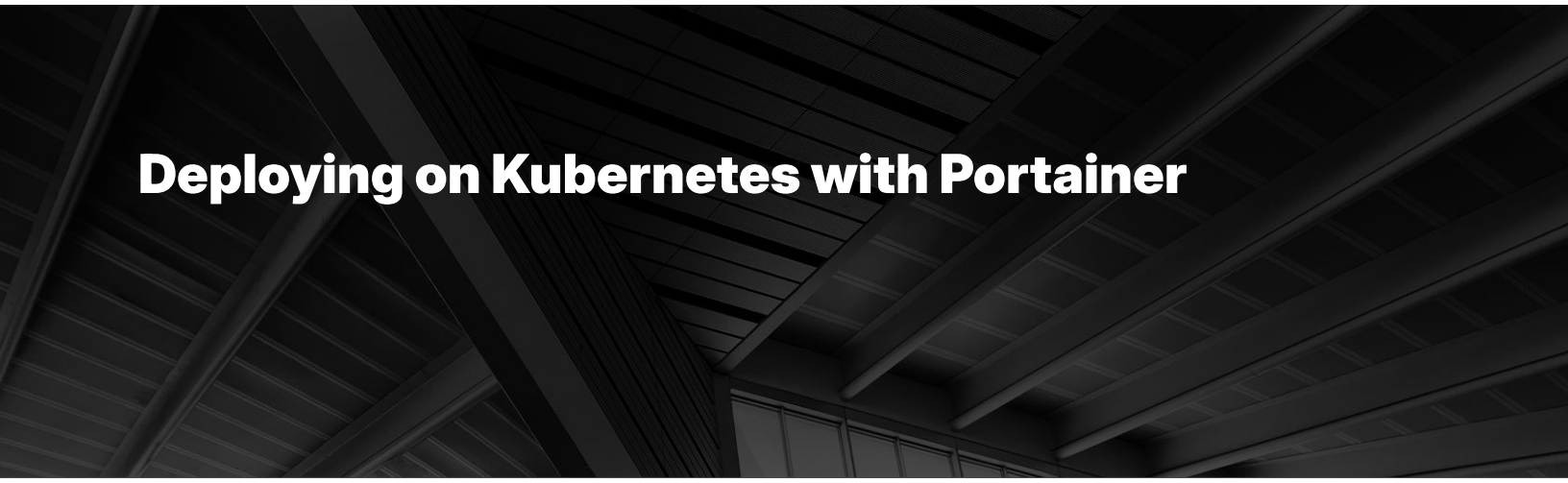


# Deploying on Kubernetes with Portainer

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


Introduction

INTRO TO KUBERNETES

What is Kubernetes?

Anatomy of a Kubernetes cluster

Objects in Kubernetes

WORKING WITH APPLICATIONS

Deploy your first application

Editing your application

Deploying via manifest

SUMMARY

Summary

# Introduction

## Welcome!

Thank you for starting the Deploying on Kubernetes with Portainer course!

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

1. An introduction to Kubernetes itself, including when to use it, what elements make up a cluster, and a look at the objects you can deploy on Kubernetes.
2. Creating an application with Portainer using our form-based approach, as well as editing after deployment.
3. Creating an application with Portainer when you have an existing manifest, and again, editing after deployment.

By the end of this course you should have a good understanding of deploying on Kubernetes 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.

## What you should know before you start

In this course we will give you an introduction to Kubernetes itself, but we do make the assumption that you have an understanding of the basic concepts behind containerization. If you've used Docker before and have a good feeling for how it works, then you should be fine. If you're brand new to containerization, we recommend starting with our Deploying on Docker with Portainer course first, in particular the Anatomy of a container lesson.

Now that the formalities are out of the way, let's get started with an introduction to Kubernetes.

# What is Kubernetes?

At this point in your journey, you know what containerization is, and may be familiar with technologies like Docker that provide this. But how does Kubernetes fit in to the ecosystem?

In this lesson we'll cover:

1. An introduction to orchestration and how Kubernetes implements it
2. How Kubernetes differs from other orchestrators
3. When you might (and might not) want to use Kubernetes in your organization

START

# Orchestration and Kubernetes

A common term used to describe Kubernetes is as a **container orchestrator**. In a simple container system, such as Docker Standalone, you can spin up a container for your workload based on an image that will provide you with what you need for your particular task, nothing more. When you're done with it, you can remove that container.

What Docker Standalone doesn't provide however is a mechanism to automatically dictate what goes on with your deployments. Say, for example, your container crashed. With Docker Standalone, you would have to first notice the container had crashed, then manually recreate it again to get back up and running. What if, then, the entire node went down? How do you recover from that scenario - manually spin up a new node, create your containers, and hope for the best?

An orchestrator such as Kubernetes can help you to take care of these scenarios automatically for you, no user interaction required. You as the user would define the state in which you want your deployment to be, and Kubernetes would then use that to create the necessary containers and configurations to meet your definition. If something happens to those containers, Kubernetes will recognize that the deployment no longer matches the state you defined and will act accordingly. For example, if a container crashes, Kubernetes would automatically provision another one to replace it. If a whole node dies, Kubernetes would reallocate the workloads onto a still running node, making sure your application continues to function. We refer to this as **fault tolerance**.

Kubernetes implements orchestration in a **declarative** nature - as described above, you define the state you want and Kubernetes' job is to ensure that state is met, and retained. You can adjust this definition as desired (for example, scale up or down a particular service), and Kubernetes will apply the necessary changes to the deployment as a result.

CONTINUE

## Kubernetes vs other orchestrators

When we're looking at container orchestration platforms, there are two main players - Kubernetes, and Docker Swarm.

Docker Swarm, as the name implies, is an extension of the Docker ecosystem, and provides orchestration capabilities on top of the base Docker engine. Docker Swarm gives you the standard orchestration functionality you'd expect including fault tolerance, load balancing, and service management and scaling. It is fairly straightforward to configure, as it essentially combines multiple Docker Standalone instances into one Swarm. However, it does have some limitations - there is no automatic scaling functionality in Docker Swarm, and no storage management. It is also less extensible than Kubernetes, and is tied to the Docker platform.

Kubernetes on the other hand is a more powerful orchestrator that was designed for the needs of enterprise customers. As such, Kubernetes provides the standard fault tolerance, load balancing and scaling you'd expect but also provides self-healing and automatic scaling functionality. Kubernetes provides advanced access control functionality as compared to Docker Swarm. Storage for your containers is managed through Kubernetes' storage class system, and there are a number of third party plugins that provide additional functionality as needed such as monitoring and alerting, storage and networking options, and more.

There is a tradeoff here though, in that the additional power that Kubernetes provides comes with increased complexity. A Docker Swarm setup can be easier to deploy and manage than a Kubernetes cluster, and some of the Kubernetes concepts can take a bit of time to learn.

CONTINUE

3

## When to choose Kubernetes

Kubernetes is powerful but complex, and while it is extremely flexible in the workloads it can support and manage, there may be some situations when a different system is preferable.

If you need fault tolerance for your workloads - if they need to stay running and performant no matter what, and you want that to be taken care of automatically - then Kubernetes is a good fit. If you like the sound of defining a "desired" state and letting the orchestrator take care of the implementation, then Kubernetes will work for you. Kubernetes is generally recommended when you are running an enterprise-level solution and need the robustness to keep that solution functioning.

If fault tolerance isn't a concern for you, and you just want to get your container up and running, then you might not need the full power and complexity of Kubernetes. For local development or testing environments, or small home lab deployments, Docker Standalone or Docker Swarm might be a better bet.

Your specific requirements will be unique to you, and we'll leave the decision in your hands. This course assumes that you have decided to continue with Kubernetes.

CONTINUE

## Summary

In this lesson we've discussed the basics of Kubernetes. We've looked at:

- [ ] The concept of orchestration, and how Kubernetes applies to it.
- [ ] How Kubernetes compares to other orchestrators, primarily Docker Swarm.
- [ ] Some examples of when you might want to use Kubernetes, and when you might not.

In the next lesson we'll look at what components go together to make up a Kubernetes cluster.

# Anatomy of a Kubernetes cluster

To enable the fault tolerance and load balancing functionality that Kubernetes provides, a Kubernetes cluster is made up of a number of different components. In this lesson we'll give you an overview of those components. We'll discuss:

1. The two different types of nodes that make up a Kubernetes cluster
2. The components that run on each type of node
3. How those components work together to provide the cluster capabilities

Let's start with an introduction to nodes.

START

## Nodes in Kubernetes: Control Planes and Workers

The two types of nodes that make up a Kubernetes cluster are **control plane nodes** (also referred to as master nodes) and **worker nodes**. The names give you a good idea of what each is for, but:

- A **control plane** node is in charge of managing the cluster - it contains the API server, scheduler, controller manager and the etcd datastore. A cluster will have at least one control plane node, and depending on the size of your environment perhaps more.
- A **worker** node is where the workloads are allocated - in other words, where your applications will actually run. Most of a worker node's resources are intended to be available for your applications, so the only components that run here are the kubelet (for communication with the control plane) and the kube-proxy (which allows communication between containers across nodes).

A node can be both a control plane and a worker node, but in most production environments it is advisable to separate the responsibilities.

CONTINUE

2

## Control plane components

Let's talk briefly about those control plane components we mentioned earlier.

- The **API server** is what you're interacting with when you request something of Kubernetes. If you want to spin up a deployment, you send it to the API server which will validate your request and act accordingly.
- The **scheduler** is the component that determines where the objects that make up your deployments are, well, deployed. The scheduler checks with the API server whether there are any objects that haven't been assigned to a node (based on the desired state). If there are, the scheduler decides which nodes to deploy those objects to.
- The **controller manager** is what checks to see whether the current state of the cluster (and the objects within) matches the desired state. If it doesn't, it will tell the API server that it needs to make a change in order to reach that desired state. The controller manager is made up of a collection of individual controllers for each object type.
- Lastly, the **etcd** datastore is a key-value store that contains a record of the desired state of the cluster and objects. Whenever the API server makes a check or a change, that data is stored and persisted in the etcd datastore.

## Worker components

Besides your actual workload, worker nodes have only two system components:

- The **kubelet** is an interface between the individual nodes and the control plane. It checks in with the API server on the control plane to see whether there are any containers that are assigned to its node. If there are, it spins up the necessary containers to fill that request.

- The **kube-proxy** provides inter-node networking for containers, so that containers on different nodes can communicate with each other. The kube-proxy is responsible for directing traffic to the correct container on the node.

CONTINUE

3

## An example of component communication

We've described above some of the ways that the components of a cluster interact, but let's look at it with an example. Let's imagine you're deploying your application to your Kubernetes cluster.

1. First, you send your deployment to the **API server** component. You could do this through the Portainer interface (more on that in future lessons) or through the kubectl command line utility.

2. The API server would then validate your deployment to make sure it makes syntactical sense, and then update the **etcd datastore** with the new desired state.

3. The **scheduler** checks in with the API server to see if there are any objects that it needs to assign to a node. In the case of a new deployment this will definitely be the case. The API server reports back to the scheduler with the unassigned objects (based on what's

in etcd). The scheduler then assigns objects to nodes as required, and reports this back to the API server, which updates etcd.

4. The **controller manager** checks in with the API server comparing the current state to the desired state (which the API server pulls from etcd). As the current state does not match because of the new deployment, the controller manager requests the changes be applied via the API server, which updates etcd.

5. The **kubelets** on the individual nodes check in with the API server to see what they should be running. The API server returns a list to the kubelets, which then spin up the necessary objects on their nodes. The list the API server provides to the kubelets is based on what is in etcd, which has been updated by the scheduler to determine which nodes to run each object on, and has been updated by the controller manager, which flagged that the required objects were not running and requested the creation of the objects.

As you can see, the individual components work together to ensure that your deployment is provisioned as you requested, and that it remains as you requested. If one of the containers that make up your deployment went down:

1. The **controller manager** would notice this and request that the API server resolve the issue by creating a new object (in this case, a container).

