KUBERNETES COURSE SUMMARY

Summary:

Youssef ElZarka

Instructor:

Mumshad Mannambeth

Section	Topics	No.
Core Concepts	• Cluster Architecture	○ 5
	• ETCD For Beginners	○ 7
	• ETCD In Kubernetes	○ 9
	• Kube api-server	○ 11
	• Kube Controller Manager	○ 13
	• Kube Scheduler	○ 15
	• Kubelet	○ 17
	• Kube Proxy	○ 19
	• Recap Pods	○ 21
	• Pods With YAML	○ 23
	• Recap ReplicaSets	○ 25
	• Deployments	○ 27
	• NameSpaces & Resource Quotes	○ 29
	• Services	○ 32
	• Service ClusterIP	○ 35
	• Imperative vs. declarative	○ 37

Section	Topics	No.
Scheduling	• Manual Scheduling	○ 41
	• Labels and Selectors	○ 43
	• Taints and Tolerations	○ 46
	• Node Selectors	○ 48
	• Node Affinity	○ 50
	• Taints & Tolerations vs. Node Affinity	○ 52
	• Resource Requirements and Limits	○ 54
	• DaemonSets	○ 56
	• Static Pods	○ 58
	• Multiple Schedulers	○ 62
Logging & Monitoring	• Monitor Cluster Components	○ 65
	• Check Status & Logs	○ 66
	• Managing Application Logs	○ 69

Section	Topics	No.
Application Lifecycle Management	• Rolling Updates & Rollbacks	○ 72
	• EntryPoint & CMD In Docker	○ 74
	• Commands & Args In Kubernetes	○ 78
	• Configure Env. Vars. In Apps	○ 80
	• Configuring ConfigMaps In Apps	○ 82
	• Configure Secrets In Apps	○ 87
	• ConfigMaps vs. Secrets	○ 92
	• Multi Containers PODs	○ 94
	• Init Containers	○ 95
Cluster Maintenance	• Drain & Cordon & Uncordon	○ 99
	• K8S Software Version	○ 101
	• Cluster Upgrade Process	○ 103
	• Backup & Restore Methods	○ 106

Section	Topics	No.
Security	• K8s Security Primitives	○ 110
	• Authentication In K8S	○ 112
	• TLS Introduction	○ 117
	• TLS Basics	○ 119
	• TLS In K8S	○ 122
	• TLS In K8S – Cert. Creation	○ 124
	• View Cert. With Troubleshooting	○ 128
	• Cert. API	○ 131
	• Kube Config	○ 134
	• API Groups	○ 137
	• Role Based Access Controls (RBAC)	○ 140
	• Image Security	○ 142
	• Security Context	○ 144
	• Cluster Role & Role Binding	○ 146
	• Network Policy	○ 148
Storage	• Intro to Docker Storage	○ 151
	• Storage In Docker	○ 152
	• Volume Driver Plugins In Docker	○ 155
	• Container Storage Interface (CSI)	○ 157
	• Volume In K8S	○ 159
	• Persistent Volumes (PV)	○ 161
	• (PV) Claims	○ 163

Section	Topics	No.
Networking	• Networking – Intro	○ 166
	• Prerequisite Switching Network	○ 167
	• Prerequisite DNS	○ 169
	• Prerequisite Network NameSpaces	○ 171
	• Prerequisite Docker Networking	○ 174
	• Prerequisite CNI	○ 176
	• Cluster Networking	○ 178
	• Pod Networking & CNI In K8S	○ 180
	• CNI Weave	○ 184
	• IP Address Management (IPAM)	○ 186
	• Service Networking	○ 189
	• ExternalName Service	○ 192
	• DNS In K8S	○ 194
	• CoreDNS In K8S	○ 196
	• Ingress	○ 198
Install K8S In The Hard Way	• Install K8S – Mindset	○ 206
	• Design K8S Cluster	○ 208
	• Choosing K8S Infra.	○ 211
	• Configure HA & Networking	○ 213
	• ETCD In HA	○ 215
	• Demo – Provision VMs	○ 215
	• Demo – Install Client Tools	○ 220
	• Demo – Secure Cluster	○ 222
	• Demo – Kube Config	○ 224
	• Demo – Data Encryption	○ 226
	• Demo – Install ETCD Cluster On Master Nodes	○ 228
	• Demo – Install Control Plane Components	○ 230
	• Demo – Install Control Plane Load Balancer	○ 232
	• Demo – Bootstrap Worker Node	○ 233
	• TLS Bootstrap Worker Node	○ 235
	• Demo – TLS Bootstrap Worker Node	○ 238
	• Configure Kubectl For Remote Access	○ 240
	• Demo – Deploy POD Networking Solution (Weave)	○ 241
	• Demo – Kune-api to Kubelet Connectivity	○ 243
	• Deploy CoreDNS	○ 245
Install K8S In The Kubeadm	• Intro	○ 247
	• Deploy With Kubeadm	○ 249
	• Demo – Deployment With Kubeadm	○ 251
	• Solution – Deploy K8S Cluster using Kubeadm	○ 254
End To End Tests	• Intro	○ 257
	• Run and Analyze	○ 259
	• Demo – Run End To End using Kubetest	○ 261
	• Smoke Test	○ 262

Section	Topics	No.
T-Shooting	• Intro • Application Failure • Control Plane Failure • Demo – Control Plane Failure • Worker Node Failure • Demo – Worker Node Failure	○ 265 ○ 266 ○ 270 ○ 271 ○ 273 ○ 274
Other Topics	• Advanced Kubectl With JSONPath Queries	○ 277

Section	Topics	No.
Important Topics In CKAD (Must Know it)	• Readiness Probes • Liveness Probes • Jobs • CronJobs • Storage Classes • StatefulSets • Creating StatefulSets With Options • Storage In StatefulSets • Headless Service	○ 282 ○ 284 ○ 286 ○ 289 ○ 291 ○ 293 ○ 294 ○ 296 ○ 299

Cluster Architecture

🎯 الفكرة العامة:

دي مقدمة عن للـ Kubernetes Architecture (معمارية الكلاستر)
الشرح بيستخدم تشبيه السفن (Ships) علشان يسهّل علينا فهم الأدوار المختلفة لمكونات الكلاستر.

