



Luca Cabibbo  
Architettura  
dei Sistemi  
Software

# Orchestratura di container con Kubernetes

**dispensa asw890**  
ottobre 2023

*Kubernetes is Greek for pilot or helmsman  
(the person holding the ship's steering wheel).*

*Marko Lukša*



## - Riferimenti

- ❑ Luca Cabibbo. **Architettura del Software: Strutture e Qualità**. Edizioni Efestò, 2021.
  - Capitolo 40, **Orchestratura di container**
- ❑ Lukša, M. **Kubernetes in Action**. Manning, 2018.
- ❑ Stoneman, E. **Learn Kubernetes in a Month of Lunches**, Manning, 2021.
- ❑ Kubernetes (version 1.29, 2023)
  - <https://kubernetes.io/>
  - <https://kubernetes.io/docs/home/>



## - Obiettivi e argomenti

### □ Obiettivi

- introdurre Kubernetes
- esemplificare l'orchestrazione di container con Kubernetes

### □ Argomenti

- introduzione a Kubernetes
- architettura di Kubernetes
- risorse Kubernetes
- orchestrazione con Kubernetes
- discussione



## \* Introduzione a Kubernetes



- **Kubernetes** è una piattaforma open source, portabile ed estensibile, per la gestione automatizzata di applicazioni e carichi di lavoro a container sulla base di configurazioni dichiarative
  - il nome Kubernetes (talvolta abbreviato **K8S**) deriva dal greco e significa “timoniere” o “pilota”
  - inizialmente sviluppato da Google (sulla base di 15 anni di esperienza nell'eseguire carichi di lavoro su larga scala a Google), nel 2014 è divenuto un progetto open-source
  - oggi è uno dei sistemi di orchestrazione di container più diffusi
  - per gli obiettivi dell'orchestrazione di container e i principi generali di funzionamento si veda il capitolo sull'orchestrazione di container



- ❑ **Kubernetes** è una piattaforma open source, portabile ed estensibile, per la gestione automatizzata di applicazioni e carichi di lavoro a container sulla base di configurazioni dichiarative
  - in pratica, Kubernetes consente di definire e gestire una piattaforma costituita da un cluster di macchine in cui eseguire una o più applicazioni a container
  - può essere eseguito in una varietà di ambienti – in un singolo PC (come ambiente per lo sviluppo e l'apprendimento) oppure come un cluster di macchine fisiche o virtuali, on premises oppure nel cloud (come ambiente di produzione)
  - nel cloud è possibile creare facilmente un cluster Kubernetes in un gruppo di macchine virtuali – ma è ancora più semplice usare uno dei numerosi servizi completamente gestiti per container basati su Kubernetes – come Google **GKE** (**Google Kubernetes Engine**), Amazon **EKS** (**Elastic Kubernetes Service**) oppure Microsoft **AKS** (**Azure Kubernetes Service**)

5

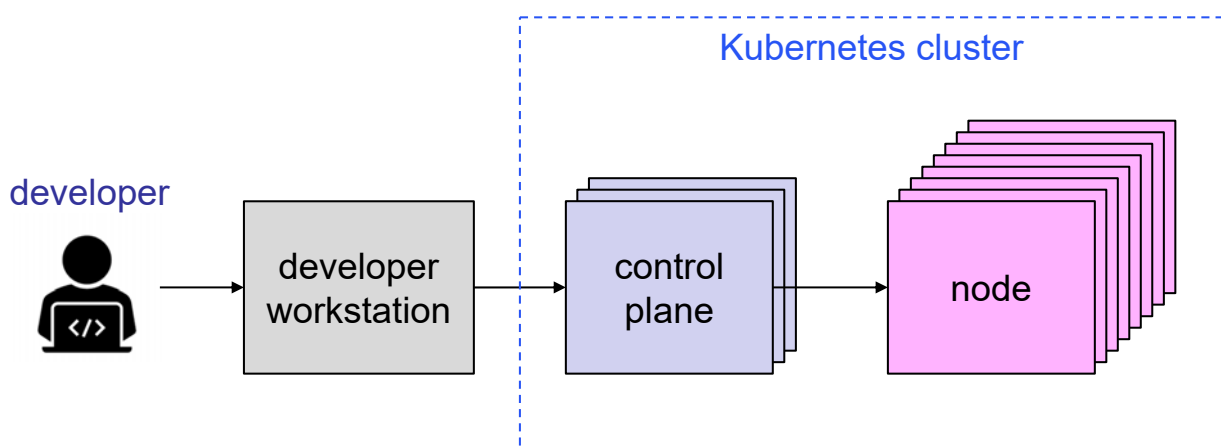
Orchestrazione di container con Kubernetes

Luca Cabibbo ASW



## \* Architettura di Kubernetes

- ❑ Descriviamo ora l'architettura di Kubernetes



- un cluster Kubernetes
  - ha l'obiettivo di consentire l'esecuzione di una o più applicazioni a container (in genere una sola applicazione per cluster)
  - è basato su diversi componenti software – che vengono eseguiti in due tipi di macchine

6

Orchestrazione di container con Kubernetes

Luca Cabibbo ASW



# Cluster Kubernetes

- Un *cluster Kubernetes* è composto da due tipi di macchine
  - uno o più *nodi (nodi worker)* – in cui vengono effettivamente eseguiti i container delle applicazioni (chiamati *pod* in Kubernetes)
  - una o più macchine per il *control plane* – il control plane controlla e gestisce l'intero cluster Kubernetes
  - il control plane gestisce lo stato del cluster, ma (in genere) le sue macchine non partecipano all'effettiva esecuzione delle applicazioni, che viene invece effettuata sui nodi worker
  - gli sviluppatori possono rilasciare in un cluster Kubernetes le proprie applicazioni a container
    - in questa dispensa, usiamo il termine “sviluppatore” per indicare, genericamente, chi si occupa di rilasciare un'applicazione in un cluster Kubernetes

7

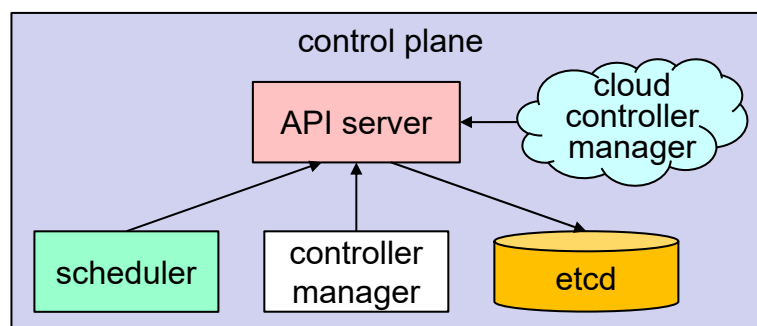
Orchestrazione di container con Kubernetes

Luca Cabibbo ASW



## Control plane

- Il control plane gestisce il cosiddetto *Kubernetes Control Plane*, che controlla il cluster e lo fa funzionare



- le macchine del control plane vengono in genere replicate, per garantire alta disponibilità e scalabilità (dell'orchestratore)

8

Orchestrazione di container con Kubernetes

Luca Cabibbo ASW



## Control plane

- Il control plane ospita i seguenti componenti software
  - *kube-apiserver* (Kubernetes API Server)
    - il front-end del Control Plane, che gestisce la comunicazione tra i diversi componenti software di Kubernetes
  - *kube-scheduler*
    - lo scheduler, che schedula i pod delle applicazioni – ovvero, assegna un nodo worker a ciascun pod che deve essere eseguito nel cluster
  - *kube-controller*
    - il controller manager, che esegue funzioni a livello di cluster per controllare (in senso attivo) che lo stato delle applicazioni rilasciate corrisponda a quello desiderato – ad es., controlla i nodi e i pod
  - *etcd* – un data store distribuito che gestisce in modo persistente e affidabile i dati e la configurazione del cluster



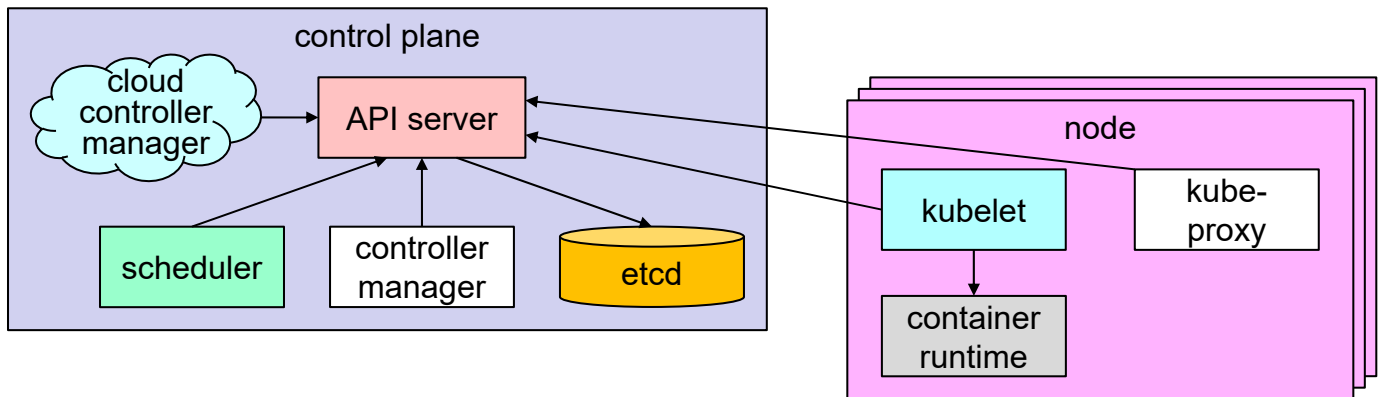
## Control plane

- Il control plane ospita i seguenti componenti software
  - *cloud-controller-manager* (è presente solo se il cluster viene eseguito nel cloud)
    - gestisce la logica di controllo specifica per il cloud
    - collega il cluster alle API del provider di cloud utilizzato, e separa i componenti che interagiscono con la piattaforma di cloud da quelli che interagiscono solo con il cluster



## Nodi (worker)

- I nodi (worker) eseguono i container (pod) che costituiscono le applicazioni



- vengono in genere usati più nodi (che possono anche essere aggiunti dinamicamente al cluster), per garantire alta disponibilità e scalabilità (delle applicazioni)



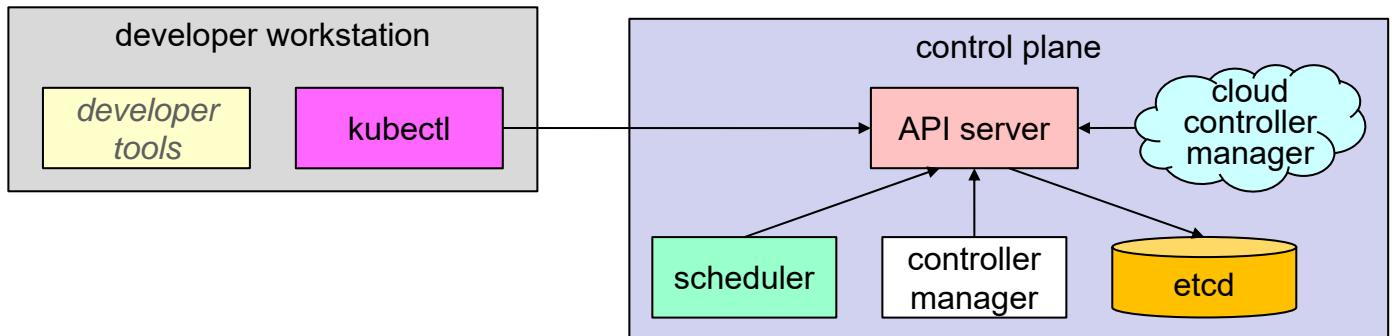
## Nodi (worker)

- Ogni nodo worker ospita i seguenti componenti software
  - un *container runtime*
    - per eseguire i container associati ai pod (come containerd oppure Docker) – non è parte di Kubernetes
  - *kubelet*
    - un agente che gestisce i pod di quel nodo – è un intermediario tra Kubernetes e il container runtime
  - *kube-proxy* (Kubernetes Service Proxy)
    - un proxy di rete che inoltra e bilancia il traffico di rete tra i pod



## Client per gli sviluppatori

- Inoltre, sul PC dello sviluppatore di un'applicazione a container va utilizzato un componente software per interagire con il cluster Kubernetes



- **kubectl**
  - un'interfaccia dalla linea di comando (CLI) che accetta comandi dallo sviluppatore e li inoltra (come chiamate REST) al cluster Kubernetes



