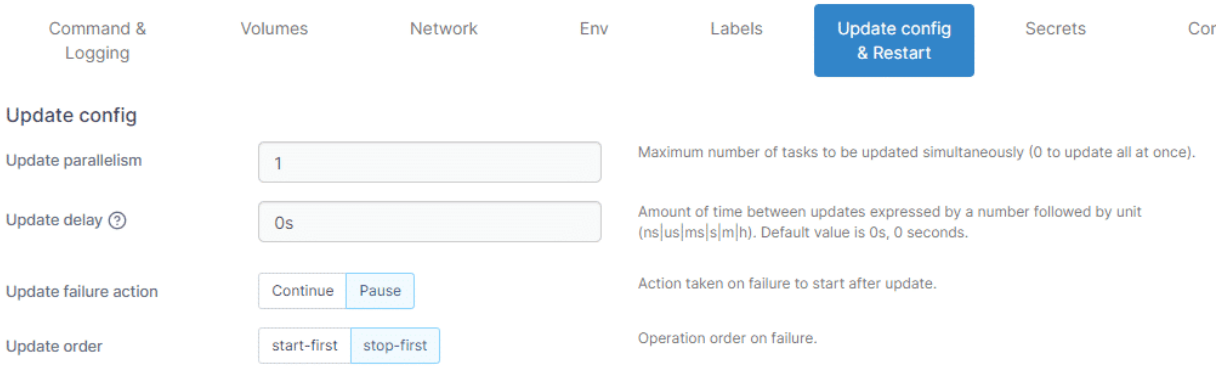
In Global mode, the service will run one task (container) per node in the cluster. So for a 3 node Swarm cluster, your service will run 3 tasks, one on each cluster.

In Replicated mode, you can choose the number of replicas to run. This can be as many as you need, and isn't limited by the number of nodes in your cluster - you could example run 6 replicas on a 3 node cluster, and you would end up with 2 tasks on each node.



Update config and Restart

Here you can configure how updates to your services behave, as well as how they restart. You can define how many tasks to update at once when you update the service. This defaults to one task at a time, but if you're running a large amount of replicas you may want to increase this value. You can also set the delay between each task (or tasks) update, how to behave when an update fails, and the order of operations on the update.



For restarts, you can configure under what condition task restarts occur, as well as delays, max restart attempts and the restart window.

Secrets and Configs

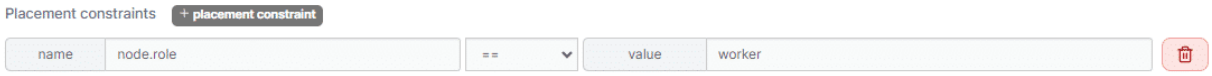
With services on Docker Swarm you can use secrets and configs to provide information to your service (and underlying tasks) outside of the image and persistent storage. This is useful for a configuration that might be specific to the environment you're running on (for example a dev environment versus a production environment), or in the case of secrets, sensitive data (such as a SSL certificate and key) that you don't want to store in the image itself.

We'll cover configs and secrets in more detail in a future lesson.

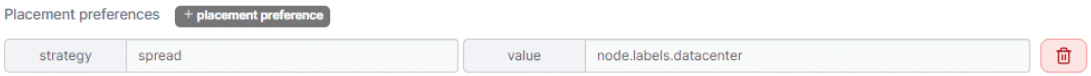
Resources and placement

Here you can configure resource reservations and limits for your service's tasks, for both memory and CPU.

You can also set placement constraints and preferences for your service's tasks here. A **placement constraint** lets you restrict a task to run only on nodes that meet the conditions you specify. This is done through the use of metadata such as labels, both built-in and custom.



A **placement preference** lets you define how you would like to distribute your tasks across the nodes in the cluster. This can be done by specifying the strategy (at the time of writing the only strategy supported by Swarm is the spread strategy) and a label that will be set on the nodes. For example, if each of your nodes had a label called datacenter which defined which datacenter they were in, you could use this to ensure your tasks were spread across nodes that were in different datacenters.