🔹 أنواع السفن = أنواع النود

- عندنا نوعين من السفن:
  1. Cargo Ships : سفن الشحن (دول زي الـ Worker Nodes)
     هما اللي بيشيلوا الـ Containers ويشغلوا التطبيقات.
  2. Control Ships : سفن التحكم (دول زي الـ Master Node / Control Plane)
     هما اللي بينظموا العملية : مين يشيل إيه، فين يتخزن، إزاي يتابعوه.

🔹 الـ Master Node = Control Plane

الماستر نود مسؤول عن إدارة الكلاستر كله، جواه مكونات أساسية:

1. etcd 🗄️
   - قاعدة بيانات Key-Value
   - بيخزن فيها كل المعلومات عن الكلاستر: النود، البودز، الحالة الحالية.
2. 🚢 ➡️ 📦 Scheduler
   - زي الرافعة (Crane) اللي بتحط الكونتينر في السفينة المناسبة.
   - بيختار أي نود هتشغّل البود على حسب:
     - موارد متاحة (CPU/RAM)
     - Constraints (tolerations / affinity)
3. 🛠️ Controllers
   - زي المكاتب الإدارية في الميناء.
   - كل Controller له دور:
     - Node Controller : يتابع النود، لو وقعت يحاول يتصرف.
     - Replication Controller : يضمن إن العدد المطلوب من البودز شغال دايمًا.
4. 🌐 kube-apiserver
   - ده زي "مركز القيادة."
   - هو الـ واجهة بين أي حد (أوامرنا أو باقي المكونات).
   - كل حاجة (نودز، كنترولرز، يوزر) بتكلم الـ kube-apiserver

🔹 Container Runtime

- الكلاستر عشان يشغل Containers : محتاج Engine زي:
  - Docker (الأشهر)
  - أو Containerd / CRI-O
- بيتسطب على كل النود (ماستر و وركر).

🔹 Worker Node

كل وركر نود فيه حاجتين أساسيين:

1. 👮 kubelet (الكابتن)
   - Agent يشتغل على النود.
   - يسمع أوامر من الـ kube-apiserver
   - يشغّل/يوقف الكونتينرز، ويرجع Status
2. 🔀 kube-proxy
   - مسؤول عن الشبكة.
   - بيظبط الـ Rules علشان البودز والخدمات يكلموا بعض جوه الكلاستر.

📌 الخلاصة

- Master Node = Control Plane : جواه etcd, scheduler, controllers, kube-apiserver
- Worker Node = Cargo Ships : جواه kubelet, kube-proxy
- الهدف كله: إدارة وتشغيل التطبيقات (Pods) بشكل منظم، scalable، وموثوق.

ETCD For Beginners

🎯 الموضوع الأساسي :

المحاضرة دي مقدمة عن Etcd

Etcd = Key-Value Store : قاعدة بيانات مبسطة لتخزين بيانات في شكل مفتاح وقيمة

◆ ليه بنستخدم Etcd ؟

- مش زي قواعد البيانات العادية اللي بتخزن جداول (Rows & Columns)
- Etcd بيخزن حاجة بسيطة : مفتاح (Key) و قيمة (Value)
- مفيش مفاتيح مكررة (كل مفتاح لازم يكون فريد).
- بيتخزن فيه حاجات زي:
  - إعدادات (Configuration Data)
  - معلومات لازم تتقرأ بسرعة وتتكتب بسرعة.
- في Kubernetes مثلًا Etcd : بيخزن حالة الكلاستر بالكامل (النودز، البودز، السيستم)

◆ الفرق بينه وبين قواعد البيانات التقليدية

- Database تقليدية (Relational مثل MySQL)
  - فيها جداول.
  - كل صف يمثل Object (شخص مثلًا)
  - كل عمود = نوع معلومة (اسم، عنوان...).
- Etcd (Key-Value) :
  - بتقول:

    Key = user1

    Value = Ahmed
  - ولما تنادي على الـ Key : يديك الـ Value

◆ إزاي تبدأ مع Etcd

1. تنزل الـ Binary (الملف التنفيذي) من GitHub Releases
2. تفك الضغط وتشغل الـ etcd
3. بشكل افتراضي هيسمع على Port: 2379
4. ممكن أي Client يتوصل بيه ويحط أو يقرأ Data.

