



# Kubernetes in Action

—

# What is Kubernetes?

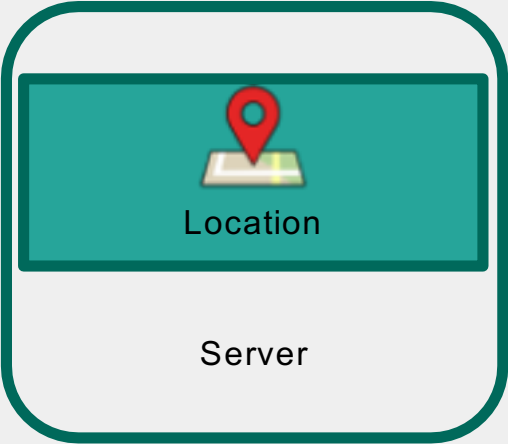


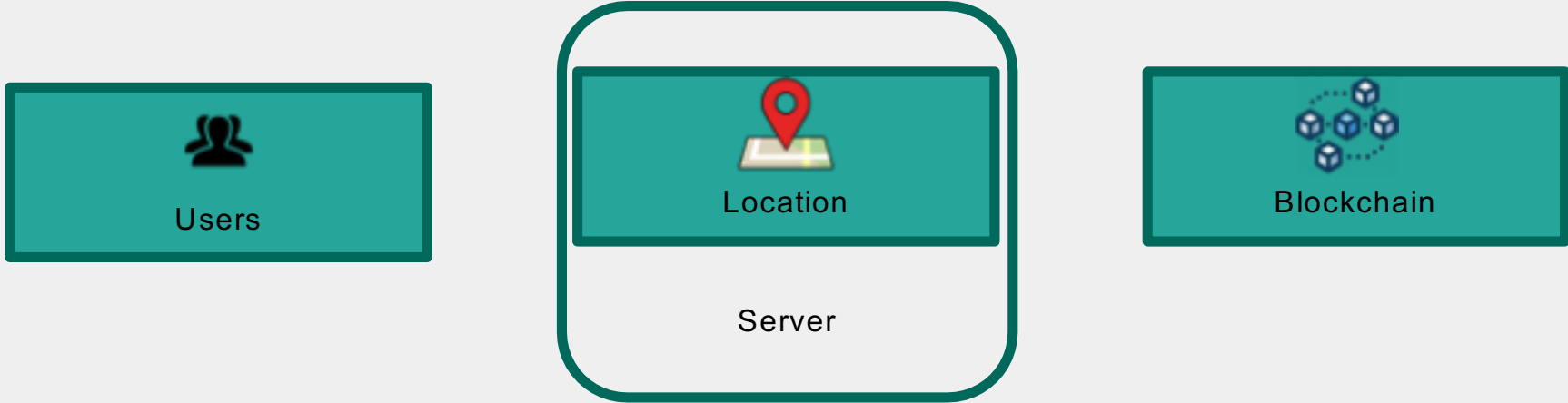
Platform for running  
applications

# WHAT PROBLEMS EXACTLY KUBERNETES SOLVE?



Location







Users

Server



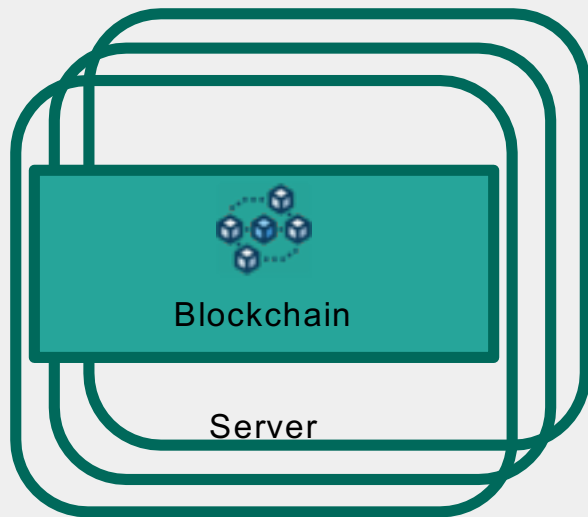
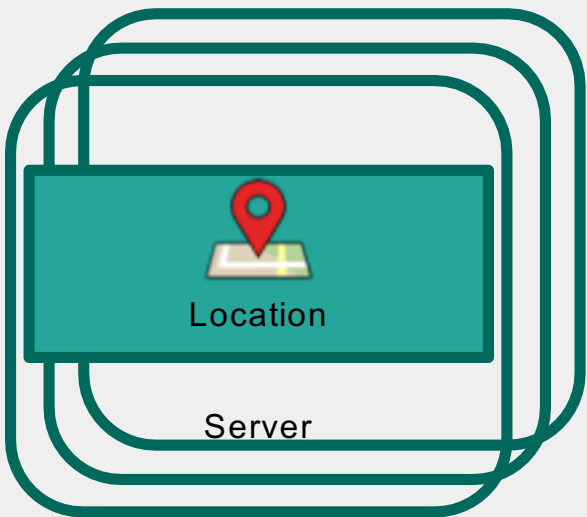
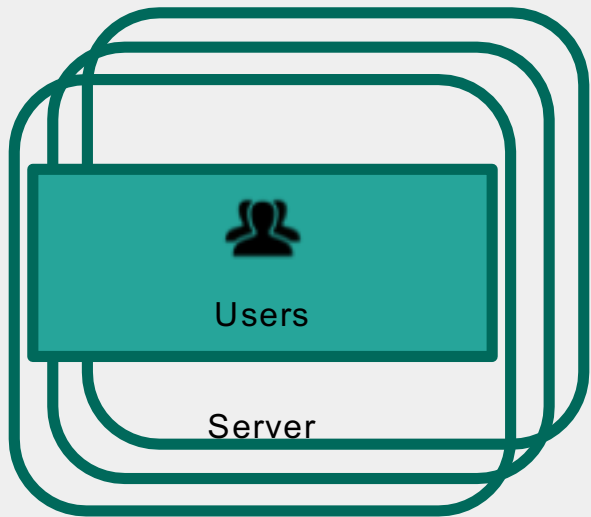
Location

Server



Blockchain

Server







# Possible problems as we scale?

- Dependency Management
- “It Works on My machine”
- Tedious Networking
- Monitoring server crashes


FOR EVERY COMPLEX PROBLEM THERE IS  
A SIMPLE SOLUTION



Users



Location



Blockchain

Server

Server

Server

Server

Server



Users



Location



Blockchain



# kubernetes

Server

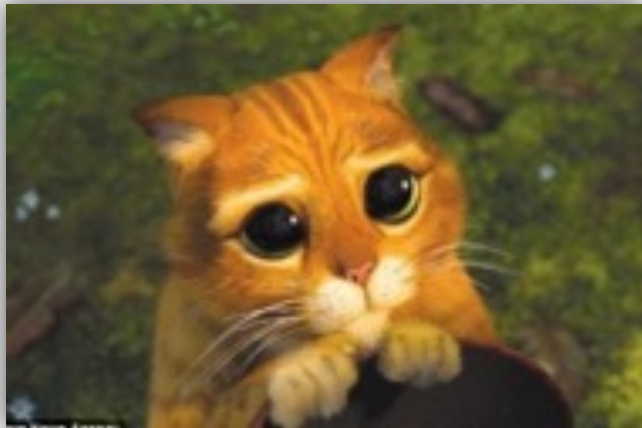
Server

Server

Server

Server

# Cat-Cow Analogy



app1-webserver-a.bank.com

**SCALE UP**

If it is sick, you do everything you can to **heal it**.