2. The **scheduler** would then see that request and determine which node to assign the new container to.

3. The **kubelet** on the assigned node would see the new object request and spin up the container.

Similarly, if you made a change to your deployment:

1. The **controller manager** would notice the actual state no longer matches the new desired state, and would ask the API server to resolve this.
2. If the change requires that a new object be created, then the **scheduler** would determine the node to place it on.
3. The **kubelet** on the assigned node would see the new object request and create it as needed.

CONTINUE

4

## Summary

In this lesson we've examined what makes up a Kubernetes cluster. We've discussed:

- [ ] The two different types of nodes: control plane and worker nodes.
- [ ] The components that each node type runs.
- [ ] The way that each of these components interact with each other to deploy and maintain your workloads.

Now that we have an understanding of the underlying infrastructure of Kubernetes, we can look at what kinds of objects we can deploy on that infrastructure.

# Objects in Kubernetes

In the previous lesson we made reference to **objects** within Kubernetes. In this lesson we'll dive into the concept of objects, including:

1. What an object is in Kubernetes
2. Some of the main object types
3. How those objects cooperate to form your application

START

1

## What is an object?

When you create a workload in Kubernetes, you are doing so by defining a desired state. This desired state is made up of a collection of components that we refer to as objects.

Each object type has a different role, and as such provides a different piece of the puzzle that completes your workload.

Some objects are obvious, such as containers and volumes, but there are others such as Pods, Deployments, Services and Ingresses that might not be as straightforward a concept. Kubernetes is also extensible, so what objects can do can be expanded beyond the built-in functionality with third-party addons.

Objects are created within Kubernetes through the use of manifests - usually YAML but sometimes JSON files that contain the necessary definitions to specify the desired state of your workload. In Portainer we support the creation of workloads through manifests as well as through a form-based interface if you prefer. We'll show some examples of YAML manifests later in this lesson.

Let's dive into some of the more common objects that you'll be using with your workloads.

CONTINUE

2

## Common Kubernetes objects

There are many object types in Kubernetes, but let's have a look at the ones you're most likely to run into when creating and managing your workloads, starting with arguably the most important object of them all - the **Pod**.

## Pods

A Pod in Kubernetes is a group of one or more containers that share their storage and network resources between themselves. The containers in a Pod will always be together on the same node so that they can share those resources, and when there is more than one container in a Pod they will be very closely related to each other.

Like the container concepts discussed in our Docker course, Pods are designed to be ephemeral - that is, if they go away, they can be easily replaced without any loss of data.

Pods are where the real work happens, and the rest of Kubernetes is built around providing access to and management of your Pods. Here's an example manifest that would create a Pod consisting of a single `nginx` container:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:latest
    ports:
    - containerPort: 80
```

The `metadata` section is where we define the name of the Pod. The `spec` section is where we specify the configuration for the container (or containers) that make up the Pod. You can see we've called the Pod `nginx` and the container will use the `nginx:latest` image and expose port `80`.

However, in most cases you wouldn't create a Pod directly - instead, you'd do so with a different object such as a **Deployment**.

## Deployments

With a Deployment object, we start to get into the declarative nature of Kubernetes. Within a Deployment, you define the desired state of your Pods. This is what the controller-manager uses to determine what objects need to exist.

A Deployment lets you set the template you want to use for your Pods as well as the number of replicas desired for those Pods. The template contains the settings you want to use to create the containers that make up the Pods, such as the image to use, ports to expose, and labels to apply. The replica setting defines how many instances of those Pods you want to create across your cluster. A Deployment also lets you define a selector, which is how you associate Pods with Deployments. Often this is done using labels.

Here's an example Deployment that will create three `nginx` Pods, using the example we provided before:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
```

```
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

Like before, we use the `metadata` section to name the Deployment. We're also attaching a `label` here. In the `spec` we're specifying the number of `replicas` as 3. In the `selector` section we're telling Kubernetes that we want to associate any Pods that have a label called `app` with the value of `nginx` to this Deployment. Then in the `template` section, we set the `metadata` on each Pod to contain an `app` label with the value `nginx` to complete that selector. The `template spec`, as before, defines the container settings.

There's a number of other settings that go into a Deployment such as resource limits, update and rollout strategies, and more, but for now let's move on to the next object type: the **Service**.

## Services

When a Pod is created in Kubernetes it is assigned a unique internal IP address that it is available on throughout the cluster. However, with the ephemeral nature of Pods, this is not a reliable way to access your workload as any of those Pods could go away, either due to a temporary failure or a reconfiguration of the Deployment. To provide this access to the Pods, you can use a Service object.

Think of Services like a network router that knows about which Pods are available and sends requests to those Pods accordingly. Using labels, you define which Pods the Service will provide access to. The Service will constantly check for Pods that match the selection criteria and update routing as necessary.

Here's an example Service definition that we could use alongside our `nginx` Deployment from above:

```
apiVersion: v1
kind: Service
metadata:
  name: my-nginx-service
  labels:
    app: nginx
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

In the spec we are defining a `selector` that works the same as our Deployment matching, looking for Pods with a label named `app` and a value of `nginx`. In the `ports` section we're defining that we want to provide access to the Pods on `port 80` with the TCP `protocol`. The `targetPort` is optional, but is handy if your Pods are exposed on a different port to the one you want to publish.

Services (using the default ClusterIP type) use a virtual IP to publish your ports internally within your Kubernetes cluster. If you want to provide access to your workload from an external location, you can use an **Ingress**.

## Ingresses

The Ingress object provides you with a way to make your workload accessible outside of your Kubernetes cluster. An Ingress will accept traffic from an external source and route it to the specified Service.

**Ingresses are generally restricted to HTTP and HTTPS traffic to a ClusterIP Service - for other traffic types you may want to look at alternative Service types such as NodePort or LoadBalancer.**

Separating Services and Ingresses means that you can limit external access to only the Services you want - for example, if your workload contained a web server and a database server, you would only want to provide external access to the web server and not to the database server. To achieve this you would configure your Ingress to only route to the web server's Service and not the database server's Service.

Let's say we wanted to make our `nginx` Deployment available externally through an Ingress. We created the necessary Service, so now we create the Ingress:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-nginx-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  ingressClassName: nginx
  rules:
  - http:
      paths:
```

```
      - path: /app
        pathType: Prefix
        backend:
          service:
            name: my-nginx-service
            port:
              number: 80
```

Ingresses are often configured using `annotations`, and in this example we're setting the `rewrite-target` to `/`. In the spec you can see we've specified the `ingressClassName` as `nginx` (more on this below), and then can define `rules` in order to specify the routing. We've only got the one rule here, called `http`, and it defines that any URL with the `/app` prefix (for example, `http://my-server/app/hello.html`) will be routed to the `my-nginx-service` Service we created earlier, on `port 80`.

There are a few caveats to using Ingresses. First, an Ingress Controller must exist for the Ingress to work. There are many different types of Ingress Controller you can configure, but for this example we're using the ingress-nginx controller. You also need an Ingress Class configured, which is what defines the settings for the Ingress Controller. In this example, we assume you have both of these set up, and your Ingress Class is named nginx. Configuring Ingress Controllers and Ingress Classes is outside the scope of this particular introductory lesson.

## Other Kubernetes objects

The four object types we've covered are some of the most important to understand, as they demonstrate how the Kubernetes systems work together. There are many other object types, such as Jobs for run-once actions, Volumes for storage, ConfigMaps and Secrets for providing configuration data and sensitive information respectively, and Namespaces for providing segregation between objects and resources. We'll cover some of these in future lessons.

CONTINUE

3

## Putting it all together

For a fully fledged workload, you will invariably need multiple objects. We've covered how the objects interact in the previous section, and using the example Deployment, Service and Ingress manifests we can build a combined workload definition to create them all at once.

**You can combine multiple object definitions in YAML by separating each one with three hyphens: ---**

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
```

```
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: my-nginx-service
  labels:
    app: nginx
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-nginx-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  ingressClassName: nginx
  rules:
  - http:
      paths:
      - path: /app
        pathType: Prefix
        backend:
          service:
            name: my-nginx-service
            port:
              number: 80
```

The above will create the Deployment and its three nginx Pods, the Service to provide access to the Pods internally, and the Ingress to provide external access to the Pods via

the Service.

CONTINUE

4

## Summary

In this lesson we've talked about objects in Kubernetes. We've covered:

- [ ] The basic concept of an object in Kubernetes.
- [ ] Some of the most common object types - Pods, Deployments, Services and Ingresses, and how they're defined in manifests.
- [ ] How we can combine multiple object definitions and deploy them at once.

Next we'll put that into practice and deploy an application.

# Deploy your first application

In the previous section we introduced the concepts behind Kubernetes from a cluster standpoint as well as discussed the objects that make up an application deployment. In this lesson we'll put those concepts into practice by creating an application in Portainer using the form creation method.

## Before we start: form vs manifest

Portainer provides two primary methods to create an application - through a form-based approach and via a pre-created manifest. This lesson will cover the form-based approach.

Each creation method has benefits and drawbacks. The form-based approach is a handy way to get an application up and running when you don't have an existing configuration to work from, or are not familiar with writing YAML manifests. By it's nature, a form-based approach doesn't provide as much configuration flexibility as a manifest, so if you need that additional configurability, or already have a manifest written for your application, the manifest approach may suit better.

For now, let's dive into creating an application through a form.

**In order to deploy an application to a Kubernetes cluster, you must first have a cluster. This lesson assumes you have your**

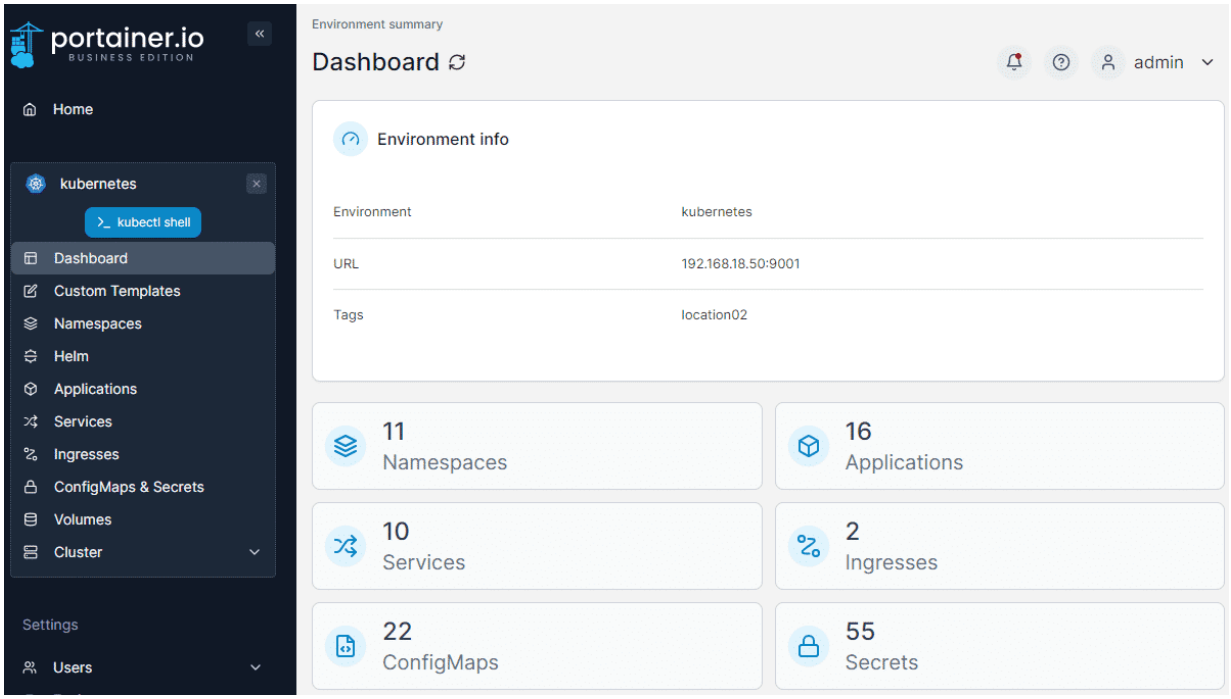
**Kubernetes environment set up and connected to Portainer.**