◆ الأداة الافتراضية etcdctl :

- دي أداة CLI (سطر أوامر) للتعامل مع Etcd
- أوامر أساسية:
  1. إضافة بيانات (Put) : etcdctl put key1 value1

     ده بيخزن key1=value1
  1. جلب بيانات (Get) : etcdctl get key1

     هيرجعلك value1 :
  1. عرض أوامر أخرى : etcdctl

🔹 **النقاط الجاية (في الكورس)**

- **بعد المقدمة دي، الكورس هيكلم عن:**
  - **Etcd في الشركات الكبيرة.**
  - **إزاي يشتغل كـ Distributed System (أكتر من نود)**
  - **بروتوكول Raft لتحقيق التوافق بين النودز**
  - **Best Practices : عدد النودز ، وتحقيق الـ High Availability**

📌 **الخلاصة:**

Etcd : قاعدة بيانات Key-Value بسيطة جدًا لكنها مهمة جدًا فيKubernetes ، لأنها المكان اللي بيتخزن فيه "كل حاجة" عن الكلاستر.

ETCD In Kubernetes

🎯 **الموضوع الأساسي:**

المحاضرة دي عن دور Etcd في Kubernetes

🔹 دور Etcd في Kubernetes

- Etcd هو الـ datastore الرئيسي اللي بيخزن كل حاجة عن الكلاستر:
  - Nodes
  - Pods
  - Configs
  - Secrets
  - Accounts
  - Roles & RoleBindings
  - ...إلخ
- أي أمر بتعمله بـ kubectl get : المعلومة جايالك من Etcd
- أي تغيير بتعمله (إضافة Node ، تشغيل Pod, ReplicaSet) مش بيُعتبر "تم" إلا لما يتسجل في Etcd

---

🔹 طرق نشر Etcd في الكلاستر

فيه طريقتين أساسيين:

1. من الصفر (From Scratch)
   - تنزل الـ Etcd binaries بنفسك.
   - تشغلها كـ Service على الـ Master Node
   - بتحتاج تحدد:
     - Certificates (للأمان TLS)
     - Cluster configs : لو فيه أكتر من Etcd instance
     - أهم حاجة دلوقتي:
       - --advertise-client-urls

       وده العنوان اللي Etcd بيسمع عليه .. عادة IP:2379
   - الـ kube-apiserver لازم يتوصل بالـ Etcd من خلال الـ URL ده.

2. باستخدام kubeadm :

- Kubeadm بيعمل deploy للـ Etcd كـ Pod في الـ namespace: kube-system
- تقدر تتعامل معاه من جوا الـ Pod باستخدام أداة etcdctl :
- مثلًا : لعرض كل الـ keys : etcdctl get / --prefix --keys-only
- كوبرنيتس بيخزن البيانات في شكل Directory Structure :
  - /registry/nodes
  - /registry/pods
  - /registry/replicasets
  - /registry/deployments
  - وهكذا...

◆ Etcd في High Availability (HA)

- في HA cluster : عندك أكتر من Master Node
- بالتالي : لازم يكون فيه أكتر من Etcd instance موزعين.
- علشان يشتغلوا مع بعض، لازم نعرف كل instance بالثاني من خلال:
  - الخيار --initial-cluster :
  - هنا بتكتب قائمة بالـ Etcd nodes اللي في الكلاستر.
- التفاصيل دي هتتشرح أكتر قدام في جزء الـ High Availability

◆ الخلاصة

- Etcd هو المخ الرئيسي لكلاستر Kubernetes
- كل الـ state (الحالة) بتتخزن فيه.
- تقدر تديره يدويًا أو تسيبه لـ kubeadm
- في HA setup : لازم كل Etcd nodes يعرفوا بعض علشان يحافظوا على التزامن (consistency)

Kupe api-server

🎯 **الموضوع الأساسي:**

المحاضرة دي عن kube-apiserver في Kubernetes

◆ مقدمة

- الـ kube-apiserver هو العقل المدبر لإدارة الكلاستر.
- أي أمر بتشغله بـ kubectl : بيتواصل مع الـ kube-apiserver
- دوره الأساسي:
  1. يعمل Authentication (يتأكد من هوية الطلب)
  2. يعمل Validation (يتأكد إن الطلب صحيح)
  3. يتواصل مع Etcd علشان يجيب أو يعدّل البيانات.

◆ مثال عملي – إنشاء Pod

1. تعمل kubectl apply -f pod.yaml
2. الخطوات:
   - الـ kube-apiserver يستقبل الطلب فيتحقق منه.
   - يسجل الـ Pod object في Etcd لكن بدون Node مخصص.
   - الـ Scheduler يراقب API server فيلاقي Pod من غير Node
   - يختار Node مناسب ويرجع المعلومة لـ API server
   - الـ API server يحدّث Etcd ويوصل المعلومة للـ Kubelet على الـ Node المناسب.
   - الـ Kubelet ينشئ الـ Pod ويشغل الـ Container runtime (زي Docker / containerd) علشان يسحب الـ Image ويشغله.
   - الـ Kubelet يبعث Status للـ API server وبعدين الـ API server يحدّث Etcd

➡️ نفس النمط ده بيتكرر مع أي عملية (إضافةNode ، حذفPod ، تعديل Deployment ... إلخ)

◆ مسؤوليات الـkube-apiserver

- Authenticate + Validate requests
- قراءة وتحديث البيانات في Etcd
- هو الـ component الوحيد اللي بيتعامل مباشرة مع Etcd
- باقي الـ components (Scheduler, Controller Manager, Kubelet) : بيتعاملوا معاه عن طريق API