## Add-on

- Oltre a questi componenti software, vengono in genere utilizzati anche degli **add-on** per estendere le funzionalità di base di Kubernetes e per fornire dei servizi utili alle applicazioni a container
  - ogni cluster Kubernetes deve avere un **cluster DNS**, un server DNS che sostiene la comunicazione tra i container delle applicazioni – ad es., CoreDNS
    - questo è l'unico add-on strettamente necessario
  - è inoltre comune utilizzare un add-on per gestire una rete per la comunicazione tra pod (**pod network**) – ad es., Project Calico
  - ulteriori esempi di add-on sono una dashboard (una web UI per rilasciare applicazioni e gestire le risorse del cluster) oppure strumenti per la gestione delle reti, degli ingress (discussi più avanti), per il monitoraggio e per il logging



## Add-on

- ❑ Oltre a questi componenti software, vengono in genere utilizzati anche degli *add-on* per estendere le funzionalità di base di Kubernetes e per fornire dei servizi utili alle applicazioni a container
  - gli add-on vengono eseguiti nel cluster come risorse Kubernetes, dunque in modo analogo alle risorse delle applicazioni
    - alcuni add-on vengono eseguiti nel control plane (ad es., CoreDNS), ma altri add-on vengono invece eseguiti nei nodi worker
  - complessivamente, le funzionalità di orchestrazione vengono svolte dai componenti software di base di Kubernetes insieme agli add-on utilizzati



## Reti e indirizzi di rete

- ❑ Alcune cose utili da sapere sulla gestione delle reti e degli indirizzi IP in Kubernetes
  - ogni macchina (fisica o virtuale) del cluster ha un proprio indirizzo IP “esterno” nella rete in cui è collocato – ad es., 10.11.1.71
  - all’interno del cluster viene inoltre definita una rete privata (*pod network*) per la comunicazione tra pod – ad es., la rete 192.168.0.0/16 – a ciascun pod viene assegnato un indirizzo IP in questa rete
  - all’interno del cluster viene definita anche un’altra rete privata (*service network*) per i servizi (discussi più avanti), per semplificare la comunicazione tra i pod – ad es., la rete 10.96.0.0/12
  - tutti i pod rilasciati nel cluster vengono configurati automaticamente per usare il cluster DNS





## L'ambiente kube-cluster



- Nel repository GitHub del corso è disponibile un ambiente **kube-cluster** per Kubernetes
  - una VM per il control plane – **kube-1** (10.11.1.71)
  - due VM per i nodi worker – **kube-2** (10.11.1.72) e **kube-3** (10.11.1.73)
  - gli add-on utilizzati nell'ambiente **kube-cluster** sono
    - CoreDNS come cluster DNS
    - Calico Project per la gestione della pod network
    - NGINX Ingress Controller – un ingress controller basato su NGINX, che ascolta per HTTP sulle porte 80 (solo sui nodi worker) e 31080 (su tutte le macchine del cluster)



## L'ambiente kube-cluster



- Nel repository GitHub del corso è disponibile un ambiente **kube-cluster** per Kubernetes
  - una VM per lo sviluppatore – **kube-dev** (10.11.1.131), con Java, Python, Docker e kubectl
  - nella rete (file **/etc/hosts** di ogni VM) sono configurati i seguenti alias
    - **kube-cluster** per le VM del cluster **kube-1**, **kube-2** e **kube-3**
      - con l'ingress controller che su tutte le macchine ascolta sulla porta 31080
    - **kube-control-plane** per le VM del control plane **kube-1**
    - **kube-node** per i nodi worker **kube-2** e **kube-3**
      - con l'ingress controller che sui nodi worker ascolta sulla porta 80



## \* Risorse Kubernetes

- Kubernetes definisce un certo numero di astrazioni – chiamate *risorse* (*API resource* oppure *Kubernetes resource object*) – per la configurazione dichiarativa delle applicazioni a container
  - le risorse Kubernetes più importanti sono
    - un *pod* incapsula un'istanza di container da eseguire nel cluster – costituisce l'unità di deployment nell'orchestratore
    - un *replica set* consente di gestire automaticamente l'esecuzione di una o più repliche di un pod
    - un *daemon set* consente di avere una replica di un pod per ciascun nodo worker del cluster
    - un *service* definisce un punto d'ingresso unico e stabile per un insieme logico di pod che forniscono uno stesso servizio
    - un *deployment* consente di gestire in modo dichiarativo il rilascio e l'aggiornamento di un'applicazione



## Risorse Kubernetes

- Ecco alcune nozioni correlate alle risorse Kubernetes
  - una *tipologia di risorsa* rappresenta un'astrazione (una classificazione) delle entità software gestite da Kubernetes
    - ad es., il concetto di pod
  - un'*istanza di risorsa* rappresenta un'entità software runtime gestita dal cluster Kubernetes
    - ad es., un'istanza di un pod per un certo servizio
  - una *specifica di risorsa* (o *configurazione di risorsa*) rappresenta lo stato desiderato di un gruppo di una o più istanze di risorse (con caratteristiche simili)
    - ad es., la specifica di un pod per un certo servizio
  - in questa dispensa, usiamo il termine generico “risorsa” (senza nessuna qualificazione) quando il suo significato può essere dedotto dal contesto in cui compare



## Specifiche di risorse

- ❑ Le specifiche di risorse consentono una configurazione dichiarativa delle applicazioni a container
  - infatti consentono agli sviluppatori di descrivere lo stato desiderato (a runtime) del cluster Kubernetes
    - ciascuna specifica di risorsa descrive un “intento” richiesto durante l’esecuzione di un’applicazione
    - quando viene rilasciata una risorsa, Kubernetes inizia a lavorare costantemente per garantire l’esistenza delle corrispondenti istanze di risorsa
  - per creare e rilasciare una risorsa, bisogna dunque fornire una specifica della risorsa, che contiene informazioni sulla risorsa (ad es., la sua tipologia e il suo nome) e sul suo stato desiderato
    - questo avviene mediante un approccio di tipo *infrastructure-as-code* – in pratica, di solito, mediante un file YAML



## - Esempio

- ❑ Come primo esempio, consideriamo un servizio REST **hello** che ascolta al path / sulla porta 8080 e risponde con un saluto
  - si può realizzare come una semplice applicazione Spring Boot
  - ecco il relativo **Dockerfile**

```
# Dockerfile per il servizio hello
FROM eclipse-temurin:17-jdk
ADD build/libs/hello.jar hello.jar
EXPOSE 8080
ENTRYPOINT ["java", "-Xmx128m", "-Xms128m", "-jar", "hello.jar"]
```

- l’immagine Docker è stata salvata su Docker Hub come **aswroma3/hello-kube:2023-10** e come **aswroma3/hello-kube:latest**
- discutiamo il rilascio in Kubernetes di un tale servizio applicativo



## - Pod

- Un *pod* è l'unità di esecuzione di un'applicazione a container
  - intuitivamente, un pod rappresenta e incapsula un'istanza di container da eseguire nel cluster – a ogni pod sono associate delle risorse computazionali, come un indirizzo IP unico e delle risorse di storage
  - un pod è la più semplice e piccola unità che uno sviluppatore può creare o rilasciare in Kubernetes



## Pod

- Ecco la specifica di un pod *hello-pod* (file *hello-pod.yaml*)

```
apiVersion: v1
kind: Pod
metadata:
  name: hello-pod
  labels:
    app: hello
spec:
  containers:
  - name: hello-container
    image: aswroma3/hello-kube:latest
```

- nella specifica di un pod, è necessario far riferimento a delle immagini Docker in un registry (che deve essere accessibile da tutti i nodi del cluster) – non è invece possibile far riferimento a immagini nella cache Docker oppure a dei **Dockerfile**
- tutte le immagini usate in questa dispensa sono disponibili su Docker Hub



# Kubectl

- Ecco alcuni comandi **kubectl** di uso generale
  - **kubectl create *resource*** o **kubectl create -f *resource-file.yaml***
    - crea la risorsa specificata o le risorse del file specificato
  - **kubectl apply -f *resource-file.yaml***
    - crea o aggiorna le risorse del file specificato
  - **kubectl delete -f *resource-file.yaml***
    - elimina le risorse del file specificato
  - **kubectl get *resource-type***
    - elenca le risorse della tipologia specificata – ad es., **pods** (**po**), **nodes** (**no**), **services** (**svc**), ...
  - **kubectl describe *resource-type/resource-name***
    - fornisce informazioni sulla risorsa specificata
  - **kubectl delete *resource-type/resource-name***
    - elimina la risorsa specificata

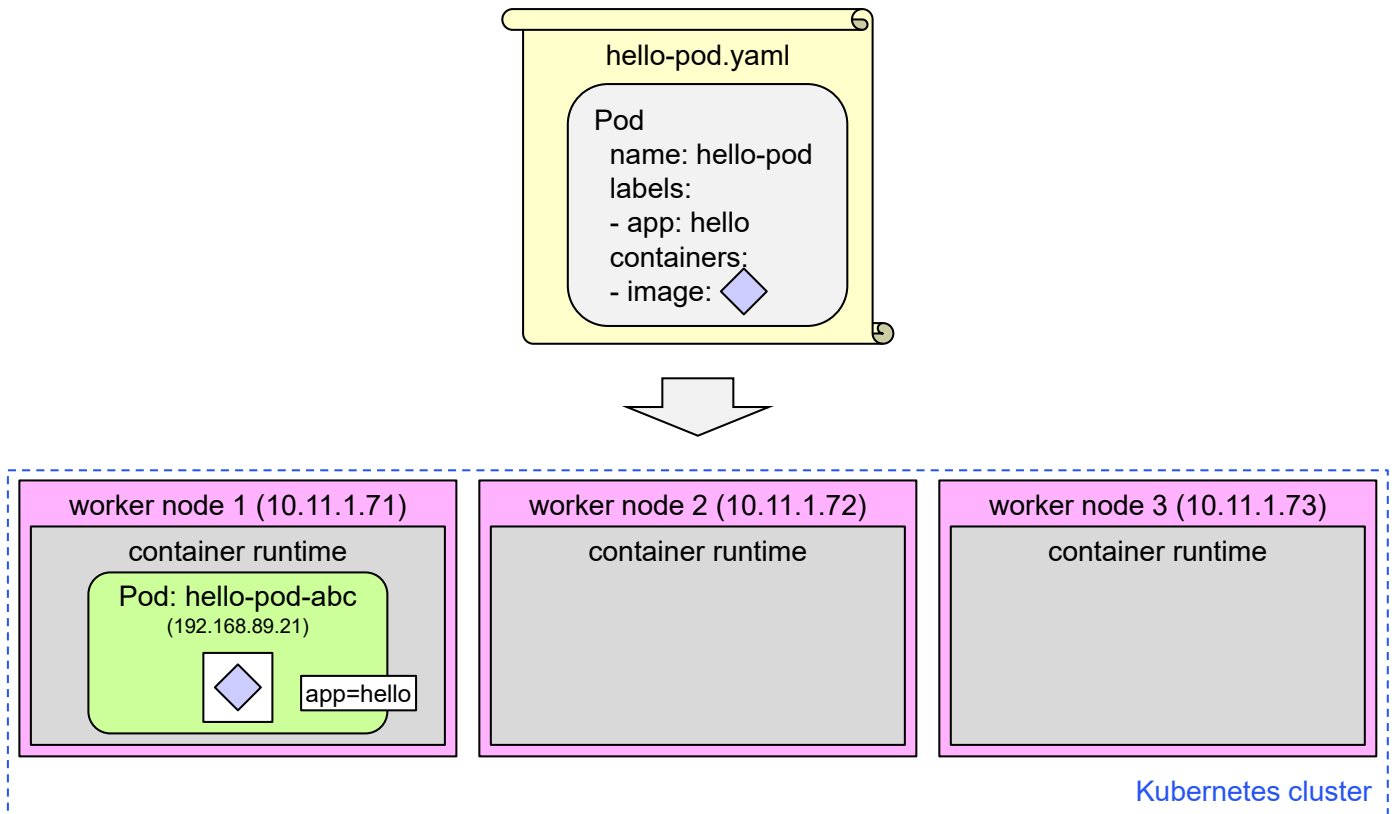


# Kubectl e pod

- Ecco alcuni comandi **kubectl** per la gestione dei pod
  - **kubectl apply -f *hello-pod.yaml***
    - in questo caso, crea il pod **hello-pod**
  - **kubectl get pods [ -o wide ]**
    - elenca i pod rilasciati nel cluster [mostrando una descrizione estesa]
  - **kubectl describe pods/*hello-pod***
    - fornisce informazioni sul pod specificato – come l'immagine, il container per il pod e il suo indirizzo IP
    - è possibile accedere al pod mediante il suo indirizzo IP
      - attenzione: per ora è possibile accedere a questo servizio applicativo solo dalle macchine interne del cluster (ad es. **kube-1**)
  - **kubectl delete pods/*hello-pod***
    - elimina il pod specificato



## Rilascio di un pod nel cluster



27

Orchestrazione di container con Kubernetes

Luca Cabibbo ASW



## Rilascio di un pod nel cluster

- Il rilascio di un pod nel cluster avviene in questo modo
  - uno sviluppatore richiede (tramite **kubectl**) il rilascio di un pod nel cluster
  - **kubectl** inoltra la richiesta di creazione del pod all'API server
  - l'API server chiede allo scheduler di selezionare un nodo worker in cui creare e avviare il pod
  - l'API server notifica la richiesta di creazione del pod al kubelet del nodo worker selezionato
  - il kubelet del nodo selezionato chiede al container runtime locale di creare e avviare un nuovo container per il pod

28

Orchestrazione di container con Kubernetes

Luca Cabibbo ASW



## Pod con più container



- I pod possono essere usati in due modi
  - **pod a singolo container** – è il caso più comune – in questo caso, si può pensare a un pod come a un'entità che incapsula un singolo container, con Kubernetes che gestisce i pod (e non direttamente i container, che vengono invece gestiti dal container runtime, ad es., containerd)
  - **pod multi-container** – per eseguire più container in un pod – questo è utile se i diversi container del pod devono essere colocalizzati (in un singolo nodo del cluster), ad es., perché devono condividere delle risorse (in genere, dei volumi)
    - tutti i container di un pod vengono infatti rilasciati in uno stesso nodo worker
    - poiché questi container condividono l'indirizzo IP del pod, possono comunicare facilmente tra loro (su localhost) – ma devono comunicare con l'esterno del pod mediante porte distinte



## Specifica di un pod

- La specifica di un pod (o di ogni altra risorsa Kubernetes) può contenere numerosi campi
  - la tipologia (**kind**) di risorsa – ad es., Pod
  - metadati della risorsa – tra cui il nome, il namespace e delle etichette
  - la specifica (**spec**) della risorsa – nel caso di un pod
    - i container che compongono il pod, ciascuno con
      - nome
      - immagine
      - porte su cui il container comunica
      - variabili d'ambiente
      - ...



