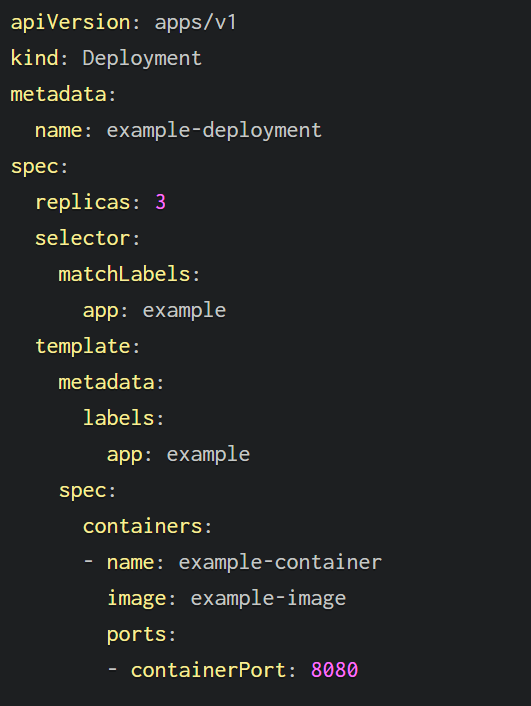
A typical Kubernetes Deployment YAML file includes the following key components:

* **apiVersion**: Specifies the Kubernetes API version, such as “apps/v1” for Deployments.
* **kind**: Specifies the type of Kubernetes resource, in this case, “Deployment.”
* **metadata**: Provides metadata for the Deployment, including the name, labels, and annotations.
* **spec**: Defines the desired state of the Deployment, including the number of replicas, the pod template, and any other related specifications. It includes:
* **replicas**: Specifies the desired number of identical pod replicas to run.
* **selector**: Specifies the labels that the Replica Set uses to select the pods it should manage.
* **template**: Contains the pod template used for creating new pods, including container specifications, image names, and container ports.



Let’s break each line down…

* apiVersion: Specifies the Kubernetes API version. In this case, it’s using the “apps/v1” API version, which is appropriate for Deployments.
* kind: Specifies the type of Kubernetes resource. Here, it’s “Deployment,” indicating that this configuration file is defining a Deployment.
* spec: This section defines the desired state of the Deployment.
* replicas: 3: Specifies that you want to run three replicas of your application.
* selector: Describes the selector to match pods managed by this Deployment.
* matchLabels: Specifies the labels that the Replica Set created by the Deployment should use to select the pods it manages. In this case, pods with the label app: example are selected.
* template: Defines the pod template used for creating new pods.
* metadata: Contains the labels to apply to the pods created from this template. In this case, the pods will have the label app: example.
* spec: Describes the specification of the pods.
* containers: This section specifies the containers to run in the pod.
* name: example-container: Assigns a name to the container.
* image: example-image: Specifies the Docker image to use for this container.
* ports: Defines the ports to open in the container.
* containerPort: 8080: Indicates that the container will listen on port 80.

**Kubernetes deployment YAML - Examples**

Let’s look at some Kubernetes Deployment YAML examples.

**Example 1 — Run NGINX Containers using a YAML File**

This will create a deployment running three NGINX pods.

*nginx-deployment.yaml*

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment

spec:

replicas: 3

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx-container

image: nginx:latest

ports:

- containerPort: 80

**Example 2 — Pass environment variables in Kubernetes deployment YAML (*and why not to do that!)***

You can pass environment variables to containers in a Kubernetes Deployment YAML to configure application settings, provide secrets, or customize the behavior of your application.

[Environment variables in Kubernetes](https://spacelift.io/blog/kubernetes-environment-variables) are often used to decouple configuration from the container image, making it more flexible and easier to manage. To define environment variables in a Deployment YAML file, you can use the env field under the containers section.

*env-deployment.yaml*

apiVersion: apps/v1

kind: Deployment

metadata:

name: my-app-deployment

spec:

replicas: 3

template:

spec:

containers:

- name: my-app-container

image: my-app-image

ports:

- containerPort: 80

env:

- name: DATABASE\_HOST

value: db.example.com

- name: API\_KEY

valueFrom:

secretKeyRef:

name: my-secret

key: api-key

### Example 4 — Kubernetes deployment YAML with resource limits

Setting resource limits helps K8s manage the resources allocated to each container, ensuring that they do not exceed the specified limits and can be scheduled on appropriate nodes.

Building on our previous Example 1 for an NGINX deployment, we can add the resources section:

nginx-deployment-with-resource-limits.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment

spec:

replicas: 3

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:latest

ports:

- containerPort: 80

resources:

limits:

memory: "256Mi" # Maximum memory allowed

cpu: "200m" # Maximum CPU allowed (200 milliCPU)

requests:

memory: "128Mi" # Initial memory request

cpu: "100m" # Initial CPU request

* resources: This section is used to define resource requests and limits for the container.
* limits: Specifies the maximum amount of CPU and memory that the container is allowed to use. In this example, the container is limited to a maximum of 256 MiB of memory and 200 milliCPU (0.2 CPU cores).
* requests: Specifies the initial amount of CPU and memory that the container requests when it starts. In this example, the container requests 128 MiB of memory and 100 milliCPU (0.1 CPU cores) initially.

You can find more on limits and requests on the [official documentation pages](https://kubernetes.io/docs/concepts/policy/limit-range/).

### Example 5 — Kubernetes deployment YAML with health checks

Again building on Example 1, we can add some health checks to our deployment YAML to include [livenessProbe](https://spacelift.io/blog/kubernetes-liveness-probe" \t "_blank) and [readinessProbe](https://spacelift.io/blog/kubernetes-readiness-probe" \t "_blank) settings.

nginx-deployment-with-health-checks.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment

spec:

replicas: 3

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:latest

ports:

- containerPort: 80

livenessProbe:

httpGet:

path: / # The path to check for the liveness probe

port: 80 # The port to check on

initialDelaySeconds: 15 # Wait this many seconds before starting the probe

periodSeconds: 10 # Check the probe every 10 seconds

readinessProbe:

httpGet:

path: / # The path to check for the readiness probe

port: 80 # The port to check on

initialDelaySeconds: 5 # Wait this many seconds before starting the probe

periodSeconds: 5 # Check the probe every 5 seconds

* livenessProbe: The liveness probe checks whether the container is still alive. It uses an HTTP GET request to the / path on port 80 of the container. If the probe fails, K8s will restart the container.
* readinessProbe: The readiness probe checks whether the container is ready to serve traffic. It also uses an HTTP GET request to the / path on port 80 of the container. If the probe fails, the container is marked as not ready, and K8s won’t send traffic to it.

Many other configurations can be applied using health checks.

### Example 6 — Kubernetes deployment YAML with persistent volumes

Building on Example 1, we can modify the deployment YAML to add a volumes section, in which we define a [Kubernetes Persistent Volume Claim (PVC)](https://spacelift.io/blog/kubernetes-persistent-volumes). The PVC defines the storage requirements, and a Persistent Volume (PV) is dynamically provisioned or statically assigned to meet those requirements.

nginx-deployment-with-pvc.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment

spec:

replicas: 3

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:latest

ports:

- containerPort: 80

volumes:

- name: data

persistentVolumeClaim:

claimName: my-pvc # Name of the Persistent Volume Claim

 PVC is defined using persistentVolumeClaim. In this case, it’s referenced by the name my-pvc.

The my-pvc PVC must be defined separately in another YAML file. The Deployment’s container can then mount the volume specified by the name: data in the volumes section.