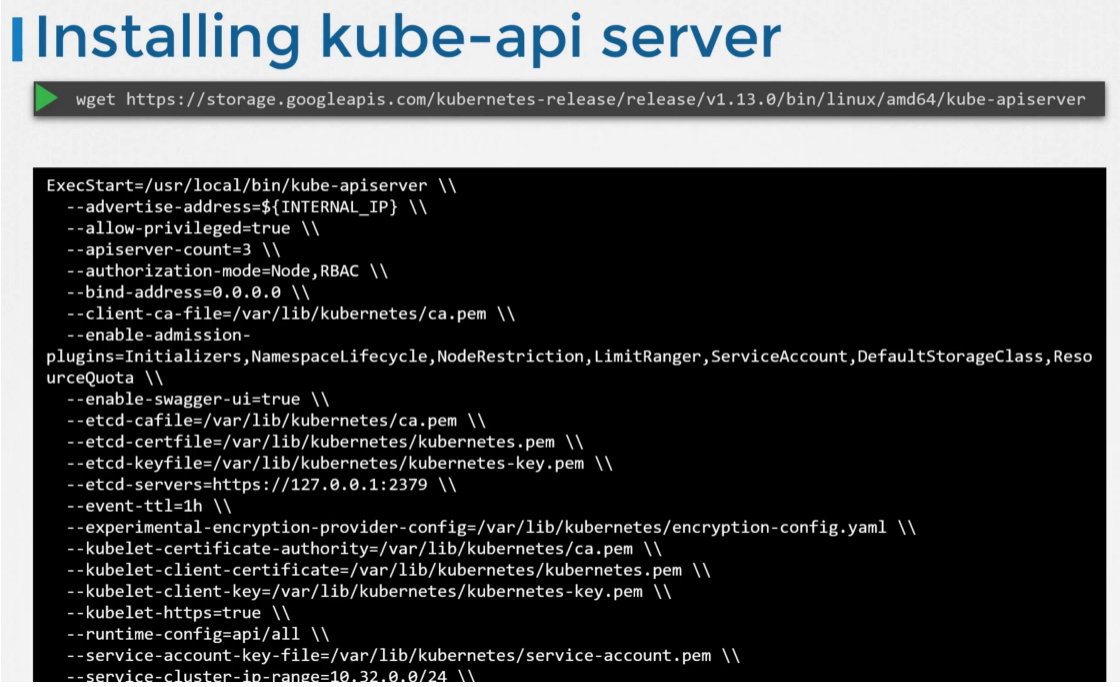
◆ تشغيل kube-apiserver

- لو استعملت kubeadm :
  - بيتنزل كـ Pod في kube-system namespace
  - ملف التعريف موجود في : /etc/kubernetes/manifests/kube-apiserver.yaml

فيه حاجات مينفعش تنساها في الكوبرنيتس عموما .. أهمها الـ path ده /etc/kubernetes/manifests/

- لو عملت setup من الصفر:
  - تنزل binary من Kubernetes release page
  - تشغله كـ Service على الـ Master node
  - ملف السيرفيس غالبًا موجود في:

/etc/systemd/system/kube-apiserver.service



🔹 أهم الـ Options (parameters)

- فيه عدد كبير من الـ options (معظمها certificates علشان الأمان)
- أهم واحد دلوقتي:
  - --etcd-servers : بتحدد فيه مكان Etcd server (IP:Port)
- لاحقًا في جزء الـ TLS/SSL certificates هنتعمق أكتر.

🔹 إزاي تشوف الـ options في Cluster قائم

- لو kubeadm :
  - بص في ملف الـ manifest

/etc/kubernetes/manifests/kube-apiserver.yaml

- لو setup من الصفر:
  - بص في ملف الـ systemd service

/etc/systemd/system/kube-apiserver.service

- أو شوف الـ process شغال بإيه : ps -aux | grep kube-apiserver

---

📌 الخلاصة

- kube-apiserver هو نقطة التحكم المركزية.
- كل المكونات في الكلاستر بتتعامل معاه علشان تدير حالة الكلاستر.
- هو الوحيد اللي بيتواصل مباشرة مع Etcd
- تشغيله بيكون كـ Pod (لو kubeadm) أو Service لو manual)