## Label

- I metadati di una risorsa Kubernetes possono contenere delle etichette (zero, una o più)
  - ogni etichetta (*label*) ha una chiave e un valore
    - nell'ambito di un'applicazione, è comune associare a ciascuna risorsa almeno un'etichetta **app** che specifica il nome dell'applicazione a cui si riferisce quella risorsa
    - nelle applicazioni a microservizi, oltre all'etichetta **app** (uguale per tutti i microservizi) è anche comune associare a ciascuna risorsa anche un'etichetta **service** che specifica il microservizio a cui si riferisce quella risorsa – a un microservizio possono essere associate più risorse



## Label

- I metadati di una risorsa Kubernetes possono contenere delle etichette (zero, una o più)
  - come discuteremo più avanti, le etichette sono importanti per consentire la selezione delle risorse
    - infatti, la selezione delle risorse avviene in genere sulla base delle etichette – e non sulla base dei nomi delle risorse (come si potrebbe invece pensare)
    - in questa introduzione a Kubernetes, utilizziamo le sole etichette **app** e, più avanti, anche **service**





## Pod – discussione

- ❑ Alcune osservazioni sui pod
  - i pod (e i loro indirizzi di rete) sono effimeri – ogni volta che viene creato (o ricreato) un pod (il che è possibile e comune), gli viene assegnato un indirizzo di rete differente
    - questo solleva il problema della comunicazione con i pod – sia della comunicazione tra pod (all'interno del cluster) che della comunicazione con i client finali (dall'esterno del cluster)
    - il problema della comunicazione tra e con i pod è risolto in Kubernetes dai service (descritti più avanti)
  - Kubernetes raccomanda di non creare pod direttamente
    - piuttosto, consigli di crearli indirettamente mediante l'uso di controller (descritti più avanti)



## - Controller

- ❑ Un *controller* è una risorsa Kubernetes che ha lo scopo di gestire e controllare un gruppo di pod correlati (in genere, basati su un'identica specifica di container)
  - i controller più comuni sono quelli che consentono di gestire e controllare un numero stabile di repliche di un pod – ne esistono di diversi tipi
    - un *replica set* consente di specificare il numero di repliche desiderato di un tipo di pod
    - un *daemon set* consente di avere esattamente una replica di un pod per ciascun nodo worker del cluster
      - i daemon set sono utili, ad es., per eseguire un agente (demone) di monitoraggio in ciascun nodo del cluster, oppure per la raccolta dei log del nodo
      - per sicurezza, i nodi del control plane di solito non eseguono repliche di questi pod



## Controller

- Un **controller** è una risorsa Kubernetes che ha lo scopo di gestire e controllare un gruppo di pod correlati (in genere, basati su un'identica specifica di container)
  - i controller hanno lo scopo di controllare il proprio gruppo di pod in modo attivo
    - un controller non solo consente di creare un gruppo di pod, ma anche di garantirne l'esistenza, in modo stabile
    - ad es., se uno dei pod di un **replica set** fallisce, allora il pod viene rischedulato (a livello di cluster) e riavviato
      - un pod fallito di un replica set potrebbe essere riallocato in un nodo worker diverso
      - in ogni caso, ad un pod che viene ricreato viene di solito assegnato un indirizzo di rete differente da quello del pod che è fallito



## Controller



- Un **controller** è una risorsa Kubernetes che ha lo scopo di gestire e controllare un gruppo di pod correlati (in genere, basati su un'identica specifica di container)
  - ulteriori tipi di controller
    - un **job** consente di garantire che un numero specificato di pod di un certo tipo vengano creati e terminino con successo
    - uno **stateful set** consente di gestire applicazioni stateful
      - in modo simile a un replica set, garantisce l'esistenza di un numero desiderato di repliche di un tipo di pod
      - tuttavia, in un replica set i diversi pod sono considerati equivalenti e intercambiabili
      - invece, in uno stateful set, i pod non sono intercambiabili, ma sono ordinati e hanno un'identità persistente che viene mantenuta nelle successive rischedulazioni dei pod – questo è utile per pod che hanno dei volumi persistenti



# Replica Set

- Ecco la specifica di un replica set (file `hello-rs.yaml`)

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: hello-rs
spec:
  replicas: 2
  selector:
    matchLabels:
      app: hello
  template:
    metadata:
      name: hello-pod
      labels:
        app: hello
    spec:
      containers:
        - name: hello-container
          image: aswroma3/hello-kube:latest
```

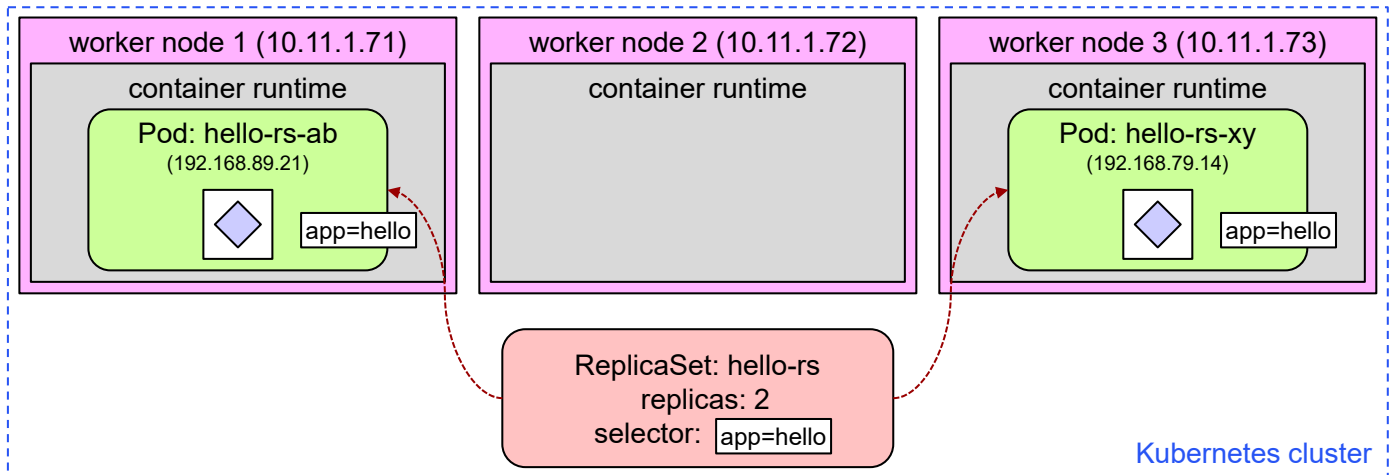
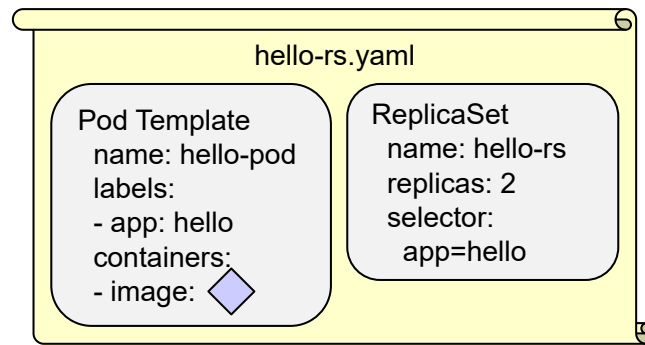


# Replica Set

- Nell'esempio, si noti l'uso dei seguenti campi
  - il campo `template` definisce una specifica di pod
    - il replica set si occuperà di controllare la creazione del numero desiderato di repliche di pod basati su questo template
    - si ricordi la raccomandazione di Kubernetes di non creare pod direttamente
  - il campo `replicas` specifica il numero di repliche desiderato
  - il campo `selector` specifica il criterio di selezione (basato su etichette) dei pod che fanno parte di questo replica set



# Replica Set



39

Orchestrazione di container con Kubernetes

Luca Cabibbo ASW



## - Deployment

- Un **deployment** è un controller Kubernetes di alto livello che consente una specifica dichiarativa di pod e replica set
  - la specifica di un deployment è simile a quella di un replica set
    - in pratica, Kubernetes gestisce un deployment mediante la creazione di un replica set per il deployment
  - Kubernetes raccomanda anche di non creare replica set direttamente
    - piuttosto, consiglia di crearli indirettamente mediante l'uso di deployment
  - inoltre, con un deployment è semplice effettuare la scalatura (in alto o in basso) di un insieme di pod, nonché di effettuare un rolling update o un rollback

40

Orchestrazione di container con Kubernetes

Luca Cabibbo ASW



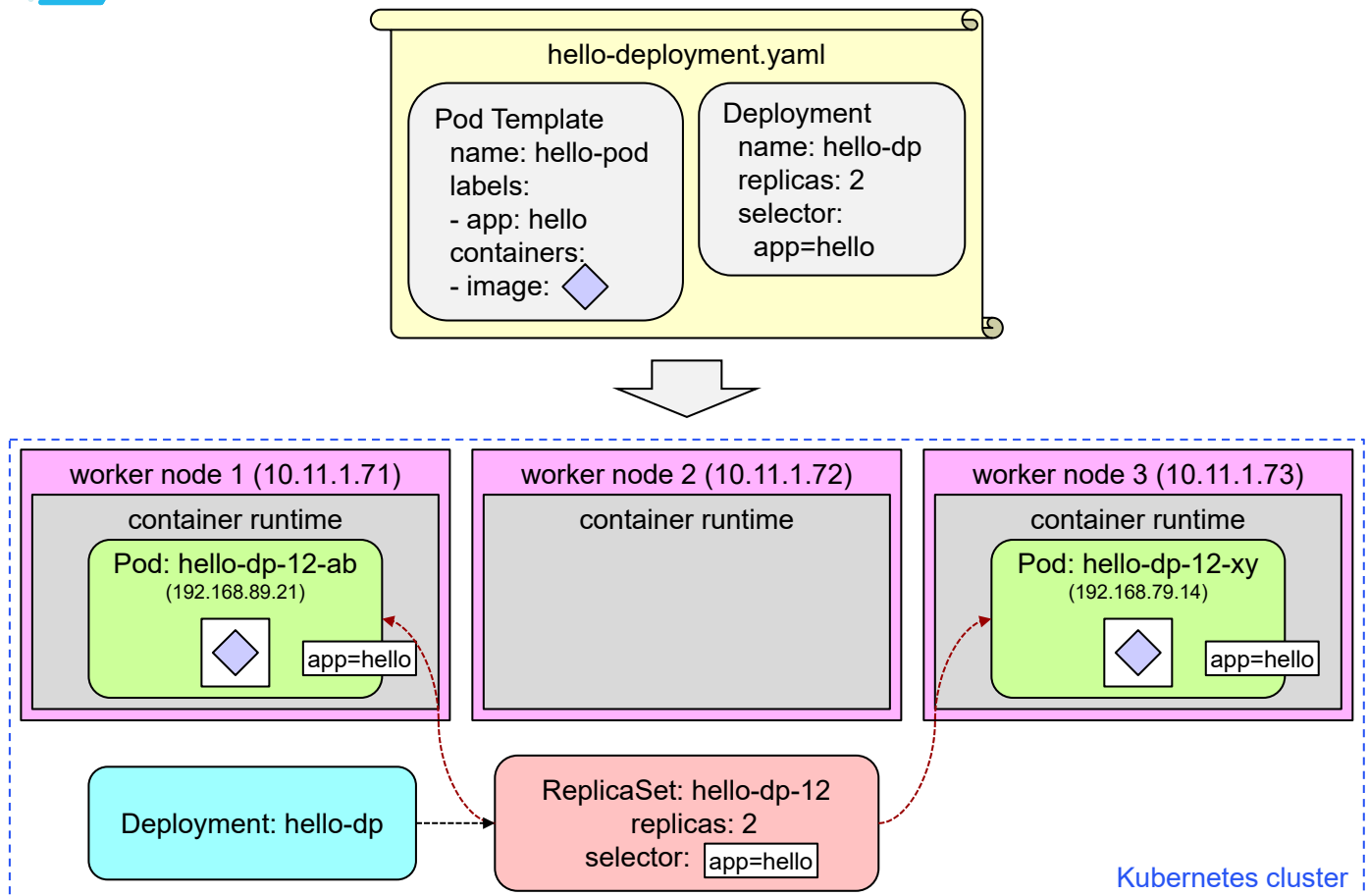
# Deployment

- Ecco la specifica di un deployment (file `hello-deployment.yaml`)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-deploy
spec:
  replicas: 2
  selector:
    matchLabels:
      app: hello
  template:
    metadata:
      name: hello-pod
    labels:
      app: hello
    spec:
      containers:
        - name: hello-container
          image: aswroma3/hello-kube:latest
```