START

1

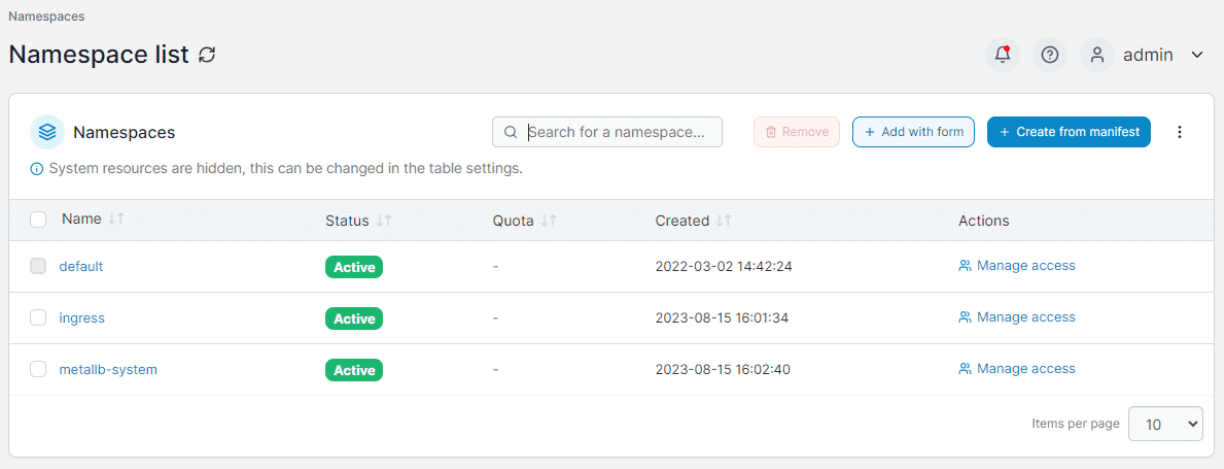
## Creating your application

Log into Portainer and select your Kubernetes environment. You'll see some information about your environment and a summary of objects that exist within it.

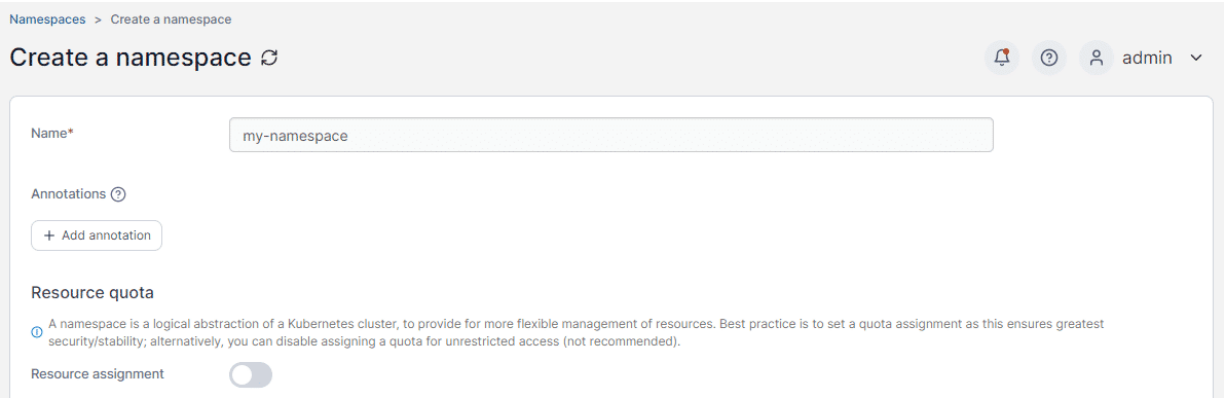


Before we create the application itself, we want to create a namespace for it. Namespaces are used to keep a separation between resources on the same cluster. You can deploy all your applications into one namespace if you want, but we recommend using namespaces to separate workloads especially when working in multi-user environments or where resource quotas are important.

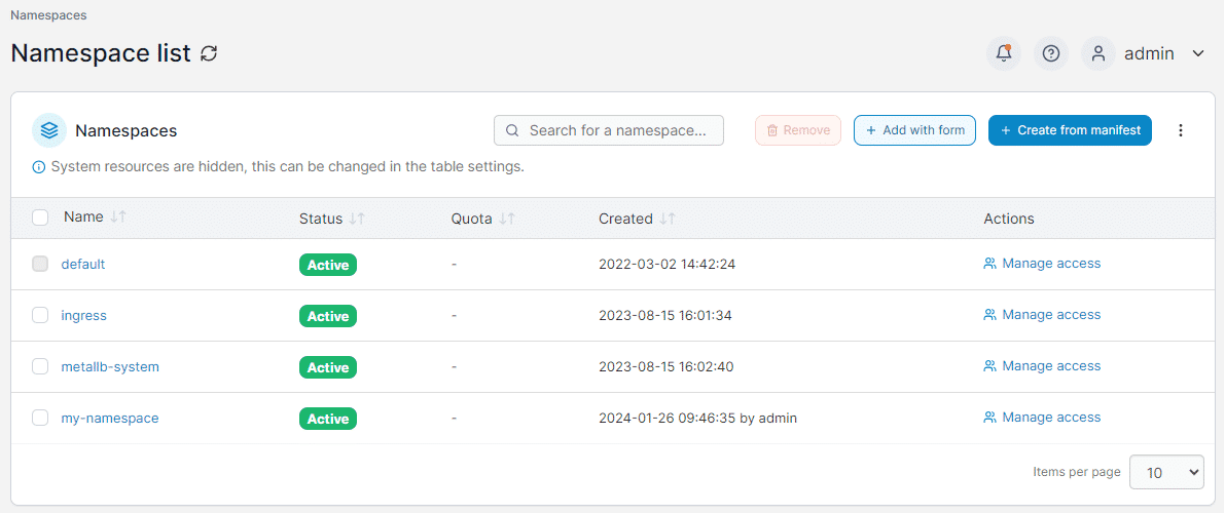
To create a namespace, first click **Namespaces** in the left menu.



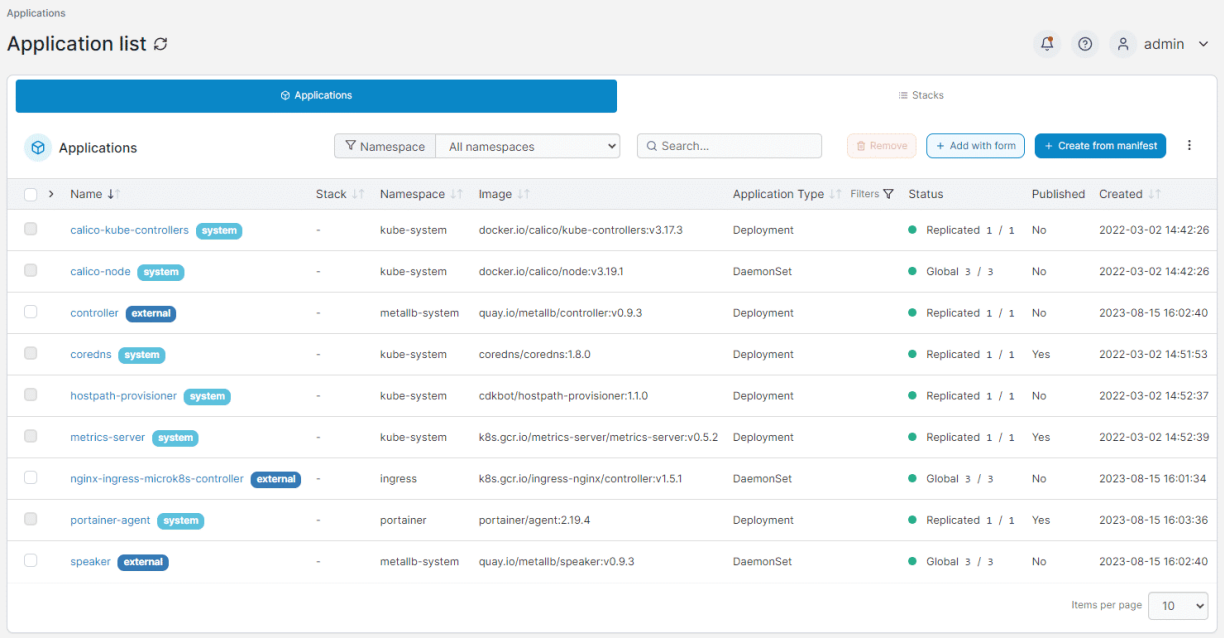

You'll see a list of the namespaces on your environment. To create a new one, click **Add with form** in the top right.



For the purposes of this tutorial we'll leave most of these settings at the default. Simply provide a name for namespace in the **Name** field, then click **Create namespace**. Your namespace will be created and you will be returned to the list of namespaces.

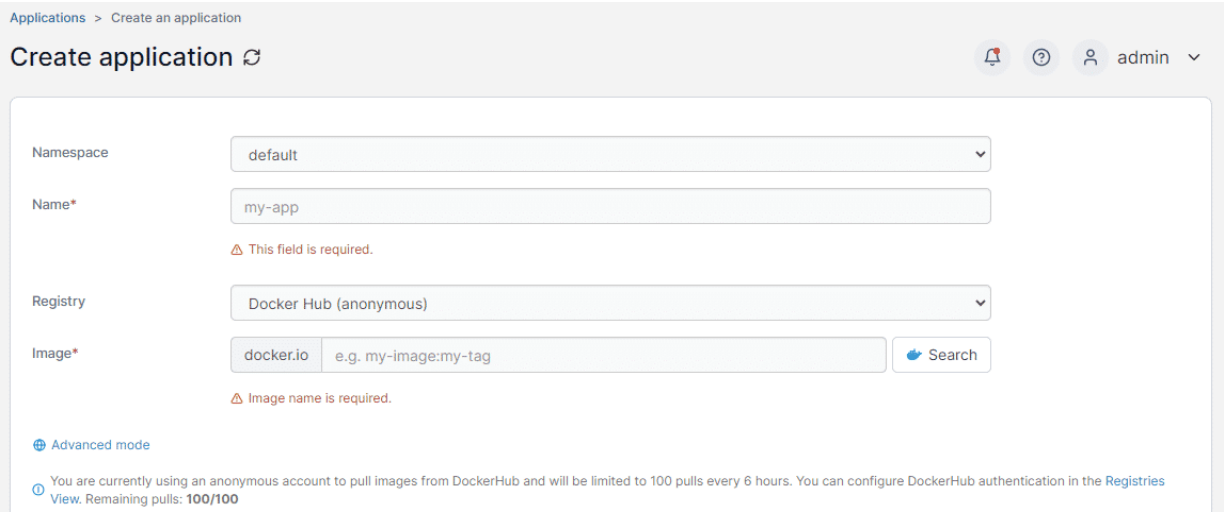



Now let's move on to the actual application creation. Click on **Applications** in the left hand menu.



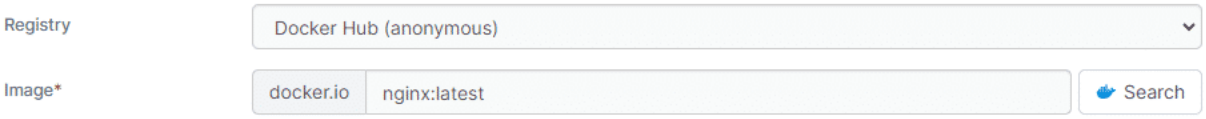

This page lists the applications deployed on your environment, filtered by namespace. You can use the filter dropdown to choose the namespace to show, or choose `All namespaces` to list applications across all of the cluster's namespaces.

To create your application, click on **Add with form** in the top right.