Kube Controller Manager

🎯 الموضوع الأساسي

المحاضرة دي عن Kube Controller Manager في Kubernetes

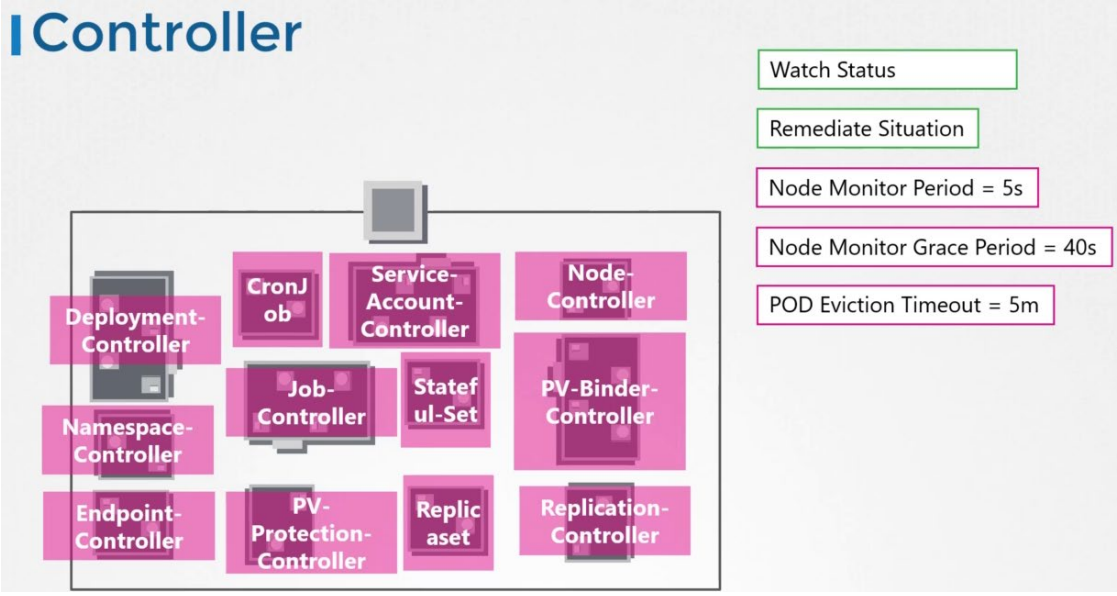
◆ الفكرة العامة

- الـ Controller Manager هو المسؤول عن تشغيل Controllers
- الـ Controller زي "مكتب/إدارة" في الكلاستر، كل واحد ليه مهمة معينة :
  - واحد بيراقب Nodes
  - واحد بيراقب Pods
  - واحد بيراقب ReplicaSets
  - إلخ...

◆ إيه هو الـ Controller ؟

- Process بيشتغل دايمًا، يراقب حالة جزء من الكلاستر.
- هدفه : يضمن إن الحالة الفعلية = الحالة المطلوبة (Desired State)
- أمثلة:

1. Node Controller
   - بيراقب صحة الـ Nodes
   - يتأكد إنها بترسل heartbeat كل 5 ثواني.
   - لو Node وقعت:
     - بعد 40 ثانية .. بيتعلم إنها Unreachable
     - يستنى 5 دقايق.
     - لو ما رجعتش .. يحذف الـ Pods اللي عليها وينقلها لغيرها.
2. Replication Controller
   - بيتأكد إن عدد الـ Pods المطلوب موجود.
   - لو Pod مات .. يخلق واحد جديد.



◆ ليه الـ Controllers مهمين؟

- أي feature في Kubernetes (Deployments, Services, PVs, Namespaces) : فيه Controller وراه هو اللي بينفّذ المنطق ده.
- تقدر تقول Controllers = دماغ الكلاستر.

◆ Kube Controller Manager

- هو Process واحد بيشغّل كل الـ Controllers مع بعض.
- يعني بدل ما يكون عندك Services كتير، كله متجمع في الـ kube-controller-manager

---

◆ التشغيل والإعداد

- لو Setup من الصفر:
  - تنزّل الـ binary من Kubernetes release page
  - تشغله كـ Service
- لو Setup باستخدام kubeadm :
  - بيتعمله Deploy كـ Pod جوه الـ namespace kube-system
  - ملف التعريف بتاعه في : /etc/kubernetes/manifests/kube-controller-manager.yaml

---

◆ أهم الـ Options

- Node monitor period : كل قد إيه يشيك على الـNodes.
- Grace period & Eviction timeout : الوقت اللي يستناه قبل ما ينقل الـPods.
- --controllers : تحدد أي Controllers يتفعّل (افتراضيًا : كلهم شغالين).

👉 لو Controller مش شغال، أول حاجة تبص عليها هي الإعدادات دي.

◆ إزاي تشوف الإعدادات؟

1. لو kubeadm :
   - بص في ملف الـ manifest في /etc/kubernetes/manifests/
2. لو manual setup :
   - الملف في /etc/systemd/system/kube-controller-manager.service
3. أو : ps -aux | grep kube-controller-manager
   علشان تشوف الـ options اللي شغالة فعلًا.

📌 الخلاصة

- الـ Controllers = العمليات اللي بتحافظ على استقرار الكلاستر.
- الـ Kube Controller Manager = " الحاوي" اللي بيشغّل كل الـ Controllers
- أمثلة مشهورة Node Controller + Replication Controller
- تشغيله بيكون Pod لو kubeadm أو Service لو manual

Kube Scheduler

🎯 موضوع المحاضرة

المحاضرة دي عن Kube Scheduler في Kubernetes

🔹 الدور الأساسي

- الـ Scheduler مسؤول عن تحديد أي Pod يروح على أي Node
- مهم : الـ Scheduler ما بيشغّلش الـ Pod فعليًا، هو بس بيقرر المكان.
  - اللي بيشغّل الـ Pod هو Kubelet (الكابتن على السفينة)

🔹 ليه محتاج Scheduler ؟

- لأن عندك كذا Node (سفن) وكذا Pod (حاويات)
- لازم تختار:
  - Pod يروح Node عندها الموارد الكافية (CPU, RAM)
  - Pod يتوزع على الـ Node المناسبة حسب الـ policy (مثلاً Node معينة مخصصة لداتابيز)
  - Pod يروح مكان يحقق الشروط (affinity, taints/tolerations)

🔹 إزاي Scheduler بيشتغل؟

1. 🔸 مرحلة الـ Filtering
   - بيشيل (يستبعد) أي Node مش مناسبة.
   - مثال Pod : محتاج CPUs4 ، والـ Node عندها CPU 2 بس .. يتم استبعادها.
2. 🔸 مرحلة الـ Ranking
   - بيرتب الـ Nodes الباقية ويحسب Score (0 → 10)
   - بيشوف مين هيكون " أفضل " Fit
   - مثال:
     - Node A بعد تشغيل الـ Pod هيفضل فيها CPU.2
     - Node B بعد التشغيل هيفضل فيها CPU.6
     - Node B تكسب لأنها عندها موارد أكتر متبقية.