CONTINUE

4

Summary

In this lesson we've looked at what makes up a service, and in particular how services compare to containers. We've looked at:

- [ ] What a service is
- [ ] When you may want to use a service instead of a container, and some of the caveats that need to be considered when doing so
- [ ] The primary configuration differences when creating a service versus a container

In the next lesson we'll put this into practice by creating our own service.

Deploy a service

In the previous lesson we examined the service model in Docker Swarm, including the concept of services and the options specific to service creation. In this lesson we're putting that learning into practice. We will:

1. Create a new service in our Docker Swarm environment
2. Check the deployment status of our service
3. Scale our service to adjust the number of tasks deployed

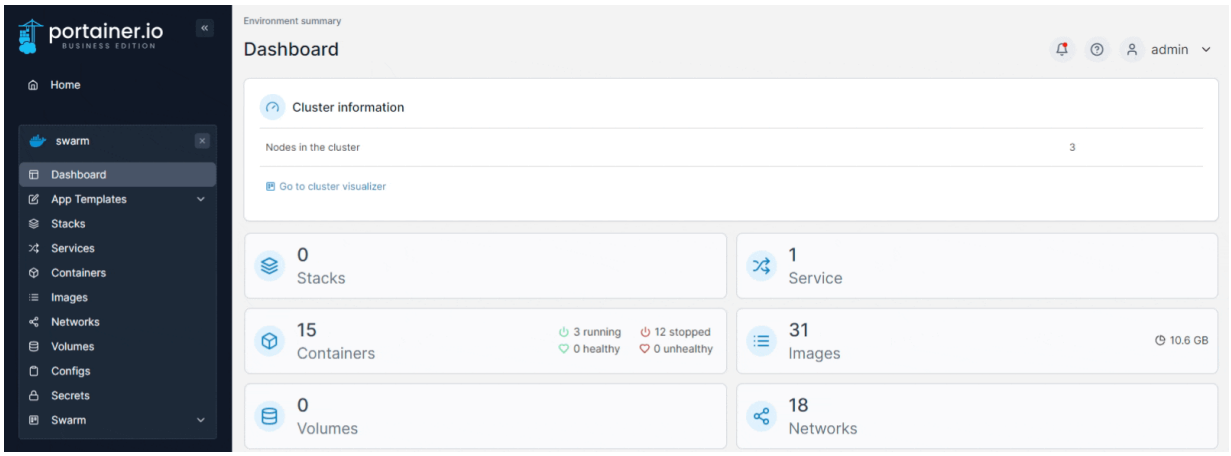
As services are a Docker Swarm feature, this lesson will require you to have a Docker Swarm environment configured in Portainer.

START

Create a service

For this demo we're going to create a nginx service deployed across your Docker Swarm environment.

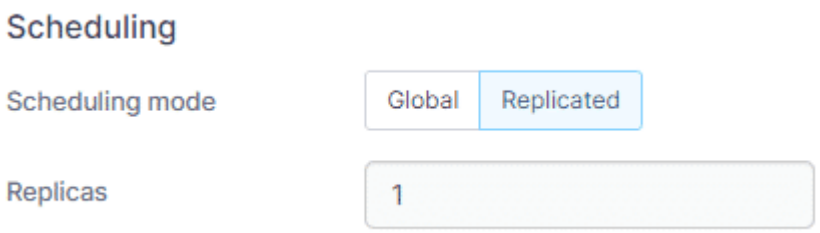
Log into Portainer and select your Docker Swarm environment. From the left hand menu select **Services**. This will take you to the list of services on your environment.



To create a service, click the **Add service** button in the top right. You'll be taken to the Create service form. Enter a name for the service (we'll use `nginx` in this example) and an image and tag (for this example, `nginx:latest`).

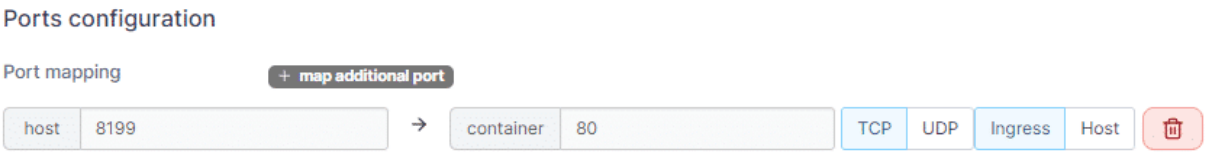
The next section is new to services - **Scheduling**. Here you define the scheduling mode of the service and, in replicated mode, how many replicas each task has.