**vs.**



server385.bank.com

**SCALE OUT**

If it's sick, you **shoot it** and **replace it**.

# Monolithic vs Microservices Architectures

	Monolithic	Microservices
<b>Code Base</b>	Harder to maintain and test as it grows larger	Code complexity is greatly reduced
<b>Bugs</b>	Coupling between modules causes random bugs when changes are made.	Service separation promotes decoupled design that have less bugs.
<b>Learning Curve</b>	Steep learning curve for new team members.	There is a lot less to learn to become productive
<b>Availability</b>	Deployments and upgrades require downtimes.	Deployment don't require downtime.
<b>Fault Tolerance</b>	If the services crashes, your entire site goes down	If microservice crashes, the rest of the system keep going.
<b>Scaling</b>	Inefficient scaling.	Each microservices can be scaled individually according to its needs.
<b>Tech</b>	Difficult to incorporate to new technologies	Service can use different tech stack.

# Why we need pods?

- Containers are designed to run on single process per container
- If you run multiple processes on a single container it is your responsibility to handle to keep all of them running and managing it
- Example, if you are using pod, you can include a mechanism to automatically restart a pod if it crashes

# Understanding pods

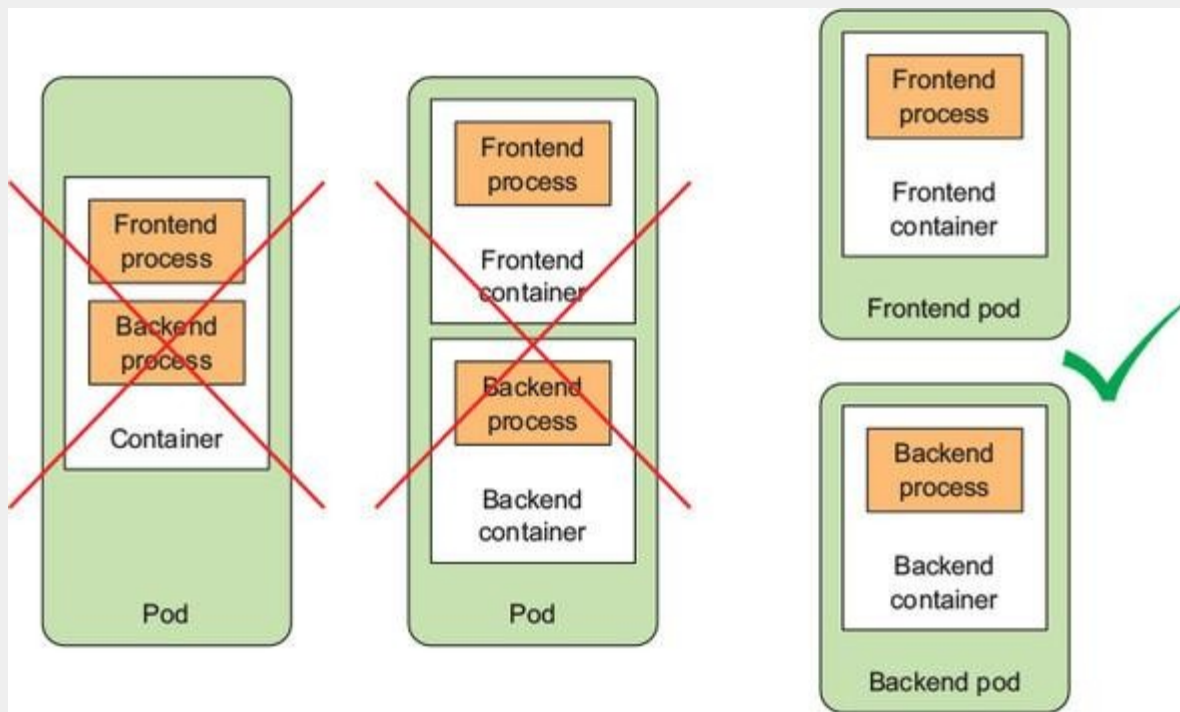
- Pods allows you to manage containers as a single unit
- All containers of pods runs on same Network and UTS Namespace (Linux namespaces), they all share same hostname and network interfaces
- Containers on different port can not run into port conflicts



# Organising pods properly

- Instead of stuffing all your application logic into single pod, you organise them into multiple pods
- In usual scenario you have one container per pod
- Do you think both frontend and backend processes shall be on same pod as different container? Or each container as a different pod?

# Organising pods properly



# When to use multiple container on a single pod?

- Do they need to run together or can they run on different hosts ?
- They represent as a whole entity or a individual component?
- Must they scale together or individually?

# Creating pods from yaml file

Pods and other Kubernetes resources are usually created by posting a JSON or YAML manifest to the Kubernetes REST API endpoint.

## Main part of pod definition

- **Metadata** includes the name, namespace, labels, and other information about the pod.
- **Spec** contains the actual description of the pod's contents, such as the pod's containers, volumes, and other data.
- **Status** contains the current information about the running pod, such as what condition the pod is in, the description and status of each container, and the pod's internal IP and other basic info

# Creating pods from yaml file

Listing 3.2. A basic pod manifest: kubia-manual.yaml

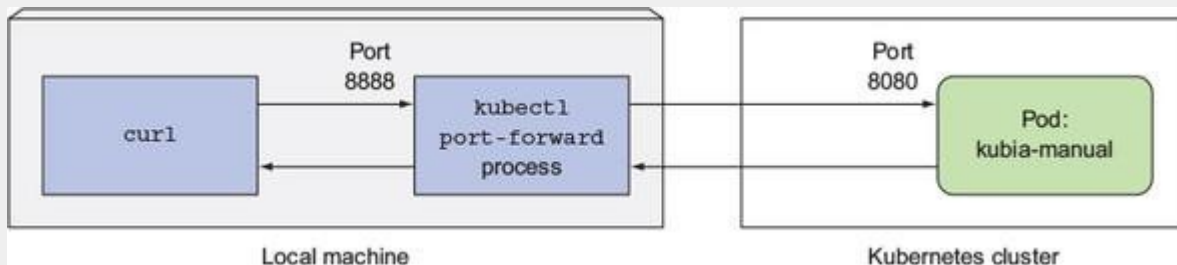
```
apiVersion: v1           1
kind: Pod                 2
metadata:
  name: kubia-manual      3
spec:
  containers:
    - image: luksa/kubia  4
      name: kubia         5
      ports:
        - containerPort: 8080  6
          protocol: TCP
```