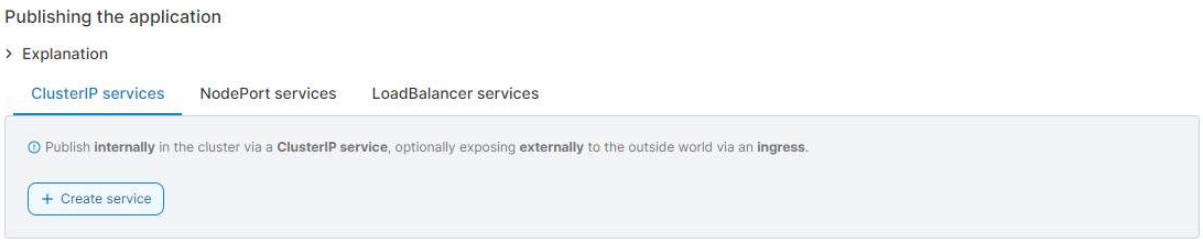


First, let's choose our namespace. From the **Namespace** dropdown, select the namespace we just created. Under that, let's give our application a **name**. For this lesson, let's use `nginx-app`.


Next we choose the registry and image to use to create the application. Ensure `Docker Hub (anonymous)` is selected as the **Registry** and enter `nginx:latest` in the **Image** field.

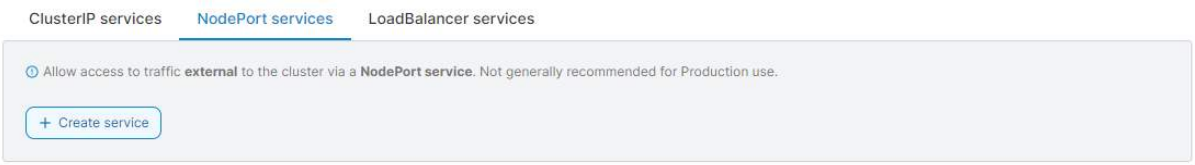



There are a number of other settings we can configure here including adding annotations, environment variables, ConfigMaps and Secrets, persistent storage, resource reservations and deployment methods. For now we'll skip past all of these and on to the **Publishing the application** section.

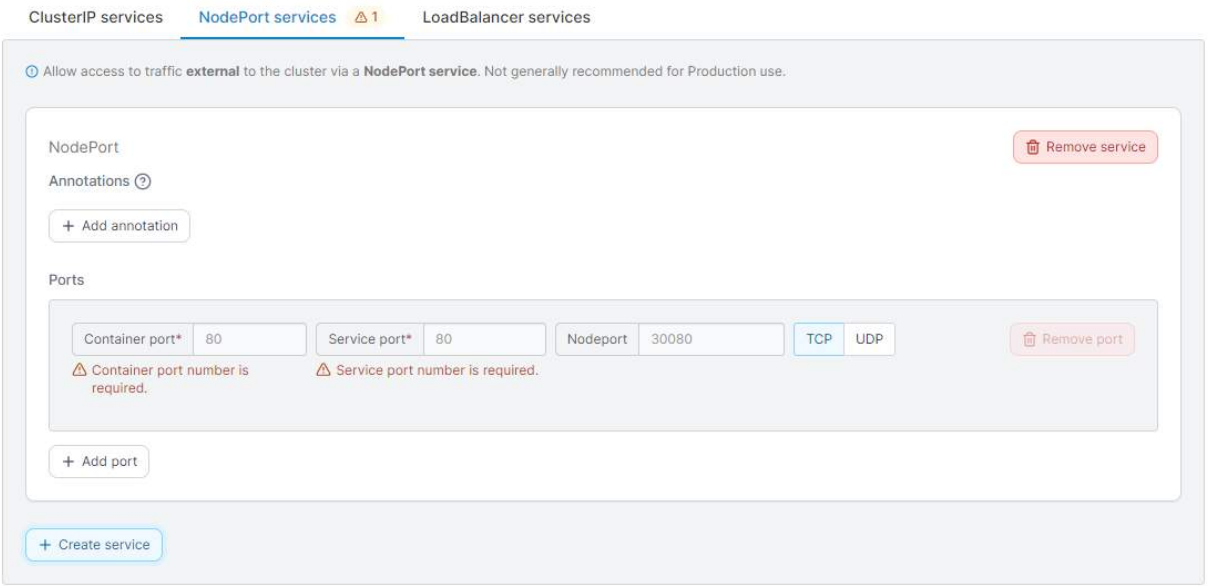


When creating an application in Kubernetes, we are only creating the application itself. Configuring access to that application is done through the use of a Service, which you'll remember from our previous lesson.

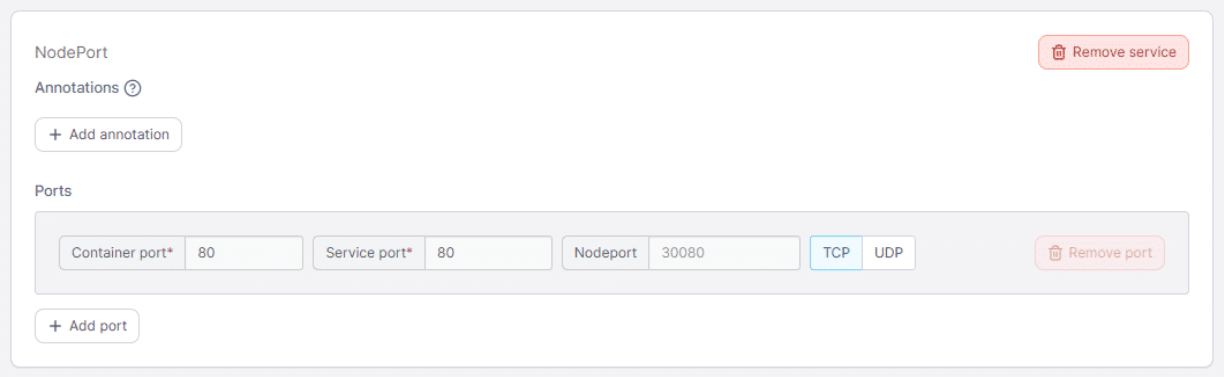
In this section you'll see three types of Services you can use to publish your application: **ClusterIP**, **NodePort** and **LoadBalancer**. For this lesson we'll use the **NodePort** option as it requires the least amount of external configuration, but we'll go into more detail on the options here in a later lesson. For now, select the **NodePort services** tab.



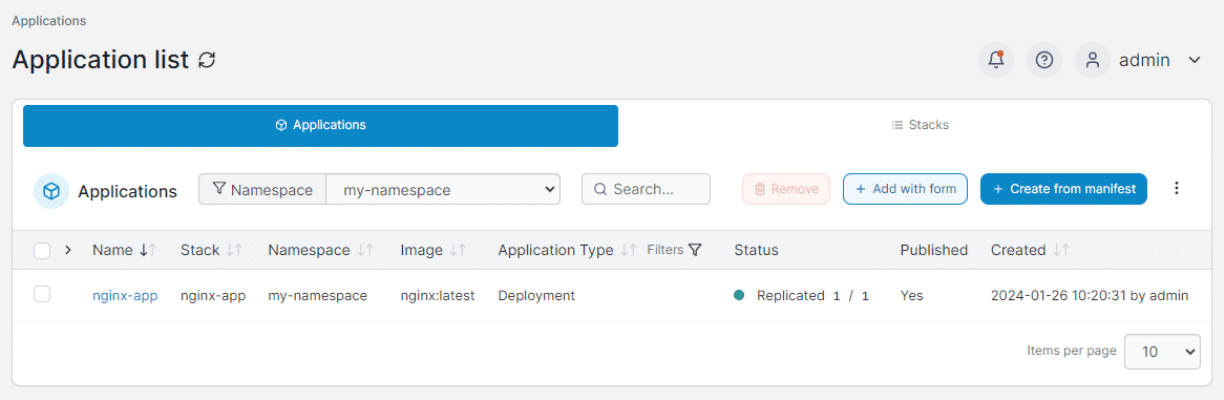

With NodePort, we can provide access to your application from outside the cluster on a specified port. If you're experienced with publishing ports on Docker, this will sound familiar. Let's create a new NodePort service by clicking the **Create service** button.



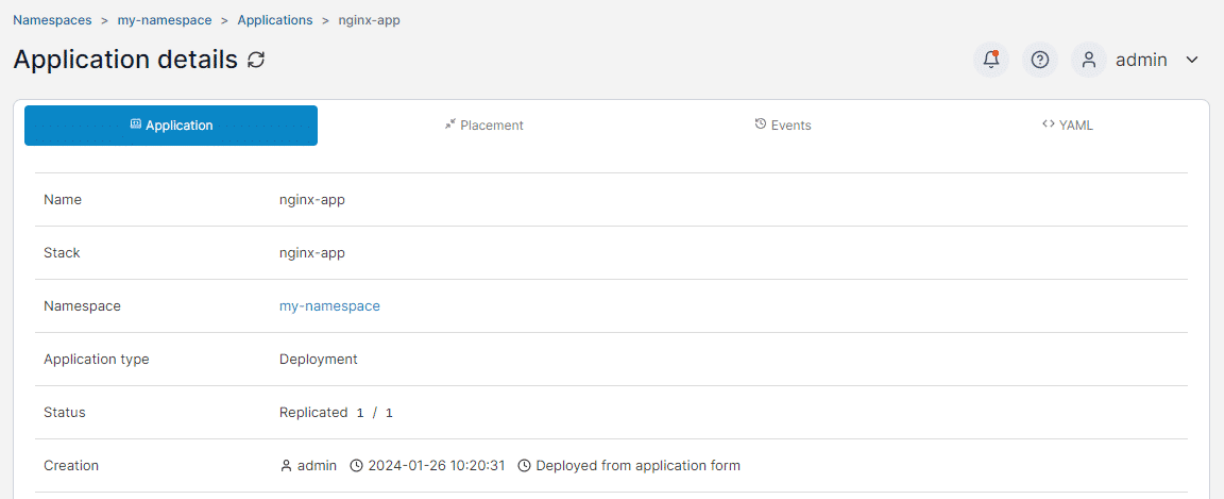
Here you can set annotations for the service, which we don't require. What we will need to set however are the ports. The **Container port** should be set to the port that the application runs on. In the case of nginx, this is port 80. The **Service port** is the port number that the Service will use to provide access to the container port - we'll set this to 80 as well. And lastly, the **Nodeport** is the port that will be exposed on all the nodes of your cluster and will route through the Service to your application. You can set a port here, but it will need to be within the range reserved for NodePort in your Kubernetes cluster (in most cases, from `30000` to `32767`). The best bet is to leave this blank and let Kubernetes assign a free port for you. You can also choose the **Protocol** here - leave this on TCP.




Once all of this is set, you're ready to go. Click **Deploy application** to begin the deployment. The application and NodePort service will be scheduled for creation and you'll be returned to the list of applications. If you don't see yours listed, make sure you have the right namespace selected.




Once the provision completes you should see the status icon turn green with 1 of 1 replicas created. If you now click on the name of the application you'll be taken to the details page for the application, showing you information about the configuration.




Scroll down to the **Accessing the application** section and you'll see the Service we created listed. At the end of the row you should see a Service port(s) definition - this is where we can find the port that Kubernetes assigned to the Service. In my case it was 31517, but yours may differ.