# Deployment





## - Un esperimento

- Prima di andare avanti, facciamo questo esperimento
  - avviamo l'applicazione con `kubectl apply -f hello-deployment.yaml`
  - eseguiamo il comando `kubectl get all -o wide`
    - a questo punto, dovremmo vedere le istanze di risorse mostrate nella figura precedente (anche se con nomi diversi)
      - un deployment, un replica set e due pod
  - “uccidiamo” ora uno dei pod, con il comando `kubectl delete pod/nome-pod`
  - eseguiamo di nuovo il comando `kubectl get all -o wide`
    - a questo punto, dovremmo vedere ancora un deployment, un replica set e due pod
      - uno dei pod è però diverso da quello che è stato “ucciso” – e, inoltre, che il suo indirizzo IP è probabilmente diverso da quello del pod che è stato “ucciso”



## Un esperimento

- Questo esperimento consente di comprendere una frase scritta in precedenza
  - “quando viene rilasciata una risorsa, Kubernetes inizia a lavorare costantemente per garantire l'esistenza delle corrispondenti istanze di risorsa”
  - quando rilasciamo un controller (come un replica set o un deployment), Kubernetes si occupa non solo di creare le risorse corrispondenti – ma si occupa anche di monitorarle e, nel caso di un loro fallimento, di ricrearle – sulla base delle specifiche fornite



## - Service (prima parte)

- Consideriamo ora il problema della comunicazione tra e con i pod
  - un primo problema da affrontare è che i pod (e i loro indirizzi di rete) sono effimeri – ogni volta che viene creato (o ricreato) un pod (il che è possibile e comune), gli viene assegnato un indirizzo di rete differente
  - una soluzione a questo problema è offerta dai service – un'altra astrazione di Kubernetes, rappresentata da un'altra tipologia di risorsa



## Service

- Un **service** è una risorsa che definisce un punto di accesso costante a un gruppo di pod che offrono uno stesso servizio applicativo
  - ogni service ha un nome, un tipo (discusso più avanti), un indirizzo IP (chiamato cluster IP, un indirizzo IP nella service network) e una porta che non cambiano mai durante l'esistenza del service
    - non cambiano nemmeno se, nel corso del tempo, cambia il gruppo di pod (istanze di container) a cui il service si riferisce
    - attenzione: la locazione del service cambia se il service viene arrestato e ricreato
  - in particolare, è utile e comune creare dei service in corrispondenza ai deployment – per definire un punto di accesso unico alle repliche di pod specificate da un deployment



## Service

- Un **service** è una risorsa che definisce un punto di accesso costante a un gruppo di pod che offrono uno stesso servizio applicativo
  - inoltre, usando un service, un pod può comunicare con un altro pod dell'applicazione tramite il suo cluster IP – o, ancora più semplicemente, tramite il nome del service associato al pod (grazie al DNS del cluster)
    - ogni accesso fatto mediante l'indirizzo IP (o il nome) e la porta del service verrà inoltrata (tramite kube-proxy) a uno dei pod del service – in genere, il service opera anche da load balancer nei confronti dei suoi pod



## Service

- Ecco la specifica di un service (file **hello-service-clusterip.yaml**)
  - la specifica del deployment è rimasta invariata

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-deploy
... come prima ...
```

```
---
apiVersion: v1
kind: Service
metadata:
  name: hello-svc
spec:
  type: ClusterIP
  selector:
    app: hello
  ports:
    - port: 8080
      targetPort: 8080
```





## Service

- Nella specifica di un service
  - il campo **type** indica il tipo di service (i tipi di service sono discussi più avanti) – il default è **ClusterIP**
  - il selettore consente di selezionare (tramite etichette) i pod a cui è associato il service
  - inoltre, il campo **port** indica la porta su cui va esposto il service, mentre **targetPort** indica la porta di interesse esposta dal pod

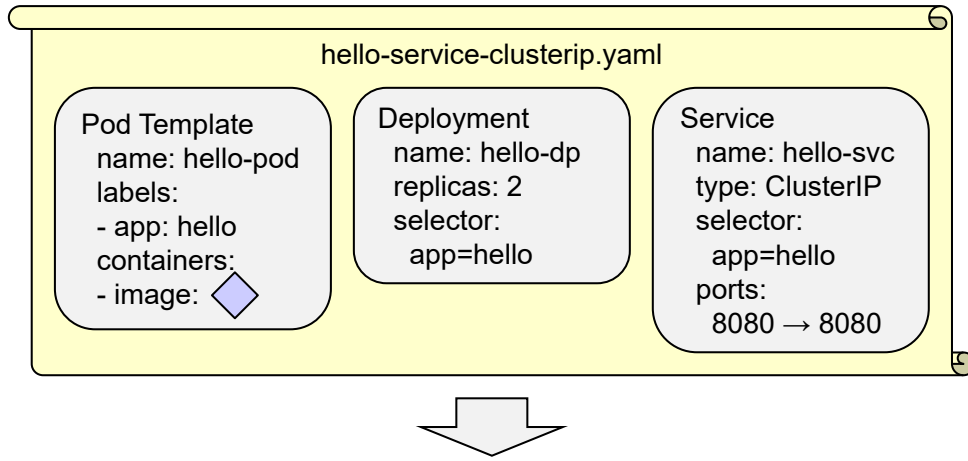


## Service

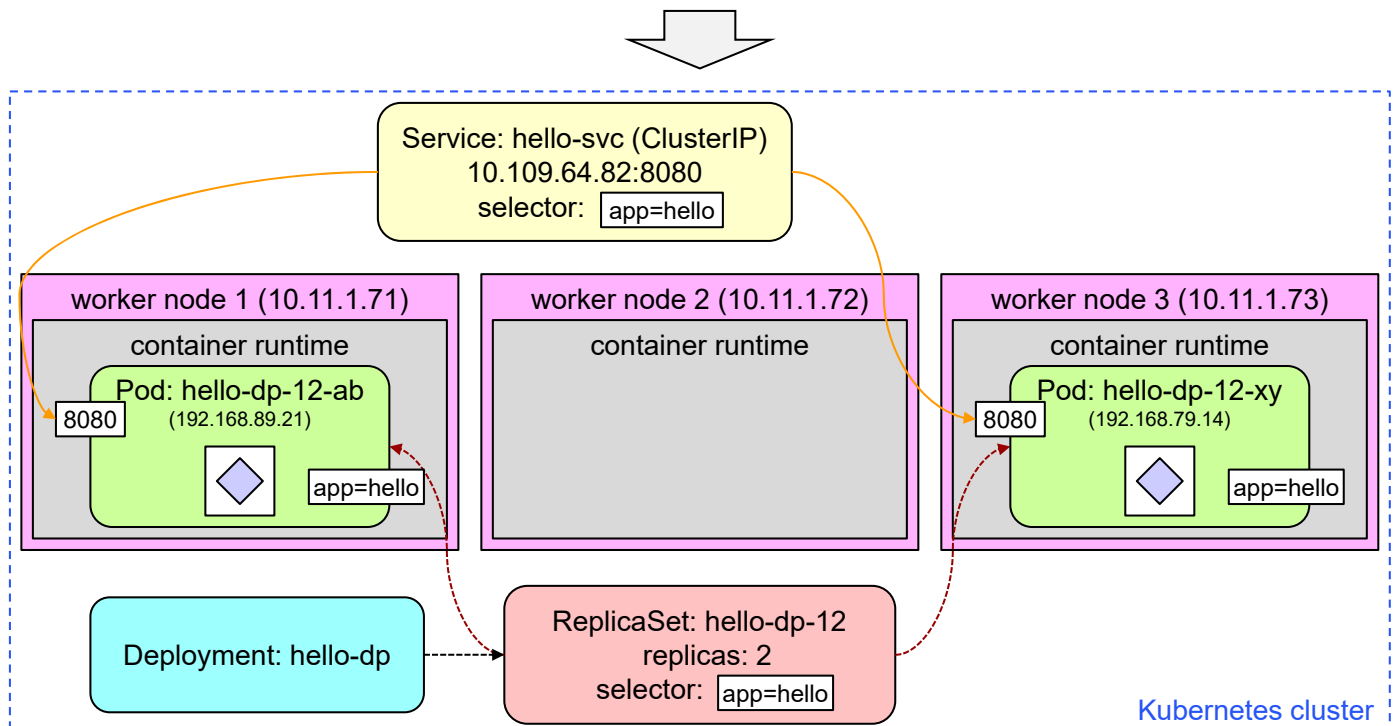
- Ecco alcuni comandi **kubectl** per la gestione dei service
  - **kubectl apply -f hello-service.yaml**
    - in questo caso, crea il deployment (con il suo replica set e i suoi pod) e il service **hello-svc**
  - **kubectl get services** oppure **kubectl get svc**
    - elenca i service del cluster
  - **kubectl describe svc/hello-svc**
    - fornisce informazioni sul service specificato – come il suo tipo (in questo caso, Cluster IP), il suo indirizzo IP, la sua porta, gli endpoint dei pod a cui è associato
    - in questo caso, è possibile accedere al service mediante il suo indirizzo IP (ma sempre solo dalle macchine interne del cluster)
  - **kubectl delete svc/hello-svc**
    - elimina il service specificato



# Deployment e Service



# Deployment e Service





## - Service (seconda parte)

- Consideriamo ancora il problema della comunicazione tra e con i pod
  - un secondo problema da affrontare è che gli indirizzi di rete assegnati ai pod appartengono alla pod network e quelli assegnati ai service appartengono alla service network
    - queste reti sono però entrambe reti private del cluster, e pertanto rimane il problema dell'accesso ai pod dai client esterni al cluster
  - i service offrono anche una soluzione a questo problema



## Service accessibili dai pod interni

- Kubernetes fornisce diversi tipi di service
  - un primo tipo (ClusterIP) sostiene direttamente solo la comunicazione tra i pod interni al cluster
    - *ClusterIP* – associa al service un indirizzo IP interno al cluster (*cluster IP*) – il service agisce da load balancer tra i pod associati al service
    - questo tipo di service è utile per semplificare la comunicazione interna tra pod
      - un'alternativa è usare un servizio di service discovery applicativo come Consul oppure il servizio di service discovery di Kubernetes (discusso più avanti)



## Service accessibili dai client esterni

- ❑ Kubernetes fornisce diversi tipi di service
  - altri tipi di service sono accessibili anche da client esterni al cluster (ad es., dalla VM kube-dev)
    - **NodePort** – estende ClusterIP, allocando anche una porta per il service sui nodi del cluster
      - in questo modo, ogni nodo del cluster è in grado di accettare richieste sulla porta associata al service e di inoltrarle al service (“ingress routing mesh”)
      - ha degli inconvenienti, ed è sconsigliato in produzione – ma è utile durante lo sviluppo
    - **LoadBalancer** – estende NodePort, allocando anche un load balancer esterno (se supportato dal cloud in cui viene eseguito il cluster Kubernetes) che effettua il routing verso il service
      - è adatto ad esporre un servizio all’esterno in produzione



## NodePort

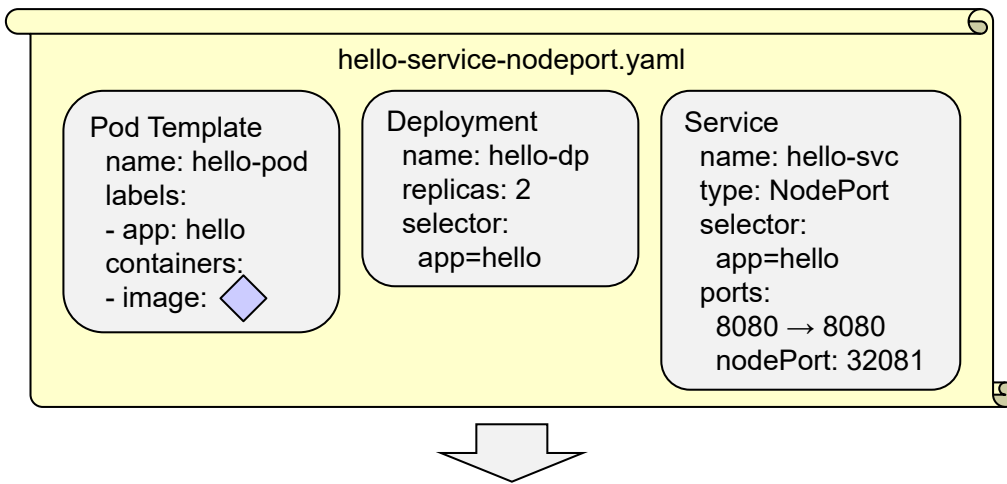
- ❑ Ecco la specifica di un service di tipo NodePort (file `hello-service-nodeport.yaml`)
  - la specifica del deployment è rimasta invariata