> \$ kubectl create -f kubia-manual.yaml

- **1 Descriptor conforms to version v1 of Kubernetes API**
- **2 You're describing a pod.**
- **3 The name of the pod**
- **4 Container image to create the container from**
- **5 Name of the container**
- **6 The port the app is listening on**

# Sending request to pods and port forwarding

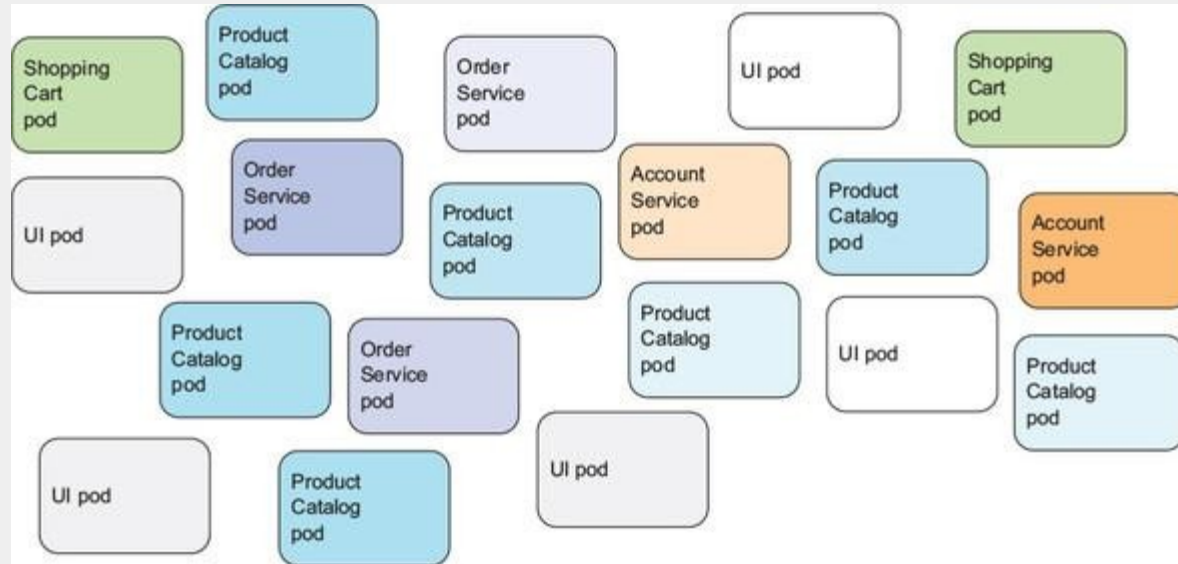
When you need to get access to pods externally



```
$ kubectl port-forward kubia-manual 8888:8080
... Forwarding from 127.0.0.1:8888 -> 8080
... Forwarding from [::1]:8888 -> 8080
```

# Organising pods with labels

When deploying actual applications, most users will end up running many more pods. As the number of pods increases, the need for categorizing them into subsets becomes more and more evident.



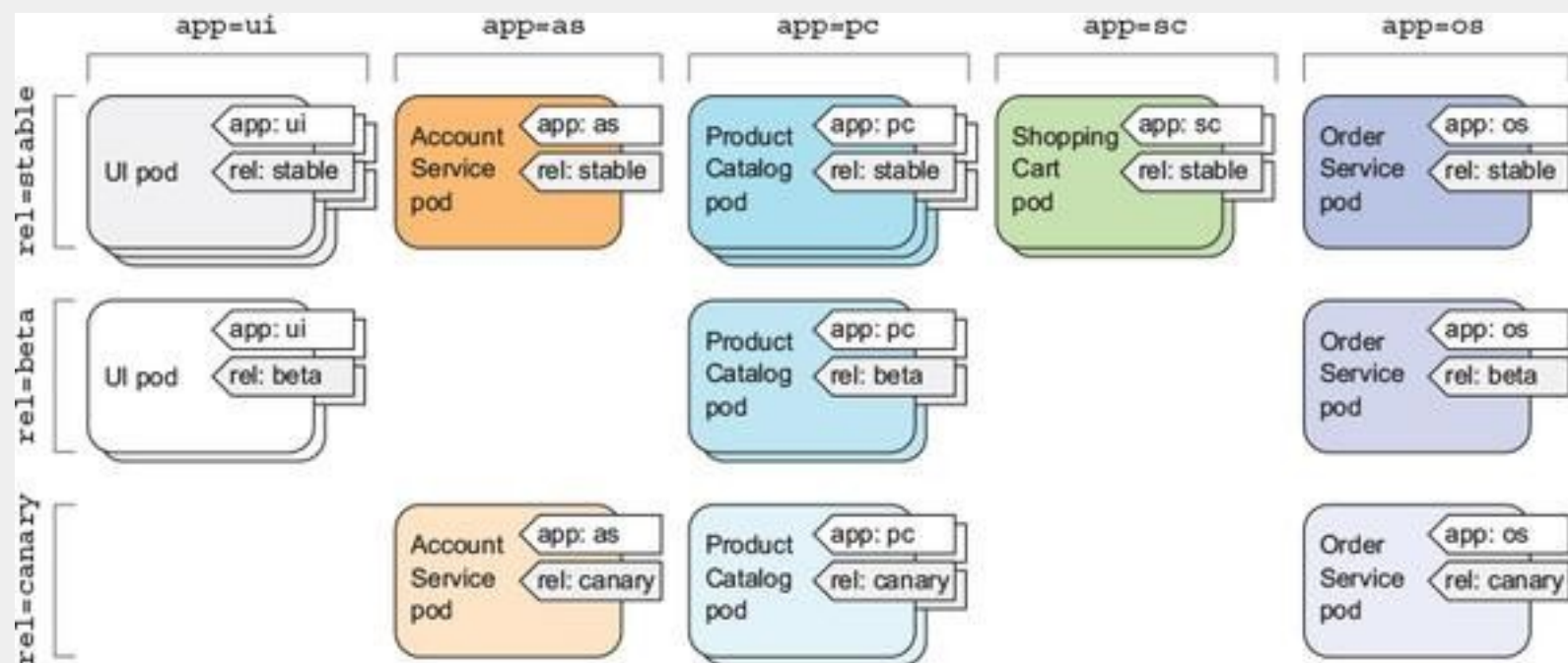
# Introducing labels

Labels are a simple, yet incredibly powerful, Kubernetes feature for organizing not only pods, but all other Kubernetes resources.