Accessing the application

This application is exposed through service(s) as below:

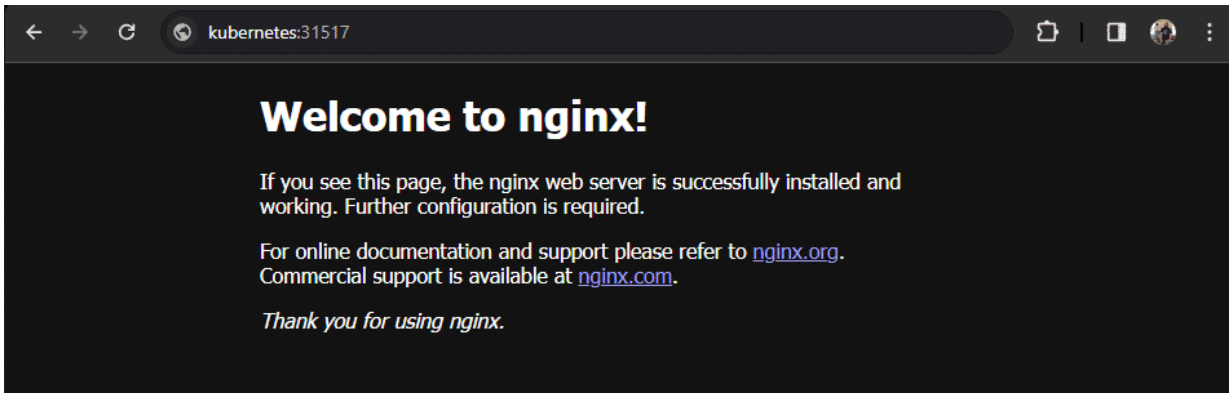
| Service name | Type | Cluster IP | External IP | Container port | Service port(s) |
|---|---|---|---|---|---|
| nginx-app | NodePort | 10.152.183.179 | - | 80 | 80 : 31517/TCP |

With this information we can now check to see whether you can access your service. In a web browser, go to:

```
http://your_node_ip:31517
```

Replace `your_node_ip` with the address of one of the nodes in your Kubernetes cluster, and replace 31517 with the port that was assigned to the NodePort service.

If all has gone well, you should see the default nginx welcome page.




You'll note this is accessible on any of your cluster node's addresses on that specific port - the NodePort service handles the routing regardless of which node you access and

sends the request to the nginx container, even if the container itself is on a different node.

CONTINUE

2

## Summary

In this lesson we've learned how to create a basic application through the form-based method within Portainer. We:

- [ ] Created a namespace for our application.
- [ ] Chose a name and image for our application.
- [ ] Configured a NodePort Service to provide access to the application from outside of the cluster.

In the next lesson we'll look at how we can make changes to the application after deployment.

# Editing your application

In the previous lesson we created our first application - a nginx deployment - using the form-based approach in Portainer. In this lesson we'll look at how we can edit our deployed application. We will:

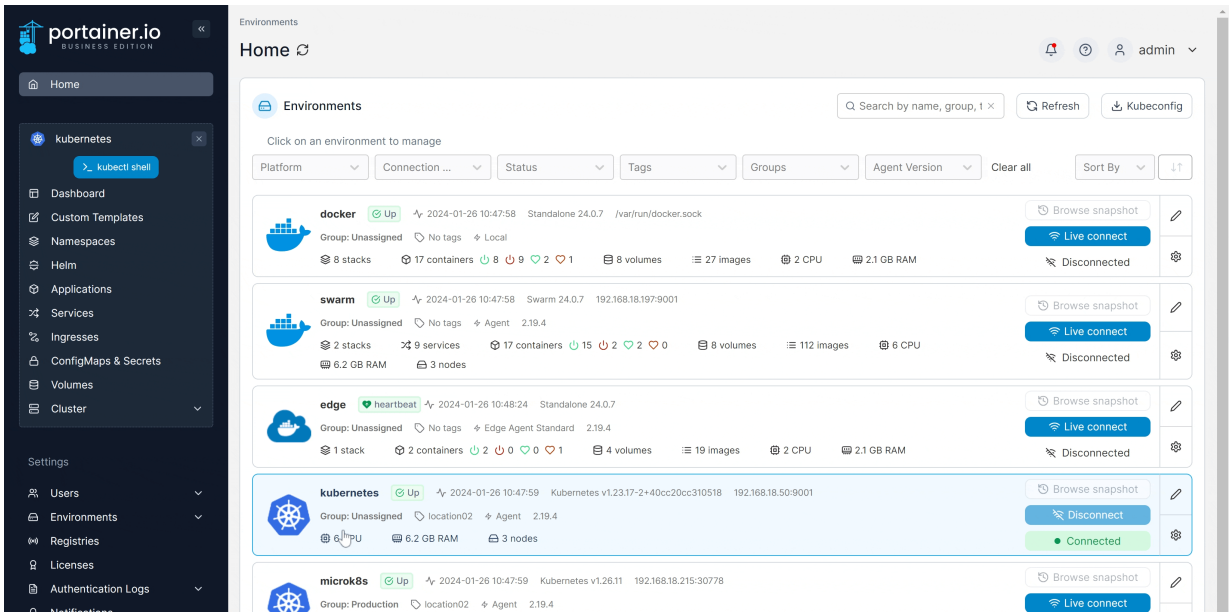
1. View the current configuration of the application
2. Enter edit mode and make some configuration changes
3. Deploy the updated configuration and confirm it has worked as expected

This lesson assumes you have completed the previous lesson and have an application called `nginx-app` on your cluster.

START

# Examining the current configuration

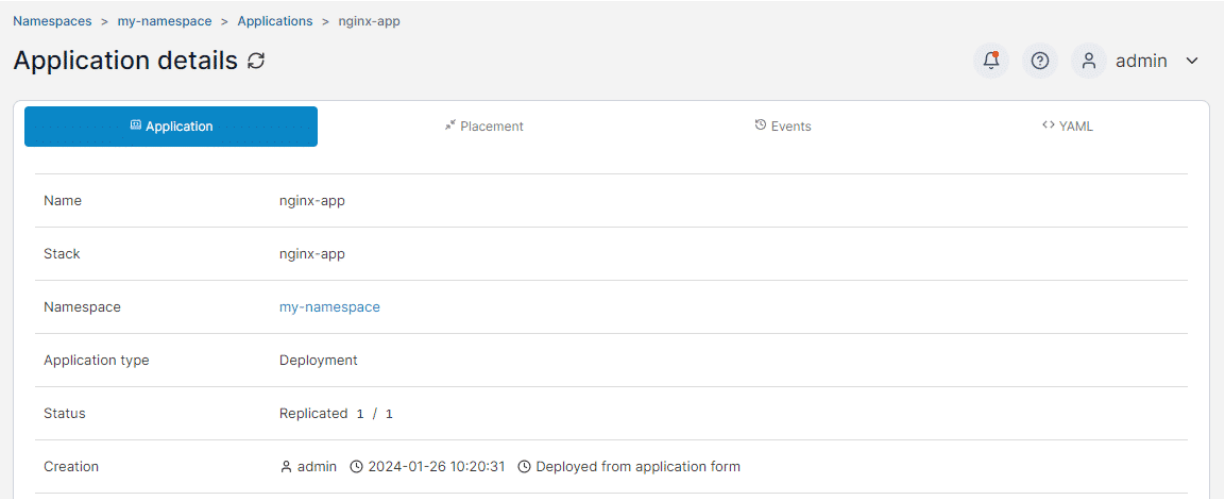
Before we start making changes, let's first have a look at what information is displayed for the current application config. In Portainer, select your Kubernetes environment then select **Applications** in the left hand menu. Select the namespace you created the `nginx-app` application in from the **Namespace** dropdown (if it isn't already visible) then click the `nginx-app` name in the application list.



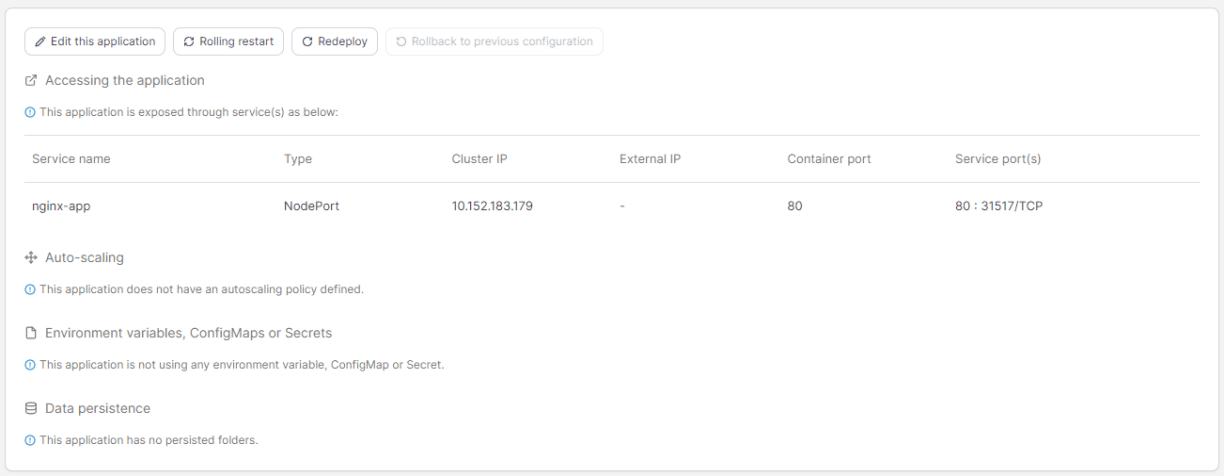

This is the Application details page, which shows you the current configuration for the application you selected. The first section consists of four tabs:

- **Application**, which shows general information about the application including the name, stack, namespace, type and status as well as the creator, creation date and creation method and any notes attached to the application,
- **Placement**, which details any placement constraints and preferences applied to the application,

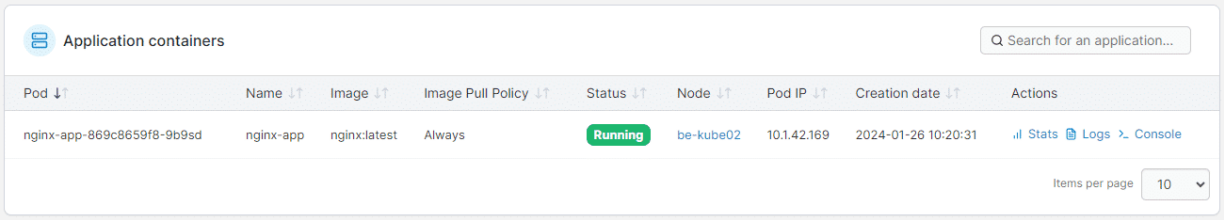
- **Events**, which lists any events related to the application,
- **YAML**, which contains the raw YAML configuration of your application.




Below this initial section you'll find detail on application access (via Services), auto scaling, environment variables, ConfigMaps and Secrets, and data persistence. In our example application, much of this is not set. Here is where you'll also find buttons to edit the application, perform a rolling restart, redeploy, or roll back to a previous config.




And finally, below this section you can see a table detailing the containers that make up the deployment, with detail on each container.




Return to the middle section of the details page and click **Edit this application**, and you'll be taken to edit mode.

CONTINUE

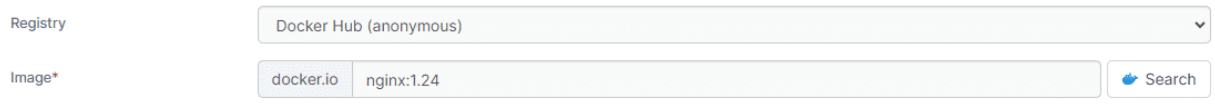
2

## Making some changes

We're now at the Edit application page for your application. You'll note this form looks somewhat similar to the form we used when creating the application, so we won't go into detail on what each field means. Instead, let's jump in and make some configuration changes to our deployment.

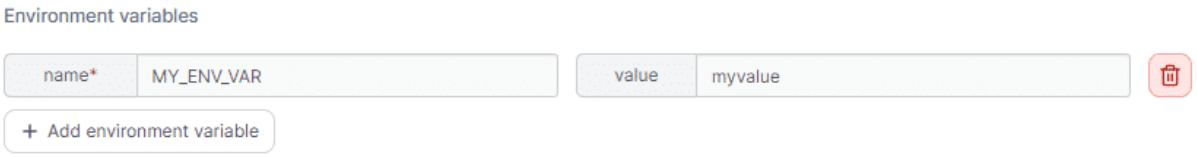
### Changing the image

First off, let's imagine that we want to, instead of the latest image, use a specific version of nginx for our deployment. In the Image field, change `nginx:latest` to `nginx:1.24`. This will mean when we redeploy the application, the Pods will be recreated to use the `nginx:1.24` image instead of `nginx:latest`.