👉 في الآخر Scheduler بيختار الـ Node "الأفضل" مش بس "الممكنة".

🔹 أمثلة على المعايير

- Resource requirements (CPU/RAM requests & limits)
- Node selectors : توجيه Pod ل Nodesمعينة
- Affinity/Anti-Affinity rules : مين يتحط مع مين أو مين يبعد عن مين
- Taints & tolerations : مين مسموحله يشتغل على Node عليها قيود

🔹 تثبيت وتشغيل Scheduler

- لو setup manual :
  - تنزّله من Kubernetes release page
  - تشغله كـ Service ومعاه ملف config
- لو setup بـ kubeadm :
  - بيتعمل Deploy كـ Pod في kube-system namespace
  - تعريفه موجود في : /etc/kubernetes/manifests/kube-scheduler.yaml
  -

🔹 إزاي تشوف إعدادات Scheduler ؟

1. لو kubeadm : الملف في /etc/kubernetes/manifests/
2. Manual setup : موجود كـ service في /etc/systemd/system/kube-scheduler.service
3. أو تشوف البروسيس شغال: ps -aux | grep kube-scheduler

📌 الخلاصة

- الـ Kube Scheduler : " مخطط" بيقرر مكان تشغيل الـ Pods
- خطواته Filtering → Ranking : : اختيار أفضل Node
- اللي ينفذ فعليًا Kubelet :
- تشغيله ممكن يكون كـ Pod (لو kubeadm أو Service لو manual)
- customizable بالكامل : ممكن تعمل Scheduler مخصص

Kubelet

🎯 موضوع المحاضرة

الموضوع عن Kubelet في Kubernetes

◆ تشبيه الكابتن (Captain Analogy)

- الـ Kubelet : الكابتن على السفينة (worker node)
- مسؤوليته :
  1. يعمل الـ ورق الرسمي عشان ينضم للسفينة الكبيرة الـ Cluster
  2. يبقى نقطة الاتصال الوحيدة مع الـ Master
  3. ينفذ أوامر تحميل/تشغيل الـ Pods والـ Containers (جاياله من الـ Scheduler)
  4. يبعت تقارير دورية عن حالة السفينة (الـ Node والحاويات (Pods/Containers) اللي عليها)

◆ دور Kubelet في الـ Cluster

1. أول ما يشتغل :
   - يعمل Register للـ Node في الـ Cluster
2. لما يجيله أمر يشغّل Pod :
   - يكلم الـ Container Runtime (زي Docker أو containerd)
   - يخليه يعمل Pull للصورة (image) ويشغّل الـ Container
3. بعد التشغيل:
   - يفضل يراقب حالة الـ Pod والـ Containers
   - يبعث التقارير دي بشكل دوري للـ kube-apiserver

◆ تثبيت Kubelet

- فيه فرق مهم عن باقي الـ Components :
  - لو بتستخدم kubeadm : هو ما بيثبتش الـ Kubelet تلقائي.
  - لازم تثبته يدويًا على كل Worker Node

خطواته :

1. تنزّل الـ kubelet من صفحة الإصدارات الرسمية.
2. تعمل Extract للملف.
3. تشغله كـ Service

◆ التحقق من تشغيله

- تقدر تشوف البروسيس والـ options المستخدمة بالأمر : ps -aux | grep kubelet

---

◆ مواضيع هنكملها لاحقًا

- إزاي نعمل Configuration للـ kubelet
- إزاي نولّد Certificates
- إزاي نعمل TLS bootstrap للـ kubelets (يعني يشتركوا في الـ Cluster بشكل آمن)

📌 الخلاصة

- الـ Kubelet هو الكابتن بتاع كل Node
- بينفذ أوامر الـ Master (عبر الـ API Server)
- بيشغّل Pods عن طريق الـ Container Runtime
- بيراقب ويبعث تقارير باستمرار.
- لازم يتثبّت يدويًا (خصوصًا في worker nodes)

Kube Proxy

🎯 موضوع المحاضرة

الموضوع عن Kube-Proxy ودوره في Kubernetes Networking

◆ التواصل بين الـ Pods

- في Kubernetes : كل Pod يقدر يوصل لأي Pod تاني.
- ده بيتحقق عن طريق:
  - Pod Network : شبكة افتراضية داخلية بتغطي كل الـ Nodes
  - كل Pod بيتوصل عليها، فيقدروا يتواصلوا مع بعض.

مثال:

- Web App على Node1
- Database على Node2
- الـ Web يقدر يكلم الـ DB باستخدام الـ Pod IP

◆ مشكلة الـ Pod IP

- مشكلة : الـ Pod IP مش ثابت (ممكن يتغير) في حالة لو Pod Restarted أو Rescheduled
- الحل: نستخدم Service

(Service) :

- بتدي اسم ثابت (زي db + Cluster IP ثابت)
- أي Pod يقدر يكلمها .. وهي توجه الترافيك للـ Pod الحقيقي (Backend)

◆ السؤال: الـ Service إزاي بتشتغل؟

- الـ Service مش حاجة ملموسة:
  - مش Pod
  - مش Container
  - مفيش Process شغالة باسمها.
- هي مجرد Object موجود في الذاكرة جوه الـ API Server

لكن ! الـ Service لازم تكون Accessible من أي مكان في الـ Cluster

إزاي؟ 🤔

◆ دور Kube-Proxy

- kube-proxy = Process بيشتغل على كل Node
- مسؤوليته:
  - يراقب أي Service جديدة.
  - يعمل Rules (قواعد) على كل Node عشان يوجّه الترافيك من الـ Service IP للـ Pod IP