Each pod is labelled with 2 labels:-

- *app*, which specifies which app, component, or microservice the pod belongs to.
- *rel*, which shows whether the application running in the pod is a stable, beta, or a canary release.





# Creating labels

Listing 3.3. A pod with labels: kubia-manual-with-labels.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: kubia-manual-v2
  labels:
    creation_method: manual      1
    env: prod                    1
spec:
  containers:
  - image: luksa/kubia
    name: kubia
    ports:
    - containerPort: 8080
      protocol: TCP
```

- **1** Two labels are attached to the pod.

# Deployments and Replica Sets

## Deploying Managed Pods

# Keeping pods healthy

One of the main benefits of using Kubernetes is the ability to give it a list of containers and let it keep those containers running somewhere in the cluster.

# Introducing liveness probe

Kubernetes can check if a container is still alive through *liveness probes*. You can specify a liveness probe for each container in the pod's specification. Kubernetes will periodically execute the probe and restart the container if the probe fails

# Mechanism by which kubernetes probe a container

- An *HTTP GET* probe performs an HTTP GET request on the container's IP address, a port and path you specify. If the probe receives a response, and the response code doesn't represent an error (in other words, if the HTTP response code is 2xx or 3xx), the probe is considered successful.
- A *TCP Socket* probe tries to open a TCP connection to the specified port of the container. If the connection is established successfully, the probe is successful. Otherwise, the container is restarted.
- An *Exec* probe executes an arbitrary command inside the container and checks the command's exit status code. If the status code is 0, the probe is successful. All other codes are considered failures.

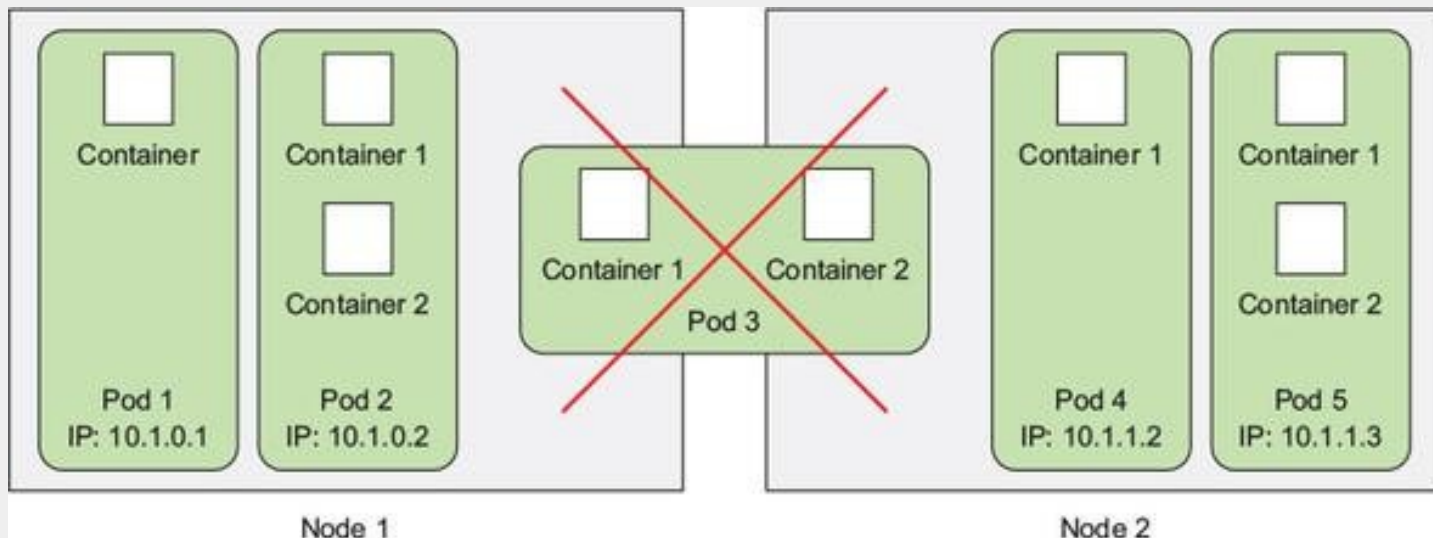
# Creating a Http based liveness probe

Listing 4.1. Adding a liveness probe to a pod: kuba-liveness-probe.yaml