```
---
apiVersion: v1
kind: Service
metadata:
  name: hello-svc
spec:
  type: NodePort
  selector:
    app: hello
  ports:
  - port: 8080
    targetPort: 8080
    nodePort: 32081
```

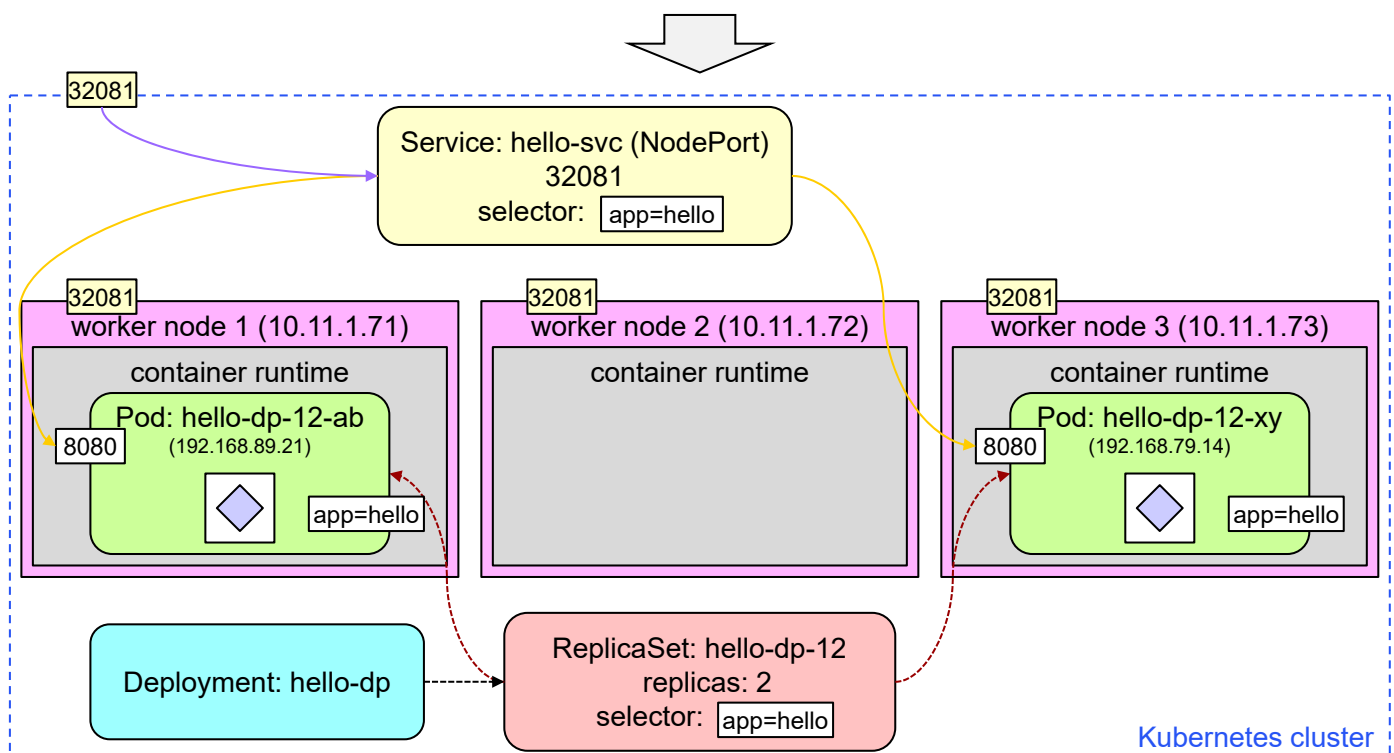
- il campo `nodePort` (opzionale) indica la porta esterna per il service – se assente, viene assegnata in modo casuale nell’intervallo (di solito) 30000-32767



# NodePort



# NodePort





- Ecco un client minimale basato su `curl`
  - il client accede al service sulla porta `32081`
  - supponiamo che il DNS usato dal client (oppure il suo file `/etc/hosts`) associ `kube-node` a qualunque nodo del cluster Kubernetes

```
curl kube-node:32081
```

- nel caso in cui la porta associata al service non sia nota, si può usare `kubectl` per determinarla

```
SERVICE_PORT=$(kubectl get svc/hello-svc \
-o go-template='{{(index .spec.ports 0).nodePort}}')
```

```
curl kube-node:${SERVICE_PORT}
```



## - Un altro esperimento

- Prima di andare avanti, facciamo questo esperimento
  - avviamo l'applicazione con `kubectl apply -f hello-service-nodeport.yaml`
  - eseguiamo il comando `kubectl get all -o wide`
    - a questo punto, dovremmo vedere le istanze di risorse mostrate nella figura precedente (anche se con nomi diversi)
      - tra di queste, il service `hello-svc`, di tipo NodePort, a cui è associato un indirizzo IP (cluster IP) e una porta
  - “uccidiamo” uno dei pod, come nell'esperimento precedente
  - eseguiamo di nuovo il comando `kubectl get all -o wide`
    - a questo punto, dovremmo vedere che non sono cambiati né l'indirizzo IP né la porta associati al service `hello-svc`
      - infatti, un service definisce un punto di accesso costante a un gruppo di pod per un servizio applicativo – mentre la composizione di questo gruppo può variare nel tempo



## - Ingress

- Un *ingress* è una risorsa per effettuare il routing di richieste HTTP e HTTPS indirizzate ai nodi del cluster verso service interni al cluster
  - si tratta di una modalità aggiuntiva per rendere uno o più service (di tipo NodePort o LoadBalancer) accessibili a client esterni al cluster
    - non solo con riferimento a una porta, ma anche a un hostname o a un path
  - l'uso degli ingress richiede l'installazione nel cluster di un ingress controller (come add-on)
    - nell'ambiente *kube-cluster* viene utilizzato NGINX Ingress Controller – è configurato per ascoltare richieste HTTP sulla porta 31080 di tutte le macchine (*kube-cluster*) ma anche sulla porta 80 dei nodi worker (*kube-node*)



## Ingress

- Ecco la specifica di un ingress (file *hello-ingress.yaml*)

```

---
apiVersion: v1
kind: Service
metadata:
  name: hello-svc
spec:
  type: NodePort
  selector:
    app: hello
  ports:
    - port: 8080
      targetPort: 8080
      # nodePort: 32081

```

```

---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: hello-ing
spec:
  ingressClassName: nginx
  rules:
    - host: hello.aswroma3.it
      http:
        paths:
          - pathType: Prefix
            path: /
            backend:
              service:
                name: hello-svc
                port:
                  number: 8080

```

- il service *hello-svc* viene esposto su <http://hello.aswroma3.it/> – sulla porta associata all'ingress controller, nel nostro caso 80



## Ingress

- Un ingress può specificare più regole di routing
  - ciascuna di queste regole può associare un service (campi `service.name` e `service.port.number`) a un URI composto dal nome di un host (campo `host`) e da un path (campo `path`)
    - nell'esempio, il service `hello-svc` (su `8080`) è esposto sull'URI `http://hello.aswroma3.it/` (su `80`)



## Ingress

- Un ingress può specificare più regole di routing
  - ciascuna di queste regole può associare un service (campi `service.name` e `service.port.number`) a un URI composto dal nome di un host (campo `host`) e da un path (campo `path`)
  - inoltre
    - il campo `host` è opzionale, e può essere specificato in modo esatto (`hello.aswroma3.it`) oppure con delle wildcard (`*.hello.com`)
    - il campo `path` può essere specificato in modo esatto (`pathType: Exact`) oppure come un prefisso (`pathType: Prefix`)
    - è possibile la riscrittura dell'URI della richiesta
    - è possibile effettuare il “canary” di alcune richieste su servizi differenti





# Ingress

- Per accedere effettivamente a un service mediante un ingress dall'esterno del cluster, è inoltre necessario che il nome dell'**host** specificato sia registrato nel DNS che serve il client esterno – e che il DNS associ questo nome ai nodi worker del cluster
  - per fare degli esperimenti
    - è possibile associare l'host all'indirizzo IP dei nodi worker del cluster nel file `/etc/hosts` del client, e poi usare

```
curl http://hello.aswroma3.it
```

- oppure si può usare `curl` con l'opzione `--connect-to` (redirige una richiesta da una coppia **host:porta** a un'altra)

```
curl http://hello.aswroma3.it \  
  --connect-to hello.aswroma3.it:80:kube-node:80
```

- oppure si può usare `curl` indicando un header di tipo **Host** (presenta la richiesta come rivolta ad un certo host)

```
curl kube-node --header "Host: hello.aswroma3.it"
```



# Ingress

hello-ingress.yaml

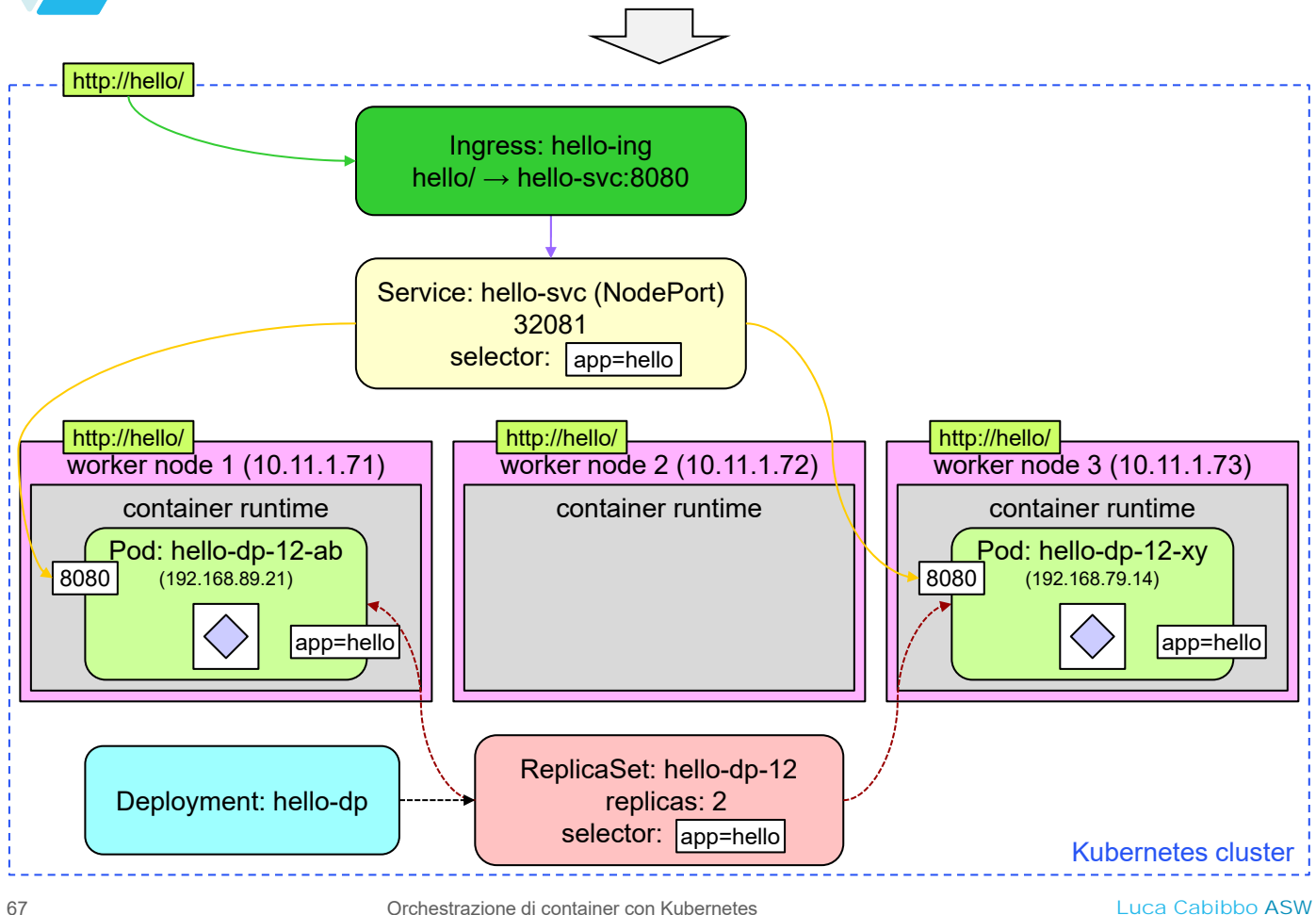
Pod Template  
name: hello-pod  
labels:  
- app: hello  
containers:  
- image:

Deployment  
name: hello-dp  
replicas: 2  
selector:  
app=hello

Service  
name: hello-svc  
type: NodePort  
selector:  
app=hello  
ports:  
8080 → 8080

Ingress  
name: hello-ing  
rules:  
- hello/ → hello-svc:8080





- ❑ Ecco un client minimale basato su **curl**
  - il client accede al service sull'host **hello.aswroma3.it**
  - supponiamo che il DNS usato dal client (oppure il suo file **/etc/hosts**) associ **hello** a qualunque nodo worker del cluster Kubernetes