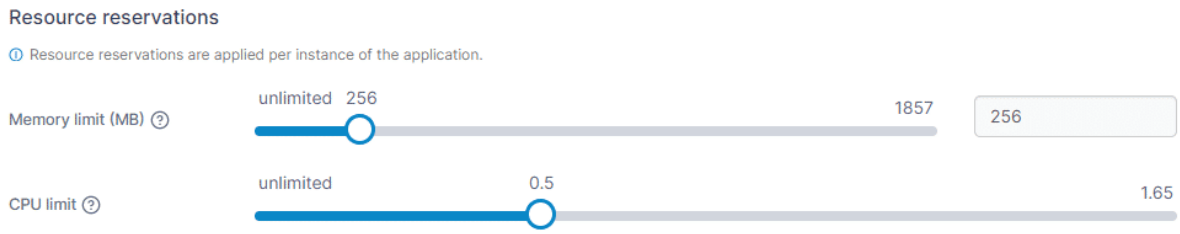

### Add an environment variable

Let's also imagine that our application needs us to set an environment variable. Under **Environment variables**, click the **Add environment variable** button. In the fields that appear, set the name to `MY_ENV_VAR` and the value to `myvalue`. This will create an environment variable called `MY_ENV_VAR`, set to `myvalue`.



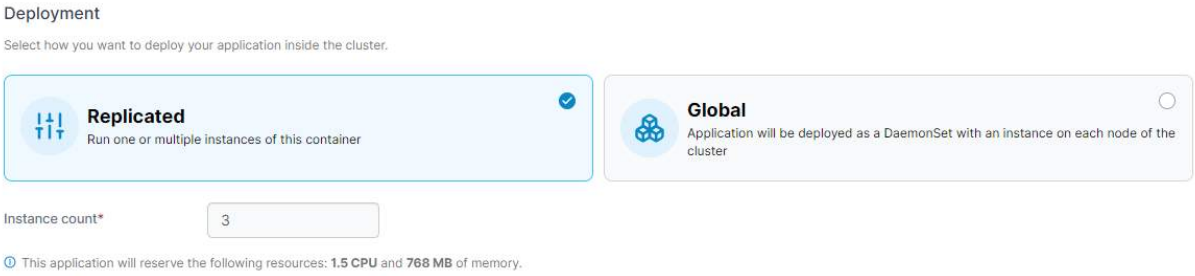

## Configure resource reservations

**Resource reservations** let you specify minimum amounts of memory and CPU to be "reserved" for use by your applications. These resources are unavailable to other workloads on your cluster, and are applied on a per-instance basis. For this example, let's reserve 256MB of memory and half a CPU core for each instance of our app. Either using the sliders or by manually entering values in the boxes, set the **Memory limit** to 256 and **CPU limit** to `0.5`.




## Add more instances

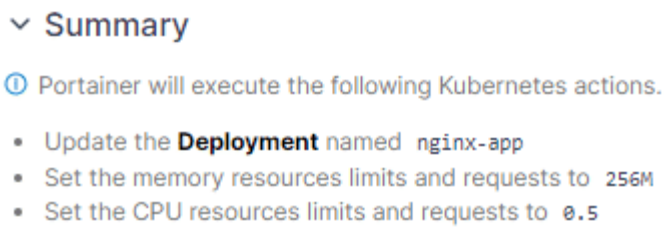
Finally, let's create some replicas of our application. In the **Deployment** section we should already be set to `Replicated`, and below this setting you'll see Instance count is set to 1. This means there will only be a single instance of our nginx application running on our cluster. We want to provide for more concurrent connections, so let's set **Instance count** to 3, which will ensure that 3 instances of the application will be running at all times. You'll also note that when you adjust the instance count, you'll see a message indicating the total resource reservation across all instances.



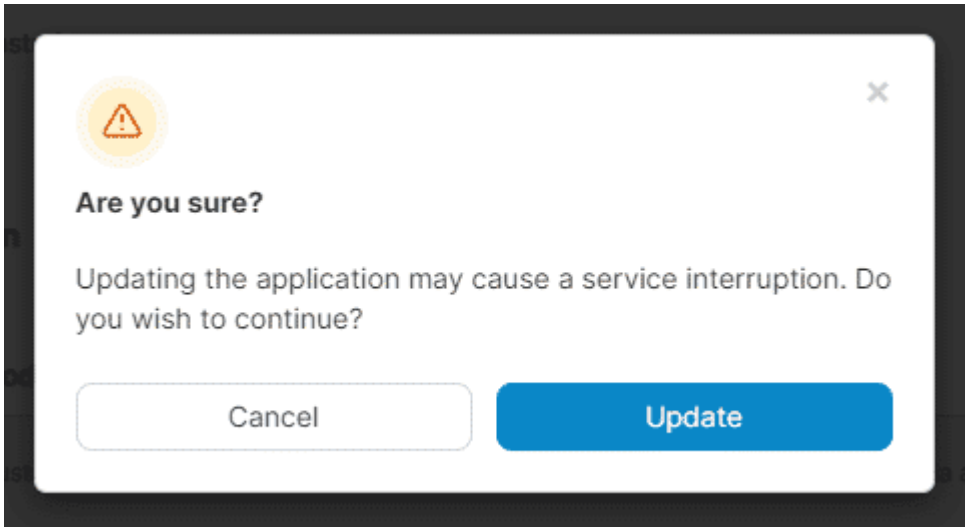

## Checking our work

Scroll down to the bottom of the page and you'll see a **Summary** section. This lists the changes that will be made to your application configuration when it is updated. In our case, you should see that we will:

- Update the Deployment named `nginx-app` (confirming we're working on the right application!)
- Set the memory limit to `256M` (per instance)
- Set the CPU limit to `0.5` (per instance)




Check this all looks good, then click the **Update application** button to begin the update. You will be asked to confirm the update - click **Update** to do so.



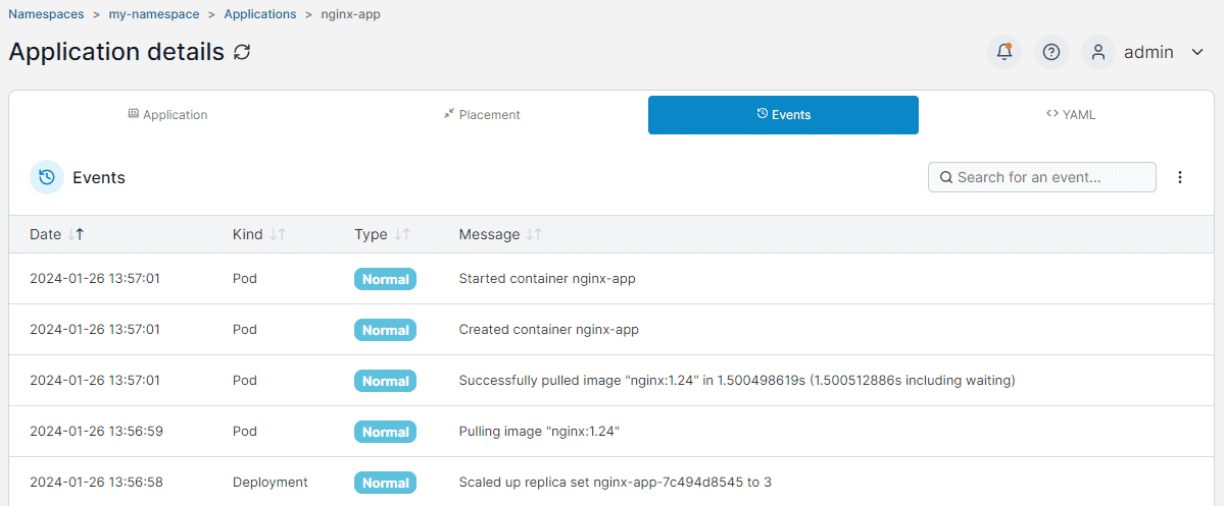

You'll be returned to the Application details page where you can view the progress of your update.

CONTINUE

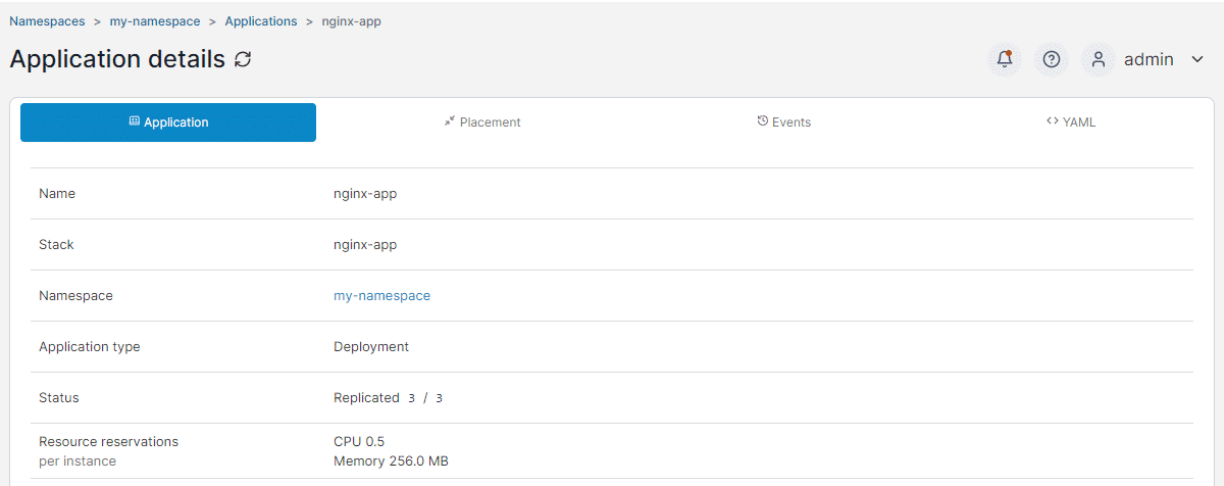
3

## Examining our updated application

When you're first returned to the Application details page after updating, you may see the status of your update still applying. For example, in the Status field you may see the status listed as `Replicated 0 / 3`. You can also scroll to the Application containers section and see three containers now listed, potentially with their status set to `Waiting`. You can click the **refresh icon** next to **Application details** to refresh the page while the update is applied, and you can also check the **Events** tab for updates as to the deployment progress.




Once the update completes, you can check to make sure the changes applied as expected. You should see the **Status** is now `Replicated 3 / 3`, and new **Resource reservations** item is listed with the CPU and Memory reservations we set.

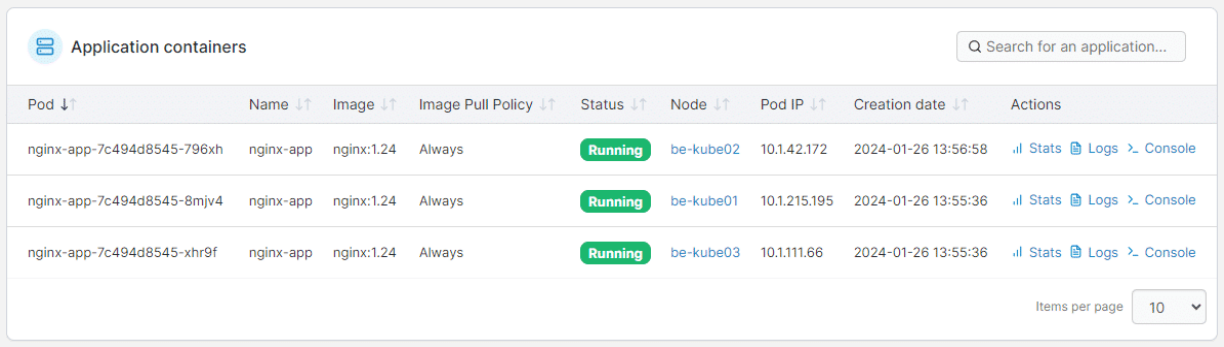



In the second section you'll see our environment variable set:

Environment variables, ConfigMaps or Secrets

| Container | Environment variable | Value | Configuration |
|---|---|---|---|
| nginx-app | MY_ENV_VAR | myvalue | - |

And in the **Application containers** section, you'll see the three containers that now make up our deployment, their status, and the node they're located on. You'll also note here that the **Image** for each container is now `nginx:1.24` as we requested.