```
apiVersion: v1
kind: pod
metadata:
  name: kuba-liveness
spec:
  containers:
  - image: luksa/kuba-unhealthy      1
    name: kuba
    livenessProbe:                  2
      httpGet:                      3
        path: /                     4
        port: 8080
```

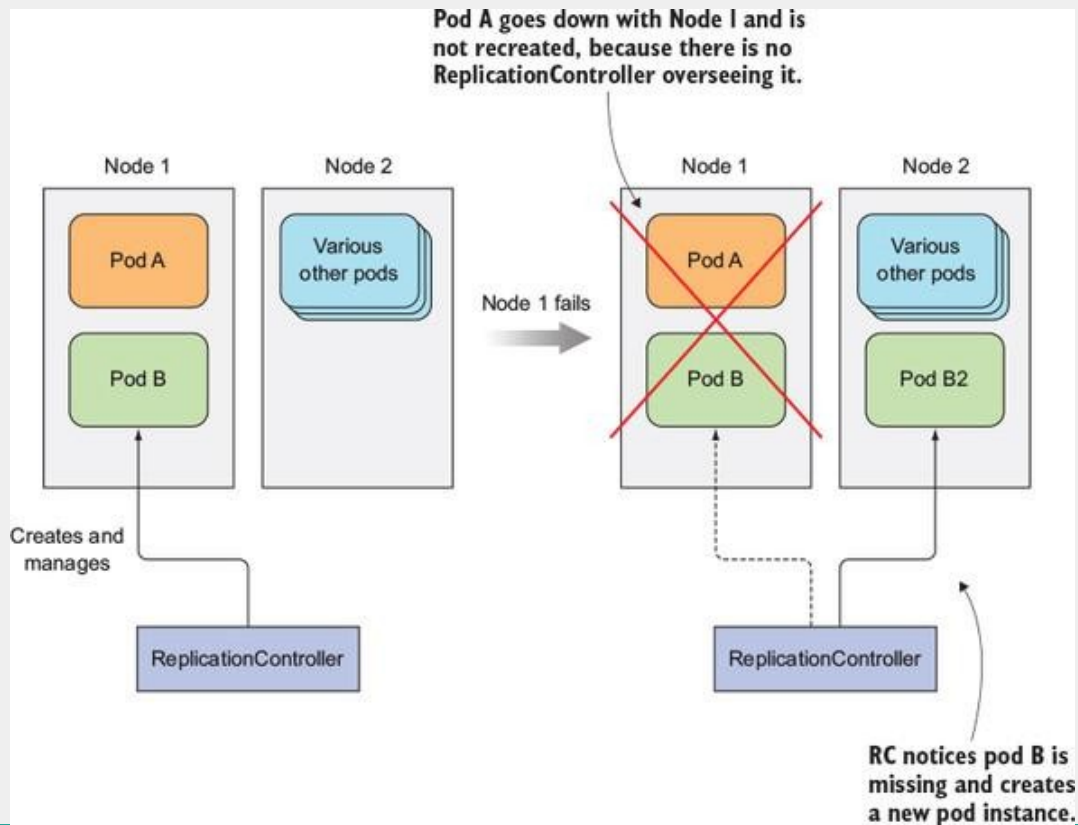
- **1** This is the image containing the (somewhat) broken app.
- **2** A liveness probe that will perform an HTTP GET
- **3** The path to request in the HTTP request
- **4** The network port the probe should connect to

# Containers on pods runs on same worker node





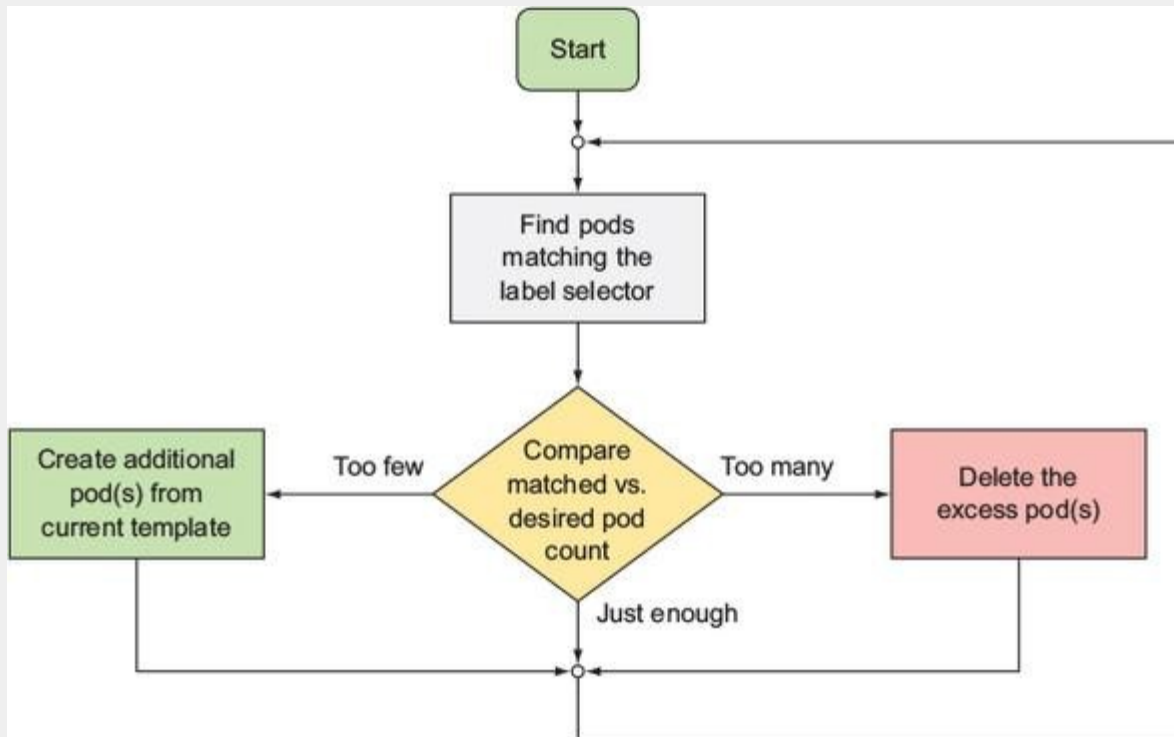
# Introducing Replication Controller



# Operation of Replication Controller

- A ReplicationController constantly monitors the list of running pods and makes sure the actual number of pods of a “type” always matches the desired number.
- If too few such pods are running, it creates new replicas from a pod template.
- If too many such pods are running, it removes the excess replicas.

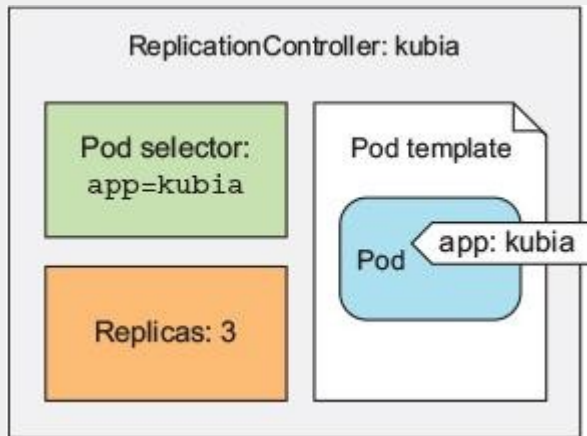
# Operation of Replication Controller



# Replication Controller Components

A ReplicationController has three essential parts

- A *label selector*, which determines what pods are in the ReplicationController's scope
- A *replica count*, which specifies the desired number of pods that should be running
- A *pod template*, which is used when creating new pod replicas



# Benefits of Replication Controller

Like many things in Kubernetes, a ReplicationController, although an incredibly simple concept, provides or enables the following powerful features:

- It makes sure a pod (or multiple pod replicas) is always running by starting a new pod when an existing one goes missing.
- When a cluster node fails, it creates replacement replicas for all the pods that were running on the failed node (those that were under the Replication-Controller's control).
- It enables easy horizontal scaling of pods—both manual and automatic

# Creating a Replication Controller