```
curl hello.aswroma3.it
```

- nel caso in cui l'ingress controller sia associato a una porta diversa dalla porta **80** – ad es., la porta **31080**

```
curl hello.aswroma3.it:31080
```

- nel caso in cui l'host **hello** non sia registrato nel DNS, per fare degli esperimenti veloci nel cluster **kube-cluster**

```
curl hello.aswroma3.it --connect-to hello.aswroma3.it:80:kube-node:80
```

- oppure anche

```
curl kube-node --header "Host: hello.aswroma3.it"
```



## - Namespace

- Un **namespace** consente di associare una portata (scope) a ciascuna risorsa
  - utile quando il cluster viene utilizzato per eseguire più applicazioni, oppure è condiviso tra più utenti
    - ovvero, quando è possibile che ci siano sovrapposizioni di nomi tra le risorse di applicazioni o utenti diversi
    - i namespace consentono di separare le risorse in “gruppi” distinti – nonché di operare all’interno di un solo “gruppo” alla volta
  - in pratica, i namespace sono un modo per gestire più cluster virtuali in un unico cluster fisico
  - se per una risorsa non viene specificato nessun namespace, allora quella risorsa viene allocata nel namespace **default**
  - nota: i namespace di Kubernetes sono una cosa diversa dai namespace di Linux



## Namespace

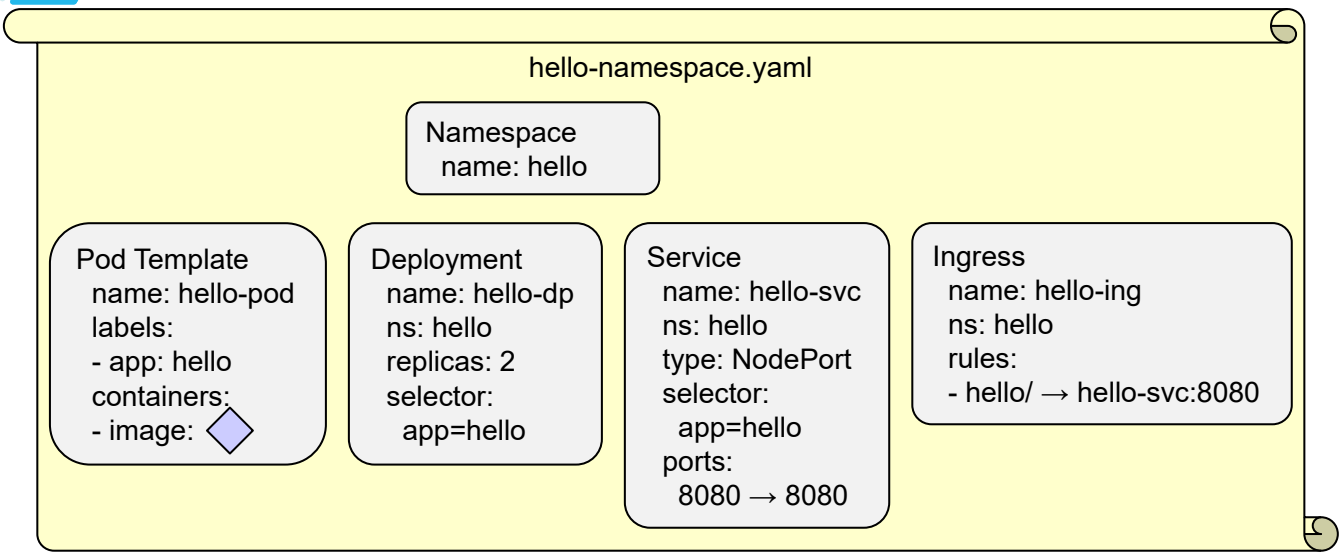
- Un primo modo di utilizzare i namespace (file **hello-namespace.yaml**)
  - scrivere la specifica di un namespace
  - nelle altre specifiche, indicare esplicitamente (tra i metadati) il namespace da usare per ciascuna risorsa

```
apiVersion: v1
kind: Namespace
metadata:
  name: hello
...
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-deploy
  namespace: hello
... come prima ...
```

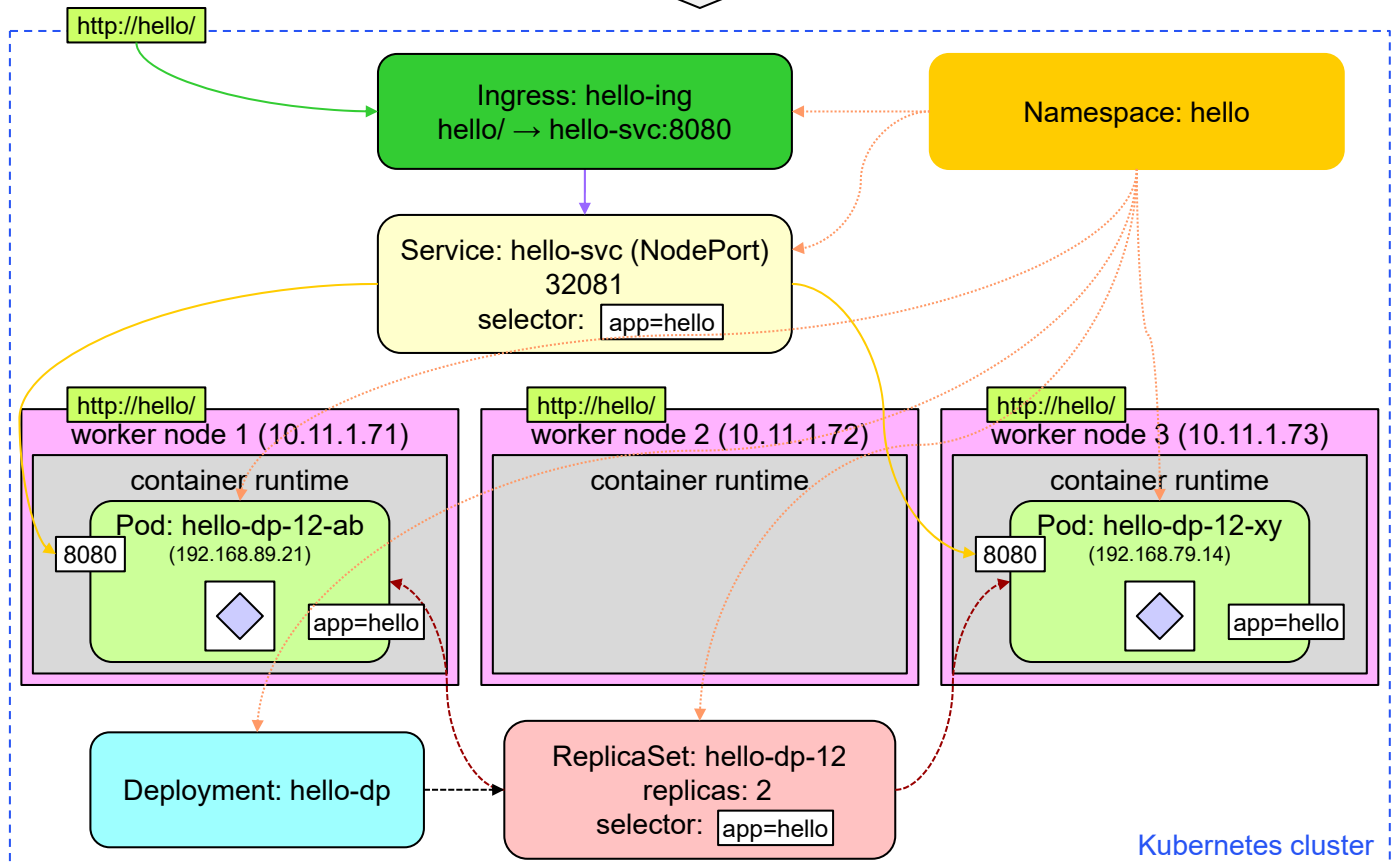
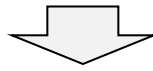
```
...
apiVersion: v1
kind: Service
metadata:
  name: hello-svc
  namespace: hello
... come prima ...
...
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: hello-ing
  namespace: hello
... come prima ...
```



# Namespace



# Namespace





## Namespace

- ❑ Ecco alcuni comandi **kubectl** per la gestione dei namespace
  - **kubectl get namespaces** oppure **kubectl get ns**
    - elenca tutti i namespace
  - **kubectl get pods --namespace hello** oppure **kubectl get pods -n hello**
    - elenca tutti i pod del namespace specificato
  - **kubectl get pods --all-namespaces**
    - elenca tutti i pod di tutti i namespace
  - **kubectl get pods**
    - elenca tutti i pod del namespace **default**



## Namespace

- ❑ Un secondo modo di utilizzare i namespace
  - usare l'ultima specifica mostrata in precedenza – senza la risorsa namespace né l'indicazione del namespace tra i metadati delle altre risorse
  - usare il seguente script per avviare l'applicazione – l'opzione **-n** consente di creare le risorse del file nel namespace specificato
    - **kubectl create namespace hello**
    - **kubectl apply -f hello-application.yaml -n hello**
  - usare il seguente script per arrestare l'applicazione
    - **kubectl delete -f hello-application.yaml -n hello**
    - **kubectl delete namespace hello**
  - questo modo di procedere consente di mantenere le specifiche delle risorse indipendenti dai namespace utilizzati per il loro rilascio



## - Probe



- Kubernetes (tramite kubelet) usa i *probe* (“sonda”) per verificare lo stato di salute dei pod

- i probe possono essere definiti nella specifica di un pod

...