And of course you can confirm the application is still working by going to:

```
http://your_node_ip:31517
```

in your web browser (adjust the IP and port as required).

CONTINUE

4

## Summary

First we created our application, and now we've edited it. We have:

- [ ] Checked the current configuration of the application and then proceeded into editing the application
- [ ] Changed the image our application uses, set an environment variable, configured resource reservations and increased the number of instances
- [ ] Deployed our changes and checked they applied successfully.

We only covered a subset of the available settings in this lesson - we'll cover more in later lessons and you'll find additional detail in our documentation. In the next lesson we'll look at how we can deploy more advanced or predefined configurations from a manifest.

# Deploying via manifest

As we covered in our last lesson, Portainer provides a form that you can fill in to deploy your application on your Kubernetes environment. But if you have a more complex deployment to create or have an existing manifest for your application, you can use that instead.

In this lesson we will learn how to:

1. Create a manifest to define an application
2. Deploy the manifest to create the application
3. Edit the application's manifest after deployment

START

## Creating our manifest

When creating an application from a manifest, you will of course need to have a manifest to start with. A manifest is a YAML document that specifies the objects that make up your application. We talked about the different types of objects in the Objects in Kubernetes lesson.

For this lesson, let's replicate what we did in the last lesson (creating an application using the form) but as a manifest - that is, a nginx deployment with 3 replicas along with a NodePort Service for external access. Here's that setup, in YAML:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-manifest
  labels:
    app: nginx-manifest
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx-manifest
  template:
    metadata:
      labels:
        app: nginx-manifest
    spec:
      containers:
      - name: nginx-manifest
        image: nginx:latest
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: nginx-manifest-service
  labels:
```

```
    app: nginx-manifest
spec:
  type: NodePort
  selector:
    app: nginx-manifest
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

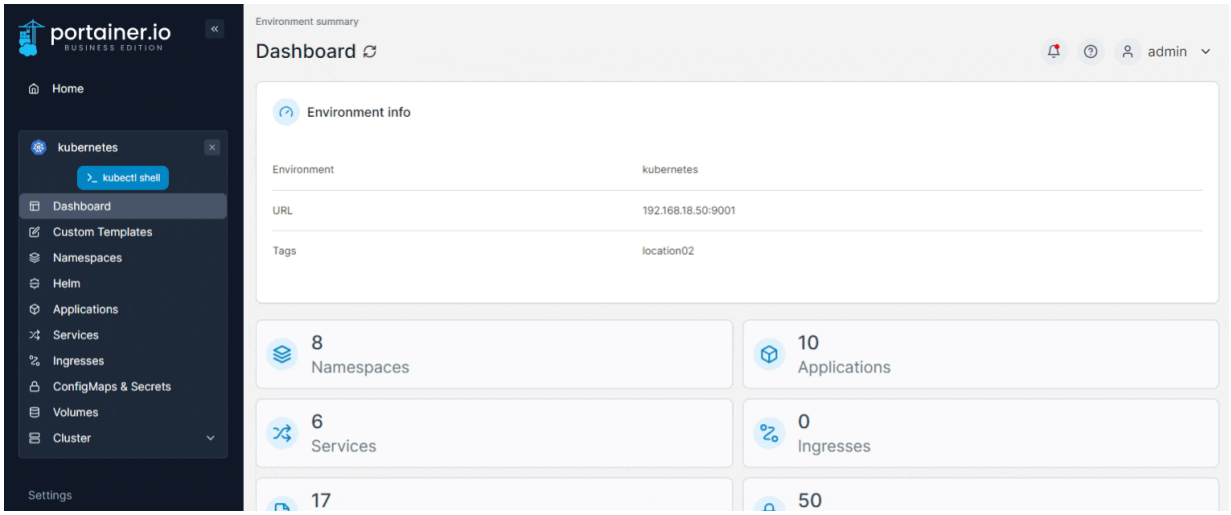
Most of this should be straightforward based on what we've learned so far - if you need a refresher on how object definitions work, have a look at the Objects in Kubernetes lesson from earlier. One change to note is that we're using the name `nginx-manifest` instead of `nginx-app` - this is so that we can deploy it in the same namespace alongside the `nginx-app` we created in the previous lesson without running into any conflicts.

Now that our manifest is ready, let's deploy it!

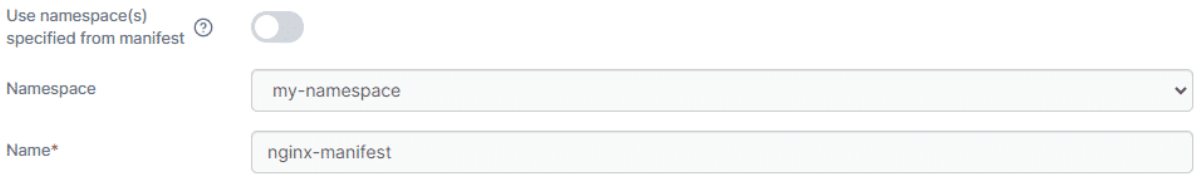
CONTINUE

# Deploy our manifest

To create an application from a manifest, from Portainer select **Applications** in the left hand menu then click the **Create from manifest** button in the top right.




You'll be taken to the **Advanced deployment** page where you can begin the provision. First off, we'll need to select a namespace and provide a name for the application. From the **Namespace** dropdown, select the namespace you want to use (it can be the same as the one we used previously for the `nginx-app` deployment) and enter a **Name** - we'll use `nginx-manifest`.




Next we choose a build method. This is how we specify the source of our manifest file. There are four options to choose from:

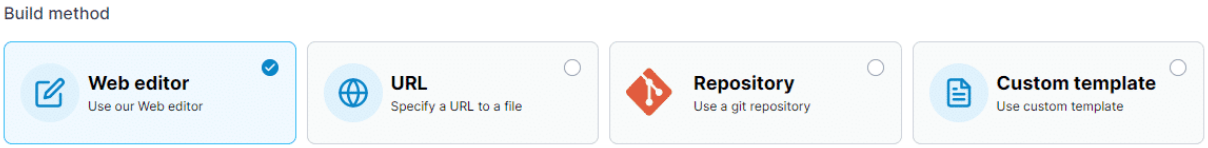
- **Web editor**
  This option lets you type or paste a manifest file directly into a text field, and is the option we'll use for this example.

- **URL**
  This option lets you specify a URL where your manifest resides. Portainer will retrieve the manifest from that URL and deploy it.

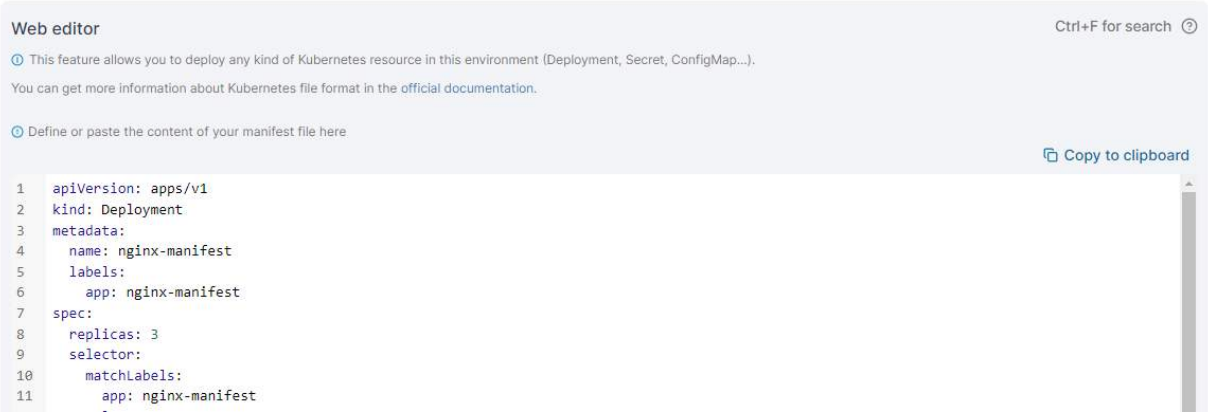
- **Repository**
  This option lets you specify a Git repository that contains your manifest. Deploying from a repository also provides some additional automatic update functionality over the other options.

- **Custom template**
  This option lets you select a predefined custom template to base your deployment from. Custom templates can be created by you for future use like this.

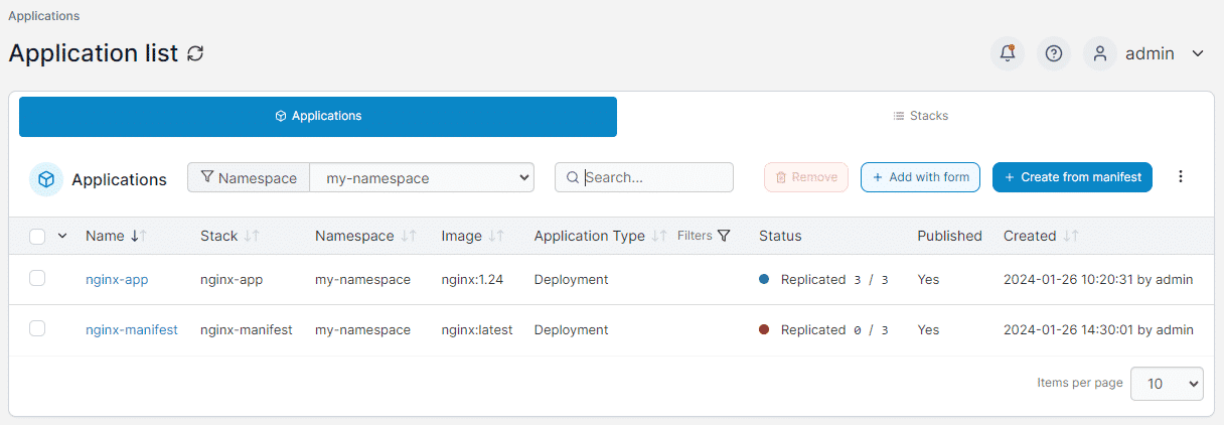
We want to use the **Web editor** option for this example, so select that.



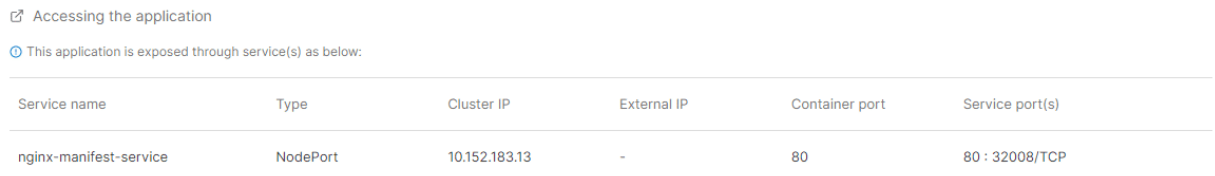

Now we're presented with a web editor field we can paste our manifest into. Go ahead and do so now.




When you're ready, click the **Deploy** button to begin the deployment. You'll now be returned to the application list while the deployment completes. You can check the status in the **Status** column or click the application name for more details.




Since we didn't explicitly define a port to use for our NodePort service, one would have been assigned automatically. You can find this in the application details page under **Accessing the application**. In my case, this was port `32008`.




You can check your deployment worked by opening the URL in your browser:

```
http://your_node_ip:32008/
```

Replace `your_node_ip` with an IP address of a node in your Kubernetes cluster, and `32008` with the port that was assigned to the NodePort service.