```
apiVersion: v1
kind: ReplicationController      1
metadata:
  name: kubia                    2
spec:
  replicas: 3                    3
  selector:                      4
    app: kubia                  4
  template:                      5
    metadata:                   5
      labels:                   5
        app: kubia             5
    spec:                       5
      containers:               5
        - name: kubia           5
          image: luksa/kubia    5
          ports:                 5
            - containerPort: 8080 5
```

- **1** This manifest defines a ReplicationController (RC)
- **2** The name of this ReplicationController
- **3** The desired number of pod instances
- **4** The pod selector determining what pods the RC is operating on
- **5** The pod template for creating new pods

# Using Replica sets instead of Replication Controller

- Initially, ReplicationControllers were the only Kubernetes component for replicating pods and rescheduling them when nodes failed.
- Later, a similar resource called a ReplicaSet was introduced.
- It's a new generation of ReplicationController and replaces it completely (ReplicationControllers will eventually be deprecated).

# Comparing Replica sets with Replica Controller

- A ReplicaSet behaves exactly like a ReplicationController, but it has more expressive pod selectors.
- ReplicationController's label selector only allows matching pods that include a certain label.
- A ReplicaSet's selector also allows matching pods that lack a certain label or pods that include a certain label key, regardless of its value.



# Defining Replica sets

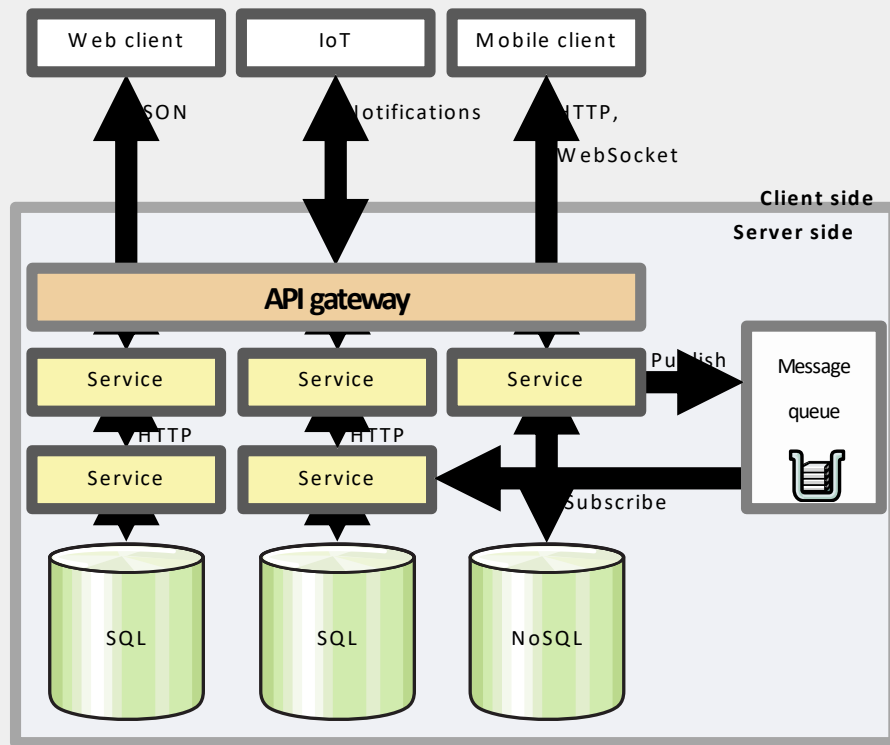
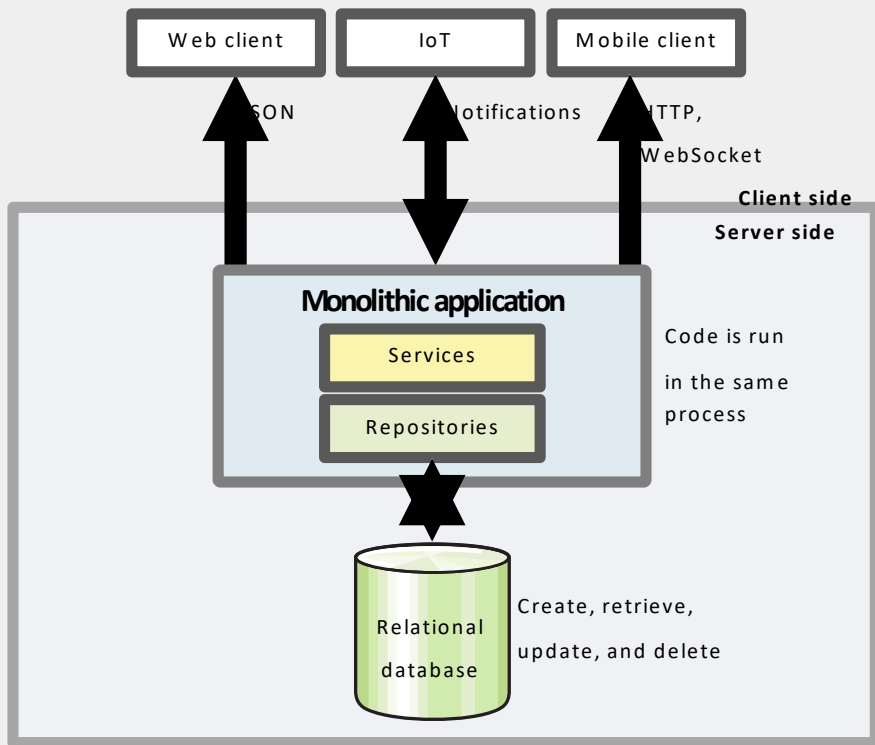
Listing 4.8. A YAML definition of a ReplicaSet: kubia-replicaset.yaml