مثال باستخدام IPTABLES :

- Service IP = 10.96.0.12
- Pod IP (DB) = 10.32.0.15
- kube-proxy يضيف Rule :
- أي ترافيك رايح لـ 10.96.0.12 .. يتحوّل لـ 10.32.0.15

🔹 تثبيت Kube-Proxy

- يدويًا:
  - تنزّله من صفحة الإصدارات الرسمية.
  - Extract وتشغّله كـ Service
- باستخدام kubeadm :
  - بيعمل Deploy للـ Kube-Proxy كـ DaemonSet
  - يعني Pod : واحد من kube-proxy بيشتغل تلقائيًا على كل Node

---

📌 الخلاصة

- Pods بتتواصل عبر Pod Network
- الـ Service بتدي اسم IP/ثابت.
- kube-proxy هو اللي بيخلّي الـ Service تبان وكأنها كيان حقيقي.
- بيستخدم Rules (زي IPTables) عشان يوجّه الترافيك للـ Pods المناسبة.
- في kubeadm : بيتنصّب كـ DaemonSet على كل Node

Recap Pods

🎯 موضوع المحاضرة

هانتكلم عن Kubernetes Pods :

- إيه هي؟
- إزاي بتشتغل؟
- إزاي نستخدمها في التشغيل والتوسيع (scaling) ؟

◆ قبل ما نبدأ

- نفترض:
  1. التطبيق مبني وموجود كـ Docker Image على Repository (زي Docker Hub)
  2. عندنا Cluster شغال (ممكن Single Node أو Multi-node)

◆ ما هو الـ Pod ؟

- Kubernetes ما بيشغلش Containers مباشرة.
- بدل كده بيغلفها في كائن اسمه Pod
- الـ Pod : أصغر وحدة تقدر تعملها في Kubernetes
- الـ Instance = Pod واحد للتطبيق.

◆ مثال بسيط

- عندنا Cluster فيه Node واحد.
- جوه الـ Node : Pod واحد فيه Container واحد بيشغل التطبيق.

◆ Scaling (توسيع التطبيق)

- لو زاد عدد المستخدمين:
  - ما نضيفش Containers جوه نفس الـ Pod
  - الصح : نعمل Pod جديدة لكل Instance إضافية.
- يعني:
  - Scale Up : إنشاء Pods جديدة.
  - Scale Down : حذف Pods موجودة.
- لو الـ Node الحالي مش كافي .. نضيف Node جديد في الـ Cluster وننشر Pods عليه.

◆ علاقة الـ Pod بالـ Containers

- غالبًا Pod ↔ Container (واحد لواحد)
- لكن ممكن يكون فيه أكتر من Container جوه Pod

ليه نعمل كده؟

- في حالة الـ Helper Container :
  - مثال : Container مساعد يعالج ملفات أو بيانات للمستخدم.
  - يتشاركوا:
    - نفس الـ Network Namespace و يتواصلوا عبر localhost
    - نفس الـ Storage

- نفس المصير (يتخلقوا سوا، يتوقفوا سوا).

📌 المهم : الـ Multi-container Pod نادر، والـ Use-case الأساسي بيكون Sidecar Pattern (زي Logging agent أو Helper)

🔹 مقارنة مع Docker مباشرة

- لو شغلنا Containers يدويًا بـ docker run :
  - لازم نربط الـ Network بنفسنا.
  - لازم نعمل Volumes يدويًا.
  - لازم نراقب Main Container ونوقف الـ Helper لو مات.
- في Kubernetes :
  - كل ده بيتم تلقائيًا عن طريق تعريف Pod واحد فيه أكتر من Container

---

🔹 إنشاء Pod عمليًا

- نستخدم أمر : kubectl run nginx --image=nginx
  - بيعمل : Pod جديد فيه Container بيشغل Nginx
- الصور بتنزل من Docker Hub (ممكن نخليها Private Repo لو عايزين)

🔹 أوامر أساسية مع Pods

- عرض الـ Pods : kubectl get pods
  - يبين الـ Status (ContainerCreating → Running)
- ملاحظة:
  - في المرحلة دي الـ Pod مش Accessible من بره الـ Cluster
  - عشان نوصلها من بره، لازم نستخدم Service (هنتكلم عنها في جزء الشبكات لاحقًا)

📌 الخلاصة

- Pod : أصغر وحدة في Kubernetes
- Pod : بيحتوي عادةً على Container واحد.
- للتوسيع: نضيف Pods جديدة بدل ما نضيف Containers جوه نفس Pod
- Multi-container Pod : نادر .. بيتعمل لحالات الـ Helper/Sidecar
- Kubernetes بيسهل الربط، التخزين، والمصير المشترك للـ Containers
- نتحكم في الـ Pods بأوامر زي kubectl run و kubectl get pods

Pods with Yaml

🎬 مقدمة

هنتكلم عن إنشاء Pod باستخدام ملف YAML
في المحاضرة اللي فاتت اتعلمنا فكرة عامة عن ملفات الـ YAML ، والنهاردة هنركز إزاي نكتبها مخصوص علشان نعرّف Objects في Kubernetes زي الـ Pods, Services, Deployments إلخ.