As we discussed previously, Global mode means the service will deploy a task to every node in the Swarm cluster. In Replicated mode, you define the number of tasks to deploy. For this example, we're going to start with a single replica. Ensure the **scheduling mode** is set to `Replicated` and that **Replicas** is set to 1.

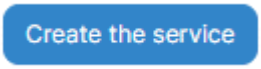


Next we'll publish a port. In the **Ports** configuration section, click **map additional port**. In the fields that appear, set `8199` as the **host** port and `80` as the **container** port. Ensure **TCP** and **Ingress** are selected.

> **If port 8199 is already in use on your cluster, you can change this to a port that suits.**



For this example there's nothing further we need to change from the defaults, so go ahead and click **Create the service** to begin the provision.



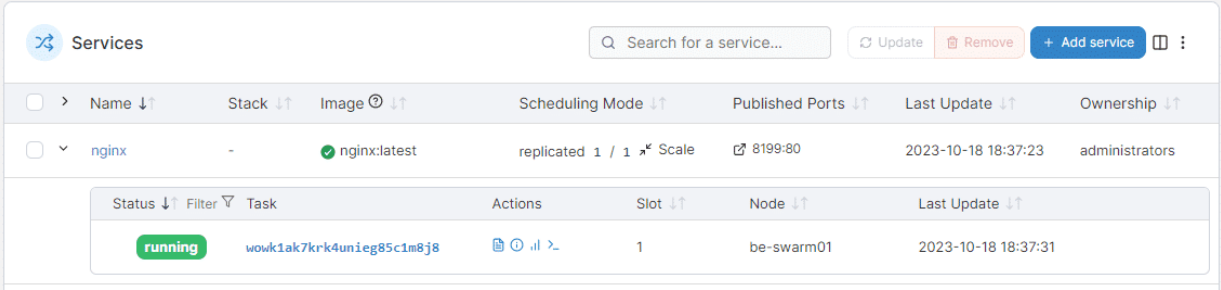
The service will now be created, and the task will be assigned to one of your Swarm nodes.

CONTINUE

2

Check on our service deployment

We now have a nginx service running on our swarm cluster, so let's have a look at it. When the provisioning completed you would have been returned to the services list, so find the `nginx` service there and expand it by clicking on the arrow to the left of the service name. You'll see details of the tasks the service contains - in our case, just the one - including the status, task identifier, and actions you can perform on that task as well as the slot the task inhabits, the node it is deployed to and the last update date and time.



You can check the nginx container is working as expected by opening a new browser tab and going to `http://any_node_ip_address:8199` (replace `any_node_ip_address` with an IP of a node in your cluster) - you should see the default nginx welcome page. Note that this works on any of the node IP addresses in your cluster, regardless of whether the service has a task scheduled there. This is because Docker Swarm's routing mesh automatically sends your request to a node and task that can fulfil it. For example, if I had a 3 node cluster with the following IPs:

```
192.168.1.101
192.168.1.102
192.168.1.103
```

I could reach that single nginx task we've created at any of the following addresses:

```
http://192.168.1.101:8199
http://192.168.1.102:8199
http://192.168.1.103:8199
```