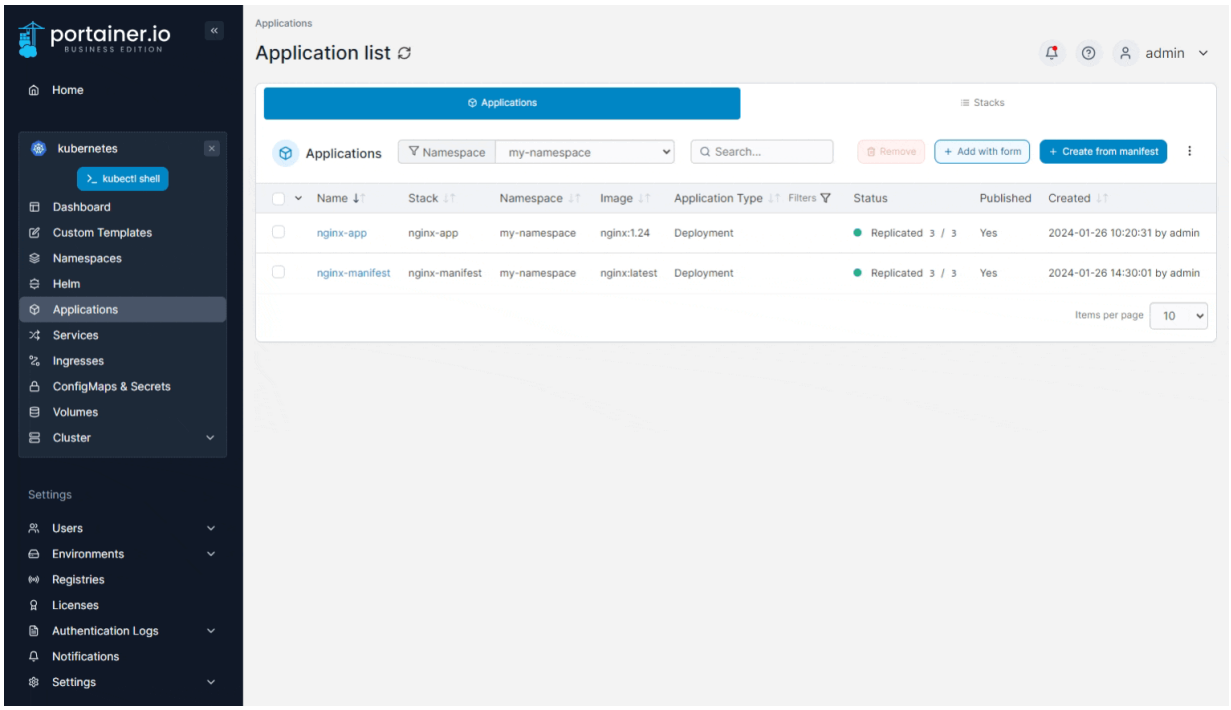
CONTINUE

3

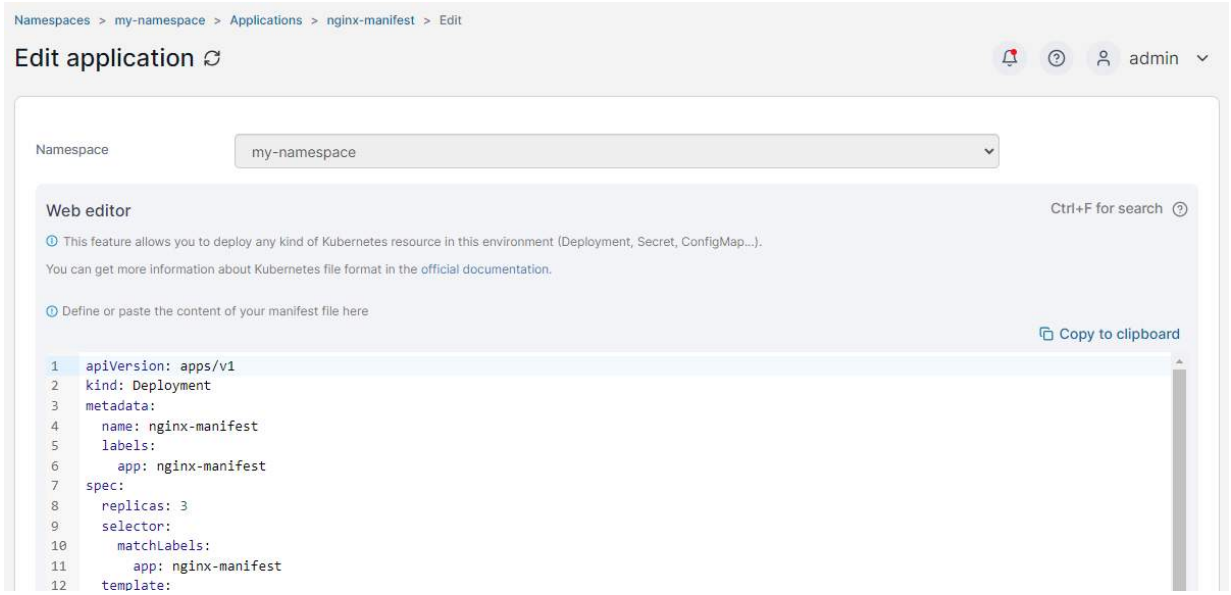
## Making some changes

So we've deployed our manifest, but let's imagine we want to make some changes. Like we did with the form-based provision in the last lesson, let's set an environment variable and some resource limits.

From the Applications list, click on the name of your application (in our case, `nginx-manifest`). You'll be taken to the Application details page. Scroll down and click the **Edit this application** button.




Since this application was created from a manifest using the Web editor, we're taken to a page where we can make changes to that manifest.




Let's add an environment variable definition to our code:

```
env:
- name: MY_ENV_VAR
  value: myvalue
```

and our resource limits:

```
resources:
  limits:
    cpu: 0.5
    memory: 256M
  requests:
```

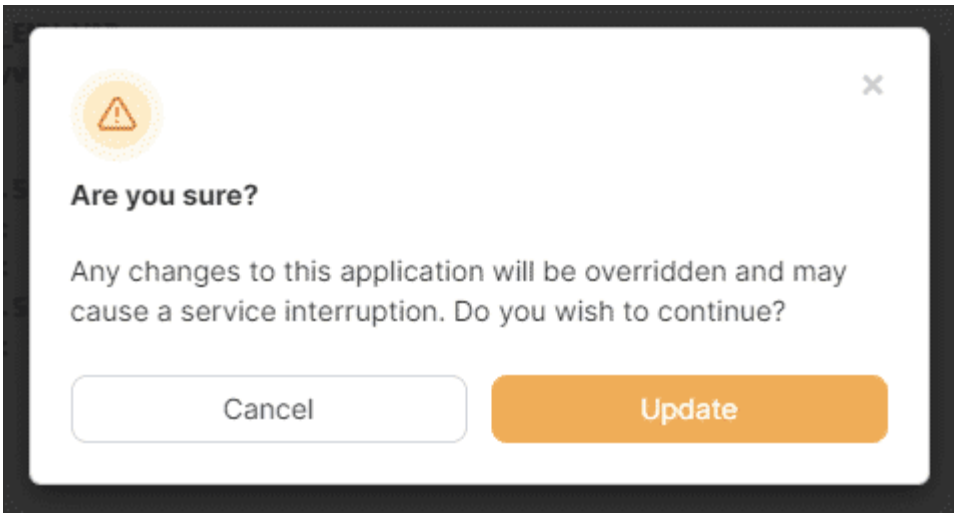
```
      cpu: 0.5
      memory: 256M
```

This matches the configuration we used for the form-based setup. Both of these configuration blocks go in the `spec` section for the `nginx` container under the `nginx-manifest` deployment. The full adjusted code should look like this:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-manifest
  labels:
    app: nginx-manifest
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx-manifest
  template:
    metadata:
      labels:
        app: nginx-manifest
    spec:
      containers:
      - name: nginx-manifest
        image: nginx:1.24
        ports:
        - containerPort: 80
        env:
        - name: MY_ENV_VAR
          value: myvalue
        resources:
          limits:
            cpu: 0.5
            memory: 256M
          requests:
            cpu: 0.5
            memory: 256M
```

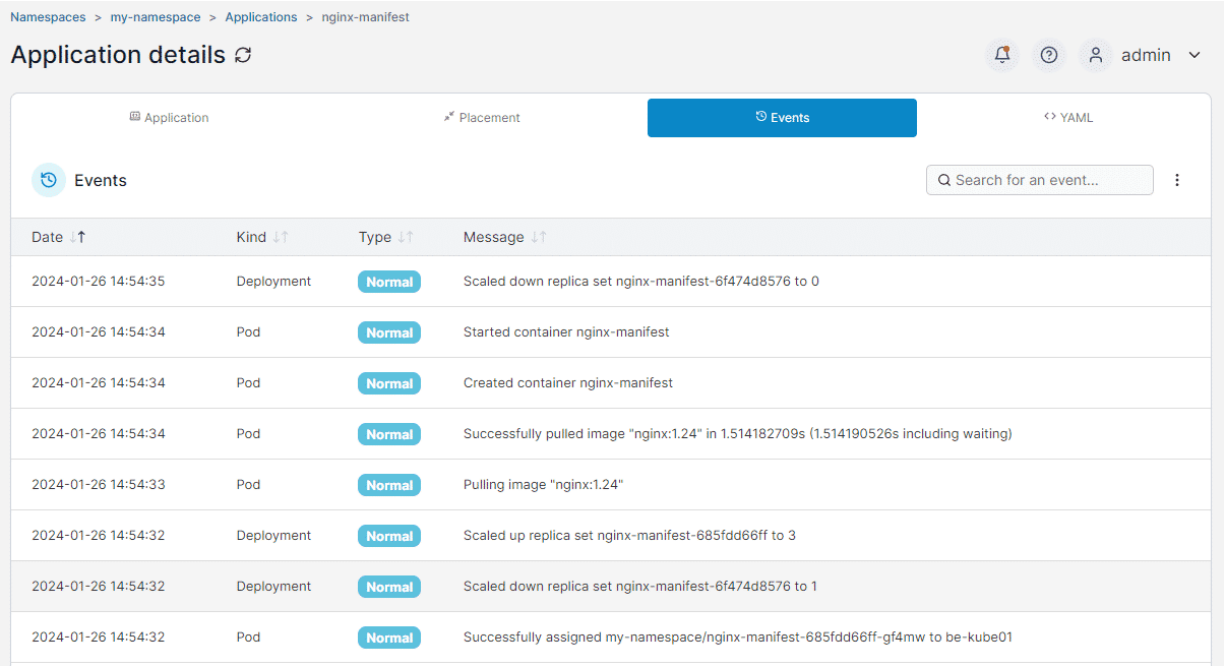
```
---
apiVersion: v1
kind: Service
metadata:
  name: nginx-manifest-service
  labels:
    app: nginx-manifest
spec:
  type: NodePort
  selector:
    app: nginx-manifest
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

Update the web editor field with the above configuration, and when you're ready click the **Update the application** button. You'll be asked to confirm the update - do so.




Your updated configuration will now be pushed to your cluster, and the application will be reconfigured with your changes. As always, you can check the status of this from the Applications list and the details page for the application itself. You can also check the

**Events** tab in the application details to see the actions that were taken as part of the reprovisioning.




CONTINUE

4

## Summary

In this lesson we've looked at how we can deploy an application from a manifest file directly, rather than using the form-based approach. We:

- Created a manifest file for our nginx deployment

- [ ] Used Portainer's Web editor to supply our manifest file and deploy it on our cluster
- [ ] Edited the manifest file within Portainer to adjust the application config and redeployed it with the new options.

Congratulations! You now know the basics of deploying on Kubernetes with Portainer. In the future we'll be adding additional lessons that will dive deeper into some of the other configuration object types - namely storage and networks.

# Summary

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

In this course we've looked at the basics of deploying on Kubernetes, including:

- [ ] An introduction to Kubernetes, including what it is (and isn't)
- [ ] The anatomy of a Kubernetes cluster
- [ ] The objects that make up a Kubernetes deployment
- [ ] Creating and editing an application through the form-based approach in Portainer
- [ ] Creating and editing an application using a manifest in Portainer

This is intended as an introduction to get you started with using and deploying on Kubernetes with Portainer, and doesn't cover every possible object, configuration and nuance of the platform. From here you should be able to dive into using Kubernetes for your deployments with knowledge of the concepts and processes that make up your clusters.