```
apiVersion: apps/v1beta2      1
kind: ReplicaSet              1
metadata:
  name: kubia
spec:
  replicas: 3
  selector:
    matchLabels:                2
      app: kubia                2
  template:                    3
    metadata:                  3
      labels:                  3
        app: kubia             3
    spec:                      3
      containers:              3
      - name: kubia             3
        image: luksa/kubia      3
```

- **1** ReplicaSets aren't part of the v1 API, but belong to the apps API group and version v1beta2.
- **2** You're using the simpler matchLabels selector here, which is much like a ReplicationController's selector.
- **3** The template is the same as in the ReplicationController.

# Services

# Monolithic vs Microservices Architectures



# Kubernetes in Summary



# What is Kubernetes?



Platform for running applications **abstracting away infrastructure**

# PODS

# What are pods?

- It is the smallest unit of deployment in kubernetes
- Instead of deploying container individually, you deploy pods on kubernetes not containers.
- One pods can have many containers

# Things to remember

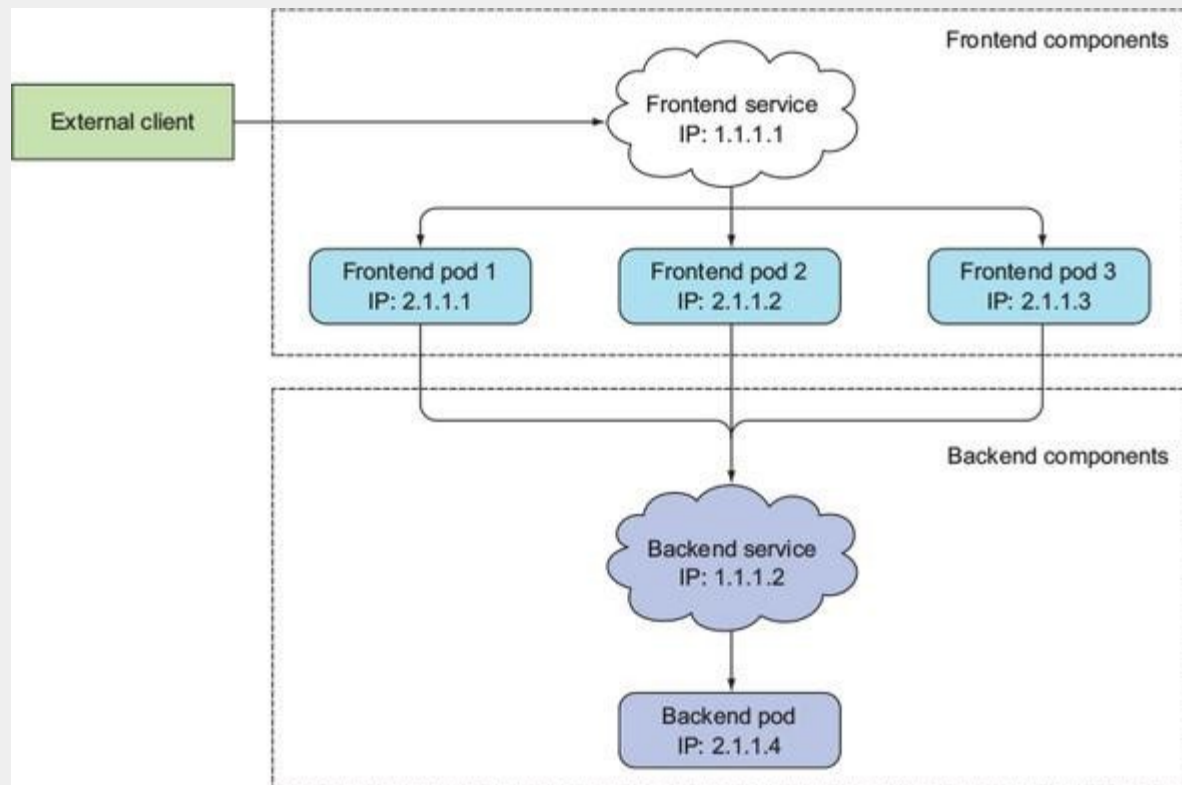
- *Pods are ephemeral*—They may come and go at any time, whether it's because a pod is removed from a node to make room for other pods, because someone scaled down the number of pods, or because a cluster node has failed.
- *Kubernetes assigns an IP address to a pod after the pod has been scheduled to a node and before it's started*—Clients thus can't know the IP address of the server pod up front.
- *Horizontal scaling means multiple pods may provide the same service*—Each of those pods has its own IP address. Clients shouldn't care how many pods are backing the service and what their IPs are. They shouldn't have to keep a list of all the individual IPs of pods. Instead, all those pods should be accessible through a single IP address.



# What are services in Kubernetes?

- A Kubernetes Service is a resource you create to make a single, constant point of entry to a group of pods providing the same service.
- Each service has an IP address and port that never change while the service exists.
- Clients can open connections to that IP and port, and those connections are then routed to one of the pods backing that service.
- Clients of a service don't need to know the location of individual pods providing the service, allowing those pods to be moved around the cluster at any time.

# Explaining services with a example



# Creating a simple service

Listing 5.1. A definition of a service: kubia-svc.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: kubia
spec:
  ports:
    - port: 80                1
      targetPort: 8080        2
    selector:                  3
      app: kubia              3
```

- **1** The port this service will be available on
- **2** The container port the service will forward to
- **3** All pods with the app=kubia label will be part of this service.

# Creating a simple multi port service

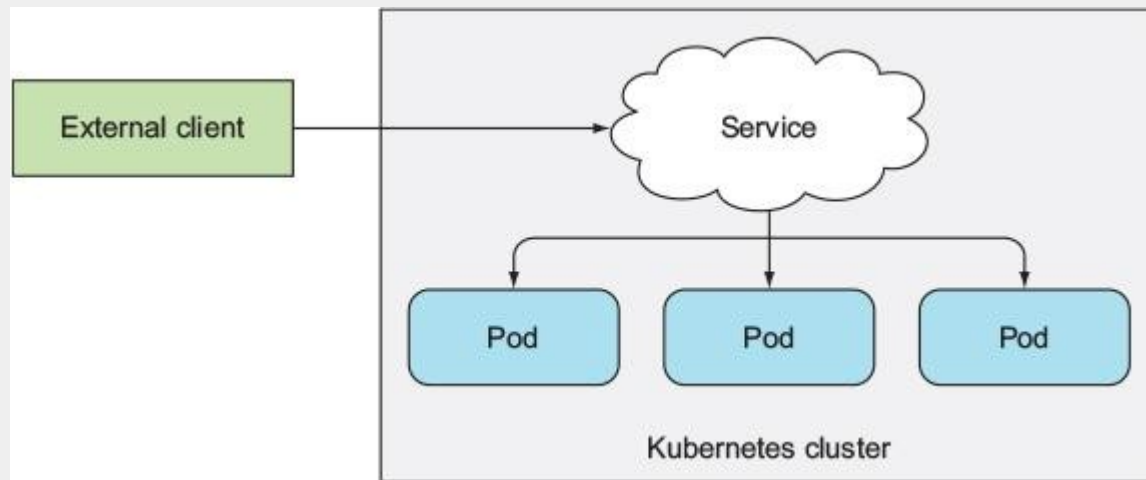
The spec for a multi-port service is shown in the following listing.

**Listing 5.3. Specifying multiple ports in a service definition**

```
apiVersion: v1
kind: Service
metadata:
  name: kubia
spec:
  ports:
    - name: http          1
      port: 80            1
      targetPort: 8080    1
    - name: https         2
      port: 443            2
      targetPort: 8443    2
  selector:               3
    app: kubia            3
```

- **1** Port 80 is mapped to the pods' port 8080.
- **2** Port 443 is mapped to pods' port 8443.
- **3** The label selector always applies to the whole service.