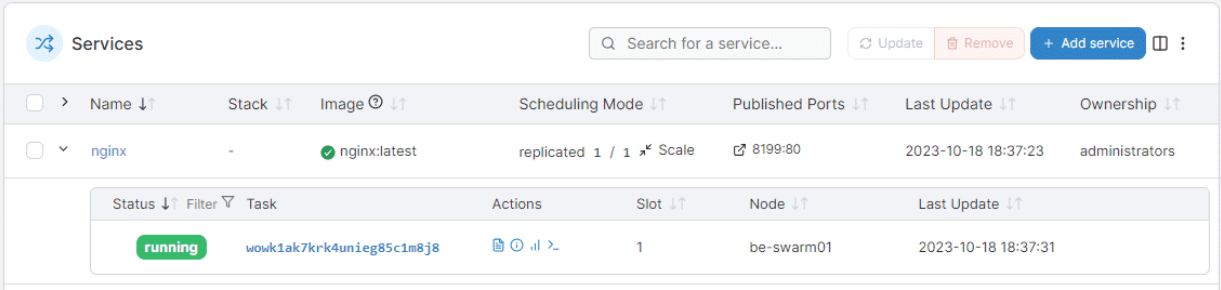
CONTINUE

3

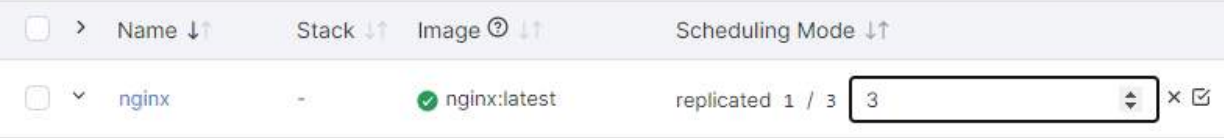
Scaling up our service

Our nginx service is deployed and working, and we've taken a look at it's status. But we're only running one task right now. Let's scale that up.

From the services list, locate the nginx service we created earlier. You'll note under the **Scheduling mode** column it will be listed as `replicated` and with `1/1` replicas. Click the arrow next to the service name to expand it and we will see the list of the service's tasks (just one, right now).

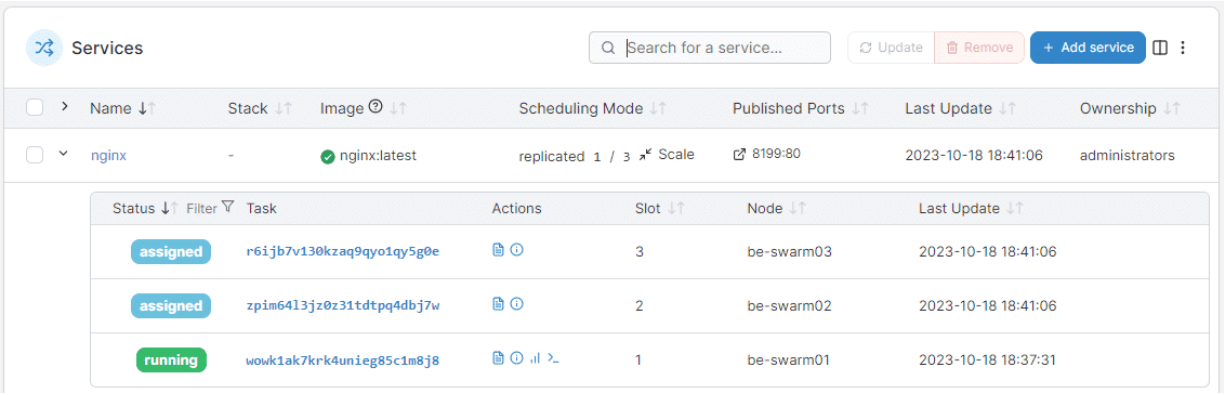


To scale up the service, click on **Scale** next to the replica count. A new box will appear allowing you to adjust the number of desired replicas. Set this to 3, either by typing it in or using the arrows to adjust the number up, then click the tick (or press enter).