```
template:
```

```
...
```

```
spec:
```

```
  containers:
```

```
    - image: aswroma3/hello-kube:latest
```

```
      livenessProbe:
```

```
        ...
```

```
        readinessProbe:
```

```
        ...
```

- ogni probe ha dei parametri propri
- il caso più semplice è un probe è basato su una richiesta HTTP GET su una porta e un path del pod
  - il pod “passa” (supera) il check se la risposta ha un codice 2xx o 3xx, e non lo passa con un codice di errore 4xx e 5xx



## Probe



- Ci sono diversi tipi di probe

- un *liveness probe* consente di verificare periodicamente lo stato di salute di un pod

- se il check fallisce, il pod viene ucciso e riavviato

```
livenessProbe:
```

```
  httpGet:
```

```
    path: /actuator/health
```

```
    port: 8080
```

```
  periodSeconds: 5
```

```
  failureThreshold: 3
```

```
  initialDelaySeconds: 120
```

- in questo caso, lo stato di salute dell’applicazione nel pod viene verificato ogni 5 secondi – e sono ammessi al più 3 fallimenti consecutivi – ma aspettando prima 2 minuti per l’avvio del pod
- è utile, ad es., se il pod può andare in loop o essere coinvolto in un deadlock, senza però andare in crash



## □ Ci sono diversi tipi di probe

- anche un **readiness probe** consente di verificare periodicamente lo stato di salute di un pod
  - tuttavia, se il check fallisce, Kubernetes si limita a non inoltrare richieste al pod – fino a quando il pod non tornerà a passerà il check

```
readinessProbe:
  httpGet:
    path: /actuator/health
    port: 8080
  periodSeconds: 10
```

- è utile, ad es., quando il tempo di avvio di un pod non è breve – in particolare, durante i rolling update



## □ Ci sono diversi tipi di probe

- uno **startup probe** è utile per i pod che richiedono un tempo alto di avvio ed inizializzazione

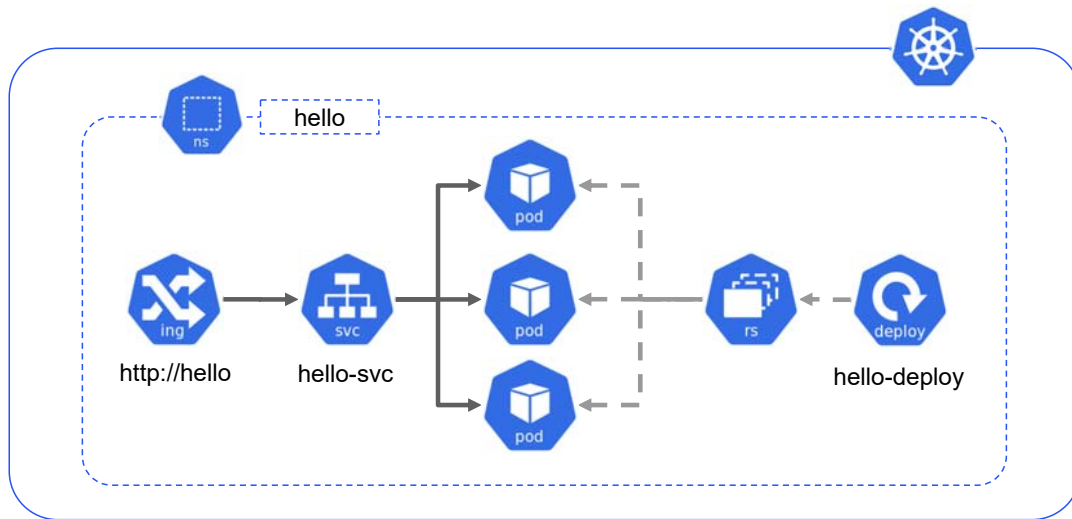
```
startupProbe:
  httpGet:
    path: /actuator/health
    port: 8080
  periodSeconds: 10
  failureThreshold: 30
```

- in questo caso, l'applicazione nel pod deve avere un tempo massimo di avvio di  $10 \cdot 30 = 300$  secondi (5 minuti)
  - altrimenti, il pod viene ucciso e riavviato
- questo probe viene usato solo durante l'avvio del pod – nel frattempo gli altri tipi di probe sono disabilitati
- utile per avere probe a granularità temporale differente – più lunga per lo startup, più breve per liveness e readiness



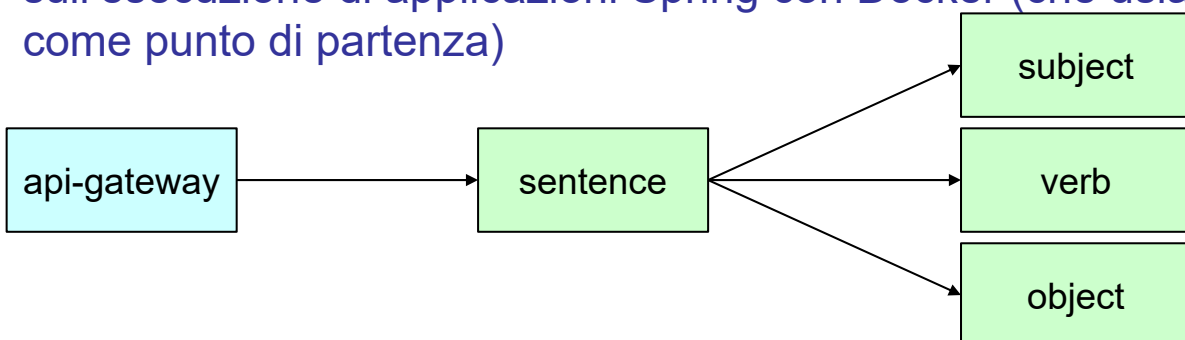
## - Architettura dell'applicazione hello

- Per riassumere, la seguente figura descrive l'architettura dell'applicazione **hello**, rappresentando le risorse Kubernetes utilizzate



## \* Orchestrazione con Kubernetes

- Discutiamo ora il rilascio su Kubernetes di un'applicazione multi-servizi e multi-container
  - come applicazione di esempio, consideriamo di nuovo l'applicazione **sentence** per generare frasi in modo casuale
  - questa applicazione è basata su un servizio principale **sentence** che utilizza degli ulteriori servizi per generare parole di tipo diverso (**subject**, **verb** e **object**) e un API gateway **api-gateway**
  - questa applicazione è stata inizialmente presentata nella dispensa su Spring Cloud – e poi ripresa nella dispensa sull'esecuzione di applicazioni Spring con Docker (che usiamo come punto di partenza)







## Un'applicazione per frasi casuali

- Per rilasciare l'applicazione **sentence** con Kubernetes, possiamo usare
  - per il servizio applicativo principale **sentence**
    - un'immagine di container **sentence-sentence-kube:latest** (**sentence-sentence-kube:2023-10**)
    - un deployment – per allocare più pod
    - un service **sentence** – di tipo ClusterIP
  - per gli ulteriori servizi applicativi **subject**, **verb** e **object**
    - un'immagine di container **sentence-word-kube:latest** (**sentence-word-kube:2023-10**), con tre profili **subject**, **verb** e **object**
    - un deployment per ciascun tipo di parola
    - un service per ciascun tipo di parola, di tipo ClusterIP

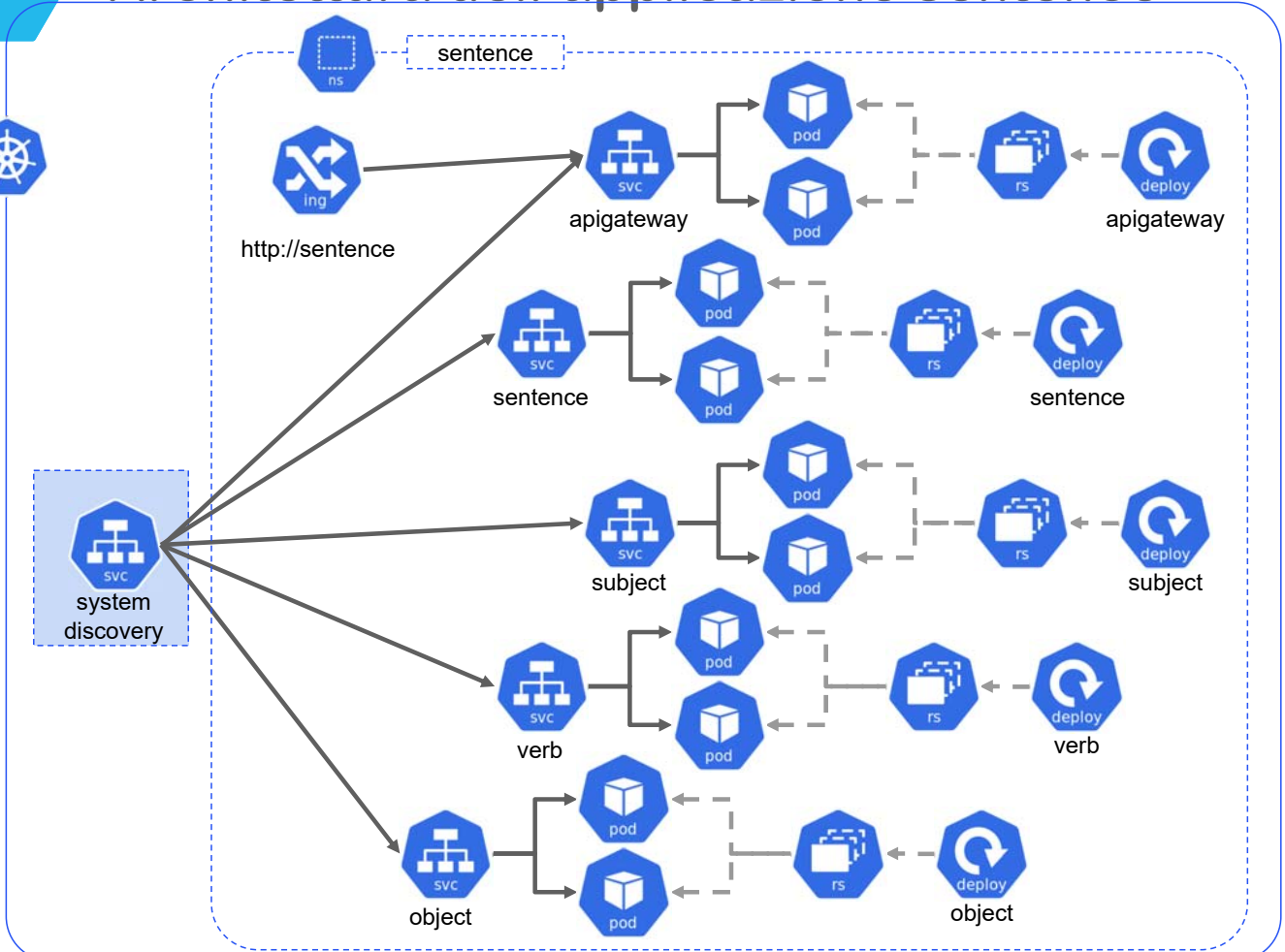


## Un'applicazione per frasi casuali

- Per rilasciare l'applicazione **sentence** con Kubernetes, possiamo usare
  - per l'API gateway **apigateway**
    - un'immagine di container **sentence-apigateway-kube:latest** (**sentence-apigateway-kube:2023-10**)
    - un deployment – per allocare più pod
    - un service **apigateway** di tipo NodePort – esposto mediante un ingress su <http://sentence.aswroma3.it/>, per effettuare il routing delle richieste dei client verso il servizio richiesto
  - inoltre, per la comunicazione tra i servizi utilizziamo il servizio di service discovery di Kubernetes
    - in alternativa, è possibile usare Consul – oppure semplicemente il DNS fornito da Kubernetes (ma l'implementazione richiesta è leggermente differente)



# Architettura dell'applicazione sentence



83

Orchestrazione di container con Kubernetes

Luca Cabibbo ASW



## Modifiche alle applicazioni

- ❑ Rispetto a quanto visto nella dispensa sull'esecuzione di applicazioni Spring con Docker, i nostri servizi applicativi vanno configurati come segue (il codice non va modificato)
  - la configurazione di tutte le applicazioni va modificata come segue
    - va rimossa la dipendenza starter per Consul (che non viene più usato)
    - al suo posto, va utilizzata la dipendenza **spring-cloud-starter-kubernetes-client-loadbalancer**
    - dalla configurazione (file **application.properties**) va rimossa la sezione per Consul (si può anche lasciare, perché comunque viene ignorata)
    - nel nostro esempio, non è invece necessaria nessuna configurazione per il servizio di service discovery di Kubernetes

84

Orchestrazione di container con Kubernetes

Luca Cabibbo ASW



## Modifiche alle applicazioni

- Rispetto a quanto visto nella dispensa sull'esecuzione di applicazioni Spring con Docker, i nostri servizi applicativi vanno configurati come segue (il codice non va modificato)
  - nel servizio **sentence**, i client REST possono continuare a fare riferimento alle diverse istanze per il servizio delle parole utilizzando gli URI **http://subject**, **http://verb** e **http://object** – così come si faceva usando Consul per la service discovery
    - se anziché il servizio di service discovery di Kubernetes si volesse invece basare la comunicazione tra servizi sul DNS di Kubernetes, si dovrebbero utilizzare gli URI **http://subject:8080**, **http://verb:8080** e **http://object:8080**



## Modifiche alle applicazioni

- Rispetto a quanto visto nella dispensa sull'esecuzione di applicazioni Spring con Docker, i nostri servizi applicativi vanno configurati come segue (il codice non va modificato)
  - nel servizio **apigateway**, nemmeno la configurazione dell'API gateway e delle sue rotte va modificata
    - se invece si volesse basare la comunicazione sul DNS di Kubernetes, andrebbero cambiati gli URI usati nelle diverse rotte – ad es., da **lb://sentence** a **http://sentence:8080**



# Dockerfile

- Per ciascuno dei servizi bisogna definire un **Dockerfile**
  - come esempio, questo è il **Dockerfile** per il servizio **sentence**
  - l'applicazione Spring per il servizio **sentence** viene associata a una porta nota (del container, non dell'host) – ad es., 8080