🗂️ البنية الأساسية لملف YAML فيKubernetes

أي ملف تعريف (Definition File) في Kubernetes لازم يحتوي على 4 أقسام رئيسية:

1. apiVersion : بتحدد إصدار الـ API اللي بنستخدمه.
   - مثلاً : v1 للـ Pods
   - ممكن تبقى قيم تانية زي : apps/v1 للـ Deployments ، أو batch/v1 للـ Jobs
2. kind : نوع الـ Object اللي هتنشئه.
   - هنا : Pod
   - ممكن تبقى Service, Deployment, ReplicaSet وهكذا.
3. metadata : بيانات تعريفية عن الـ Object
   - زي : (name (اسم الـ Pod) و labels (لتصنيف الـ Pod أو تجميعه)
   - مهم جدًا : الـ labels بتساعدك تفرق بين مئات الـ Pods (زي front-end, backend, database)
4. spec : أهم جزء، بيحدد تفاصيل تشغيل الـObject.
   - في حالة الـ Pod ، هنا بنكتب معلومات الـ containers اللي جوه الـ Pod
   - مثلاً: اسم الكونتينر + صورة (Image) من Docker Hub زي nginx

---

🖊️ مثال عملي على Pod YAML

ده مثال مبسط:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
    type: front-end
spec:
  containers:
    - name: nginx-container
      image: nginx
```

📌 شرح الكود:

- apiVersion: v1 : الإصدار المناسب للـ Pods
- kind: Pod : نوع الـ Object هو Pod
- metadata.name : اسم الـ Pod هيكون my-app-pod
- metadata.labels : ليه label باسم app: my-app
- spec.containers : هنا عرفنا أول Container :
  - اسمه nginx-container
  - صورته nginx من Docker Hub

```
kubectl create -f pod-definition.yml
```

⚙️ أوامر kubectl للتعامل مع الـPod

Kind	Version
POD	v1
Service	v1
ReplicaSet	apps/v1
Deployment	apps/v1

- إنشاء الـ Pod : kubectl create -f pod-definition.yaml
- عرض كل الـ Pods : kubectl get pods
- عرض تفاصيل Pod معينة: kubectl describe pod my-app-pod

هتشوف معلومات زي:

- تاريخ الإنشاء.
- الـ Labels
- الـ Containers اللي جوه.
- الأحداث (Events) المرتبطة بالـ Pod

📝 ملحوظة مهمة عن الـ Metadata

- في metadata : لازم تكتب بس القيم اللي Kubernetes بيتوقعها (name, labels, annotations...)
- لكن جوه labels : عندك حرية تعمل أي key-value pairs زي ما تحب.

🎯 الخلاصة

- أي ملف YAML في Kubernetes بيبدأ بـ apiVersion, kind, metadata, spec
- الـ spec بيختلف حسب نوع الـ Object (Pod, Service, Deployment...)
- لإنشاء Pod :
  1. تكتب YAML زي المثال.
  2. تعمل kubectl create -f file.yaml
  3. تتأكد منه باستخدام kubectl get/describe

Recap ReplicaSets

مقدمة

- ❖ هانتكلم عن Kubernetes Controllers، وخصوصًا Replication Controller وبعدها ReplicaSet
- ❖ الـ Controllers هم "المخ" اللي بيدير الموارد (Objects) في الكلاستر : بيراقبوا الحالة (desired state vs current state) ويتصرفوا تلقائيًا لو في مشكلة.

ما هو الـReplication Controller ؟

- الفكرة : يضمن إن عدد محدد من الـ Pods يكون دايمًا شغال.
- مثال : لو عندك Pod واحد والتطبيق وقع .. المستخدمين مش هيلاقوا الخدمة.
  الحل: تخلي فيه أكثر من Pod
- حتى لو عندك Pod واحد، الـ Replication Controller هيعيد إنشاءه أوتوماتيك لو وقع.
- أهم ميزة:
  - High Availability (التطبيق دايمًا شغال)
  - Load Balancing (توزيع الحمل بين Pods متعددة)

ملاحظة Replication Controller : كان قديم، وحاليًا اتبدل بـ ReplicaSet (الأحدث والموصى بيه)

ما هو الـ ReplicaSet ؟

- نفس الهدف زي الـ Replication Controller : يضمن وجود عدد محدد من الـ Pods
- الفرق الأساسي:
  - ReplicaSet بيلزم وجود Selector علشان يحدد أي Pods يدخلوا تحت إدارته.
  - كمان بيقدر يـ manage Pods معمولة قبل كده لو الـ Labels بتاعتها متطابقة.

إنشاء Replication Controller (RC)

الخطوات:

1. تكتب ملف YAML
2. زي أي Object : لازم يحتوي على:
   - metadata : الاسم + labels
   - spec : أهم جزء:
     - replicas : عدد الـ Pods المطلوب.
     - Template : تعريف الـ Pod نفسه (metadata + spec)

لاااحظ ان دا الـ Template اللي ممكن احطه في الـ Pod الواحدة

مثال:

ده هيعمل RC مسؤول عن تشغيل 3 Pods بنفس الشكل.

```yaml
# rc-definition.yml
apiVersion: v1
kind: ReplicationController
metadata:
  name: myapp-rc
  labels:
    app: myapp
    type: front-end
spec:
  template:
    metadata:
      name: myapp-pod
      labels:
        app: myapp
        type: front-end
    spec:
      containers:
      - name: nginx-container
        image: nginx
  replicas: 3
```

```yaml
# pod-definition.yml
apiVersion: v1
kind: Pod
```

```
> kubectl create -f rc-definition.yml
replicationcontroller "myapp-rc" created

> kubectl get replicationcontroller
NAME        DESIRED   CURRENT   READY   AGE