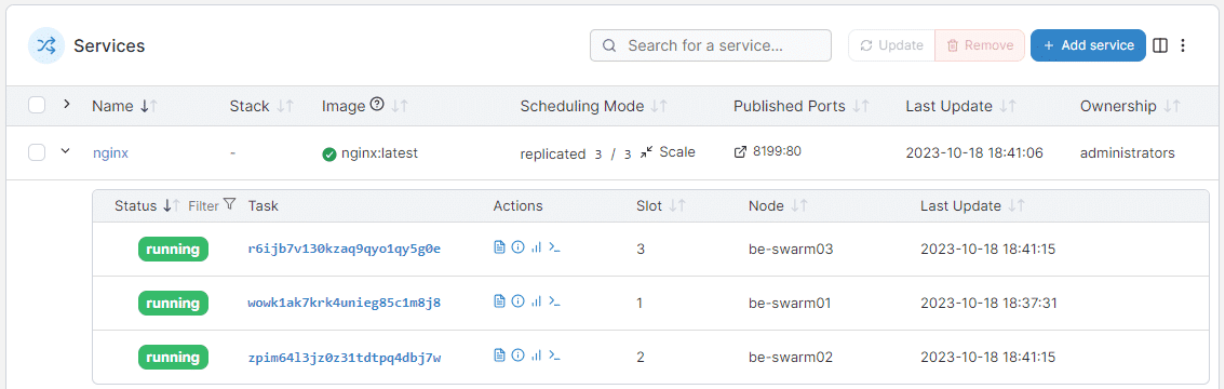


Scaling will now commence on the service. You should see the page refresh, with the task list for the nginx service now containing three items - one being the previously running task, and the other two new tasks that are being spun up.

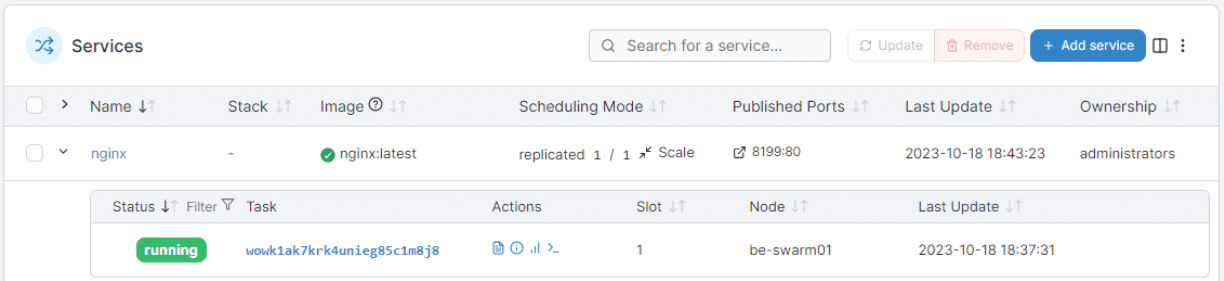
> **If you have 3 or more worker nodes in your Swarm cluster, these tasks should each be assigned to different nodes. If you have less than 3 worker nodes in your cluster, you'll see multiple tasks on some nodes.**



Initially these will be in `assigned` status. If you refresh the service list you may see these tasks going through various states until they also reach `running` status alongside the existing task. Once this completes, you'll have 3 tasks running for your nginx service. Now if one of your nodes was to go offline, the other tasks would still be able to provide the nginx service to users.



We can scale in reverse as well. Let's take our nginx service back down to a single replica. Again, click on **Scale**, and change the value from 3 to 1. Click the tick to apply the scaling. Swarm will now scale down the service to a single replica, shutting down two of the tasks to do so. If you refresh the service list, you'll see the tasks shut down and then disappear from the list.



CONTINUE

4

Summary

Congratulations, you've provisioned and scaled a service on Docker Swarm! This lesson covered:

- [ ] Creating a new nginx service
- [ ] Checking the status of the nginx service after deployment, and how swarm routing works to provide access to your service's tasks no matter what node they're on
- [ ] Scaling up (and down) our service

Next we'll summarize everything we've talked about in this course.

Summary

Congratulations, you've now completed the Deploying on Docker with Portainer course!

In this course we've covered a lot of ground, including:

- [ ] An introduction to containers - the concept, creation and editing through Portainer.
- [ ] Volumes - what they are, the differences between bind mounts and named volumes, and how to create and attach named volumes to containers.
- [ ] How to combine what we learned about containers and volumes into stacks for better organization and smoother deployments.
- [ ] Docker Swarm - what it is, how it differs from Docker Standalone, and when you'd use it.
- [ ] Swarm's big point of difference: services - including how they work, how to create them, and how to scale service deployments.

This is by no means an exhaustive exploration of deploying with Docker - there are many more configuration options and methods you can use. Some of these we will be

discussing in other courses. However, you should now be familiar with the concepts of deployments on Docker and be able to begin your deployment journey.