# Exposing services to external clients



# Ways to access service externally

- *Setting the service type to NodePort*—For a NodePort service, each cluster node opens a port on the node itself (hence the name) and redirects traffic received on that port to the underlying service. The service isn't accessible only at the internal cluster IP and port, but also through a dedicated port on all nodes.
- *Setting the service type to LoadBalancer, an extension of the NodePort type*—This makes the service accessible through a dedicated load balancer, provisioned from the cloud infrastructure Kubernetes is running on. The load balancer redirects traffic to the node port across all the nodes. Clients connect to the service through the load balancer's IP.
- *Creating an Ingress resource, a radically different mechanism for exposing multiple services through a single IP address*

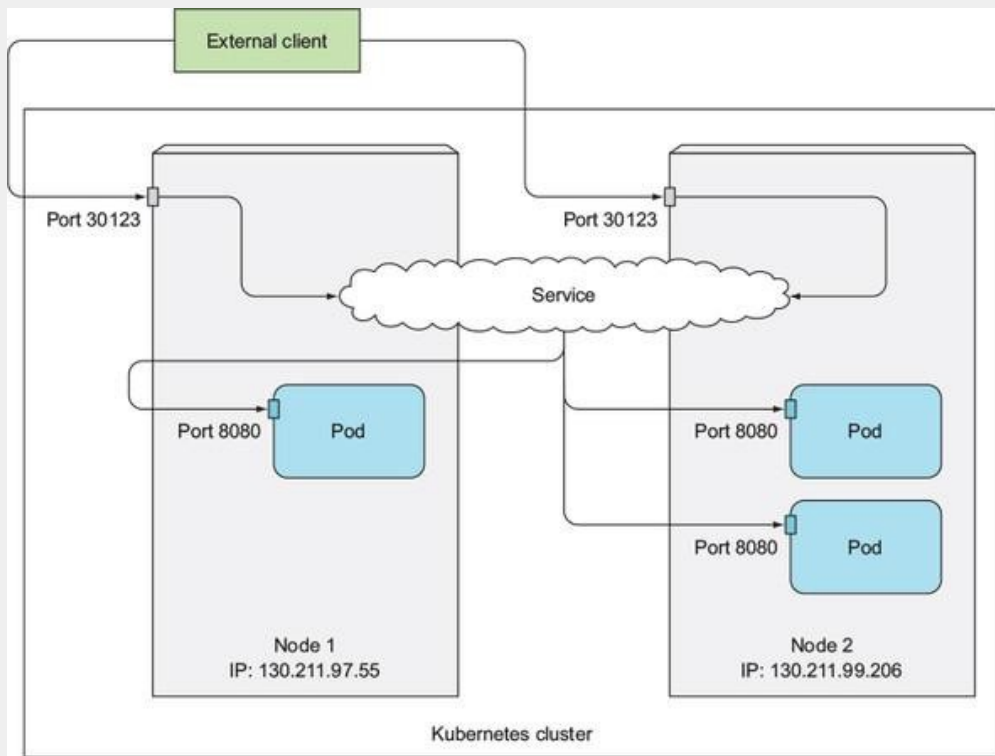
# Exposing Service as Node Port

Listing 5.11. A NodePort service definition: kubia-svc-nodeport.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: kubia-nodeport
spec:
  type: NodePort           1
  ports:
    - port: 80             2
      targetPort: 8080      3
      nodePort: 30123       4
  selector:
    app: kubia
```

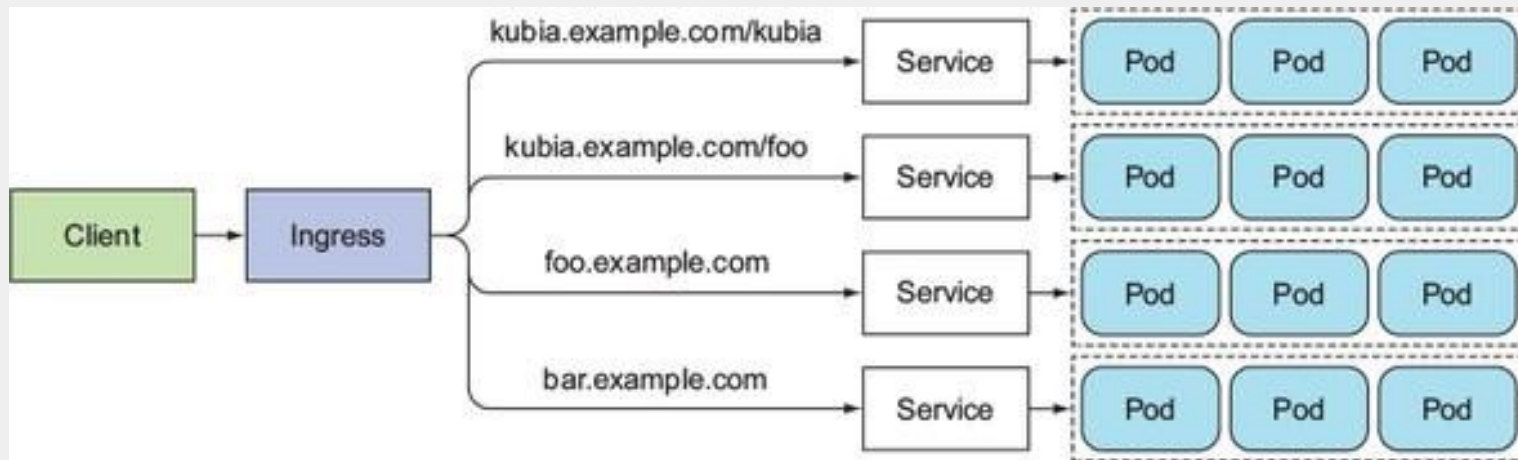
- **1** Set the service type to NodePort.
- **2** This is the port of the service's internal cluster IP.
- **3** This is the target port of the backing pods.
- **4** The service will be accessible through port 30123 of each of your cluster nodes.

# External Client Connecting to Node Port Service





# Exposing Service externally through Ingress



# Creating a Simple Ingress Resource

Listing 5.13. An Ingress resource definition: kuba-ingress.yaml

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: kuba
spec:
  rules:
  - host: kuba.example.com          1
    http:
      paths:
      - path: /                    2
        backend:
          serviceName: kuba-nodeport 2
          servicePort: 80           2
```

- **1** This Ingress maps the kuba.example.com domain name to your service.
- **2** All requests will be sent to port 80 of the kuba-nodeport service.



# HANDS-ON

## Running a Containerized Application on Kubernetes

# Resources

- IBM Page

<https://www.ibm.com/cloud/private>

- Coursera:

<https://www.coursera.org/learn/deploy-micro-kube-icp/>

- Journey:

[https://www-03.ibm.com/services/learning/ites.wss/zz-en?pageType=journey\\_description&journeyId=ICP-LJ02&tag=o-its-01-02](https://www-03.ibm.com/services/learning/ites.wss/zz-en?pageType=journey_description&journeyId=ICP-LJ02&tag=o-its-01-02)

- Join the ICP slack channel:

<https://ibm-cloudplatform.slack.com/>