```
# Dockerfile per il servizio sentence
FROM eclipse-temurin:17-jdk
ADD build/libs/sentence.jar sentence.jar
EXPOSE 8080
ENTRYPOINT ["java", "-Xmx128m", "-Xms128m", "-jar", "sentence.jar"]
```

- i **Dockerfile** per gli altri servizi sono simili
- le immagini Docker di interesse per questa applicazione sono state create e salvate su Docker Hub
  - per farlo, si può utilizzare anche Docker Compose



# Specifica dell'applicazione

- Ecco il file **sentence-application.yaml** per l'applicazione **sentence**
  - pod template e deployment per le parole

```
---
apiVersion: apps/v1      template:
kind: Deployment         metadata:
metadata:                labels:
  name: subject          app: sentence
spec:                    service: subject
  replicas: 2            spec:
  selector:              containers:
    matchLabels:         - name: subject-container
      app: sentence      image: aswroma3/sentence-word-kube:latest
      service: subject   env:
                        - name: SPRING_PROFILES_ACTIVE
                          value: subject
                        ports:
                        - containerPort: 8080
```

- analogamente per **verb** e **object**



## Specifica dell'applicazione

- ❑ Ecco il file `sentence-application.yaml` per l'applicazione `sentence`
  - pod template e deployment per le frasi

```
---
apiVersion: apps/v1      template:
kind: Deployment        metadata:
metadata:               labels:
  name: sentence        app: sentence
spec:                  service: sentence
  replicas: 2           spec:
  selector:            containers:
    matchLabels:       - name: sentence-container
      app: sentence    image: aswroma3/sentence-sentence-kube:latest
      service: sentence ports:
                       - containerPort: 8080
```



## Specifica dell'applicazione

- ❑ Ecco il file `sentence-application.yaml` per l'applicazione `sentence`
  - pod template e deployment per l'API gateway

```
---
apiVersion: apps/v1      template:
kind: Deployment        metadata:
metadata:               labels:
  name: apigateway      app: sentence
spec:                  service: apigateway
  replicas: 2           spec:
  selector:            containers:
    matchLabels:       - name: apigateway-container
      app: sentence    image: aswroma3/sentence-apigateway-kube:latest
      service: apigateway ports:
                       - containerPort: 8080
```



## Specifica dell'applicazione

- Ecco il file `sentence-application.yaml` per l'applicazione `sentence`
  - service per le parole

```
---
apiVersion: v1
kind: Service
metadata:
  name: subject
spec:
  selector:
    app: sentence
    service: subject
  ports:
  - protocol: TCP
    port: 8080
    targetPort: 8080
```

- analogamente per `verb` e `object`



## Specifica dell'applicazione

- Ecco il file `sentence-application.yaml` per l'applicazione `sentence`
  - service per le frasi e per l'API gateway

```
---
apiVersion: v1
kind: Service
metadata:
  name: sentence
spec:
  selector:
    app: sentence
    service: sentence
  ports:
  - protocol: TCP
    port: 8080
    targetPort: 8080
```

```
---
apiVersion: v1
kind: Service
metadata:
  name: apigateway
spec:
  selector:
    app: sentence
    service: apigateway
  type: NodePort
  ports:
  - protocol: TCP
    port: 8080
    targetPort: 8080
#   nodePort: 32082
```



## Specifica dell'applicazione

- Ecco il file `sentence-application.yaml` per l'applicazione `sentence`
  - ingress per l'API gateway – è il punto di ingresso dell'applicazione

```
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: sentence
spec:
  ingressClassName: nginx
  rules:
  - host: sentence.aswroma3.it
    http:
      paths:
      - pathType: Prefix
        path: /
        backend:
          service:
            name: apigateway
            port:
              number: 8080
```



## Specifica dell'applicazione

- Ecco il file `sentence-application.yaml` per l'applicazione `sentence`
  - per rendere visibile il servizio di service discovery ai servizi dell'applicazione, utilizziamo anche le seguenti regole per la sicurezza (attenzione, sono molto permissive)

```
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: service-discovery-reader
rules:
- apiGroups: ["", "extensions", "apps"]
  resources: ["pods", "services", "endpoints"]
  verbs: ["get", "list", "watch"]
```

```
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: service-discovery-reader-binding
subjects:
# all service accounts in any namespace
- kind: Group
  name: system:serviceaccounts
  apiGroup: ""
- roleRef:
  kind: ClusterRole
  name: service-discovery-reader
  apiGroup: ""
```



# Avvio, accesso e arresto dell'applicazione

- ❑ Per avviare l'applicazione
  - `kubectl create namespace sentence`
  - `kubectl apply -f sentence-application.yaml -n sentence`
- ❑ Per accedere all'applicazione nel cluster `kube-cluster`
  - `curl sentence.aswroma3.it`  
`--connect-to sentence.aswroma3.it:80:kube-node:80`oppure
  - `curl kube-node --header "Host: sentence.aswroma3.it"`
- ❑ Per arrestare l'applicazione
  - `kubectl delete -f sentence-application.yaml -n sentence`
  - `kubectl delete namespace sentence`

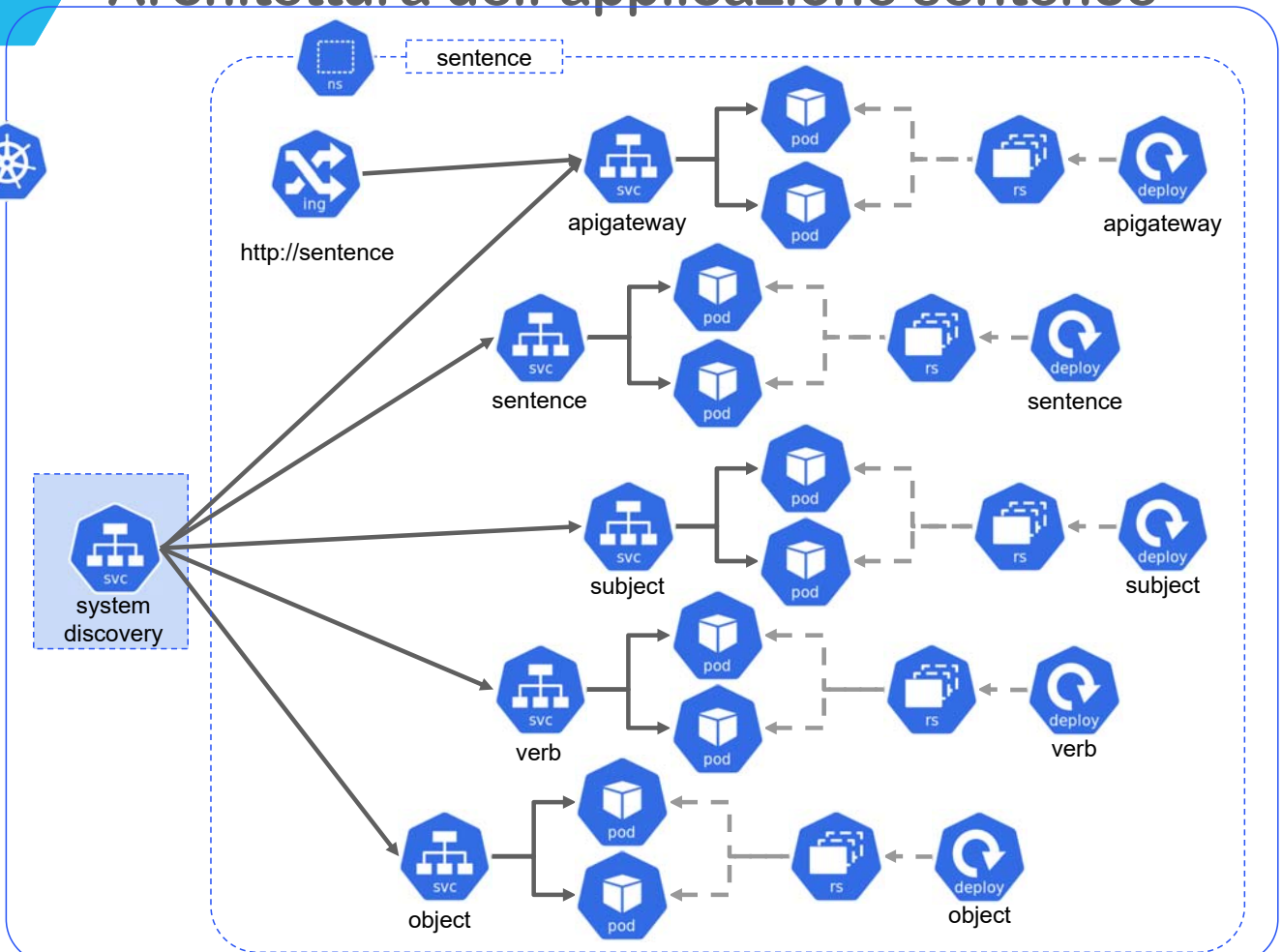
95

Orchestrazione di container con Kubernetes

Luca Cabibbo ASW



## Architettura dell'applicazione sentence



96

Orchestrazione di container con Kubernetes

Luca Cabibbo ASW





## - Rilascio su AWS



- ❑ Discutiamo ora (in modo semplificato) come rilasciare l'applicazione **sentence** a container su AWS
  - creiamo un cluster Kubernetes utilizzando il servizio completamente gestito Elastic Kubernetes Service (EKS)
  - installiamo e configuriamo nel cluster un ingress controller
  - EKS crea automaticamente una nuova istanza di Elastic Load Balancing (ELB) associata all'ingress controller e gli assegna anche un indirizzo IP esterno (visibile attraverso la dashboard di AWS) – ad es. [xyz.eu-west-1.elb.amazonaws.com](http://xyz.eu-west-1.elb.amazonaws.com)
  - usiamo una configurazione di deployment per l'applicazione simile a quella vista (con ulteriori configurazioni relative alla sicurezza) – ed esponiamo l'applicazione mediante un ingress, ad es., sull'host [sentence.aswroma3.it](http://sentence.aswroma3.it)
  - aggiungiamo sul servizio Route 53 di AWS un record DNS che risolve [sentence.aswroma3.it](http://sentence.aswroma3.it) con [xyz.eu-west-1.elb.amazonaws.com](http://xyz.eu-west-1.elb.amazonaws.com)

97

Orchestrazione di container con Kubernetes

Luca Cabibbo ASW



## - Un errore comune

- ❑ Attenzione ad evitare i seguenti errori comuni
  - dopo aver modificato (il codice sorgente di) una delle applicazioni, ricordarsi (sempre!) di fare quanto segue
    - effettuare (o ripetere) la build (Java) delle applicazioni
    - effettuare (o ripetere) la build (Docker) delle immagini Docker
    - effettuare (o ripetere) il push su Docker Hub delle immagini Docker (in questo caso è necessario!)
    - potrebbe anche essere necessario cancellare le immagini modificate dalla cache delle immagini dei container nei nodi worker del cluster (soprattutto se non è cambiato il numero di versione delle immagini che si vogliono utilizzare)

98

Orchestrazione di container con Kubernetes

Luca Cabibbo ASW



## \* Discussione

- L'orchestrazione di container è fondamentale per poter rilasciare **in produzione** le applicazioni multi-servizi e multi-container – in un singola macchina oppure in un cluster di macchine, fisiche o virtuali – on premises oppure nel cloud
  - l'orchestrazione di container sostiene infatti la disponibilità e la scalabilità delle applicazioni di questo tipo – e consente di sfruttare l'elasticità delle piattaforme virtualizzate e nel cloud
  - per questo, i container e gli orchestratori di container sono diventati delle tecnologie abilitanti per le applicazioni altamente scalabili – di solito realizzate come applicazioni a microservizi
  - nel contesto dei sistemi di orchestrazione di container, oggi Kubernetes è certamente tra quelli più diffusi



## Discussione

- Le funzionalità di orchestrazione offerte da Kubernetes che sono state esemplificate o discusse in questa dispensa
  - architettura a servizi/microservizi
  - uso di un linguaggio dichiarativo per la specifica delle applicazioni, basato su un insieme di astrazioni (risorse)
  - scalabilità e disponibilità dell'orchestratore
  - disponibilità delle applicazioni
  - comunicazione interna tra servizi
  - comunicazione con i client esterni



## Discussione

- Kubernetes offre anche delle ulteriori funzionalità di orchestrazione – che non sono state discusse in questa dispensa
  - scalabilità delle applicazioni – modifica del numero di repliche di ciascun tipo di pod – gestita manualmente oppure automaticamente, sulla base del carico della CPU o di altre metriche
  - aggiornamento delle applicazioni senza interruzione di servizio (basato sull'aggiornamento delle versioni delle immagini per i pod dei deployment)
    - Kubernetes supporta direttamente diverse strategie: rolling update, ri-creazione dei pod (con una breve interruzione di servizio), rollback – e ne supporta altre indirettamente
  - gestione di dati persistenti e volumi
  - gestione di dati di configurazione e segreti
  - gestione della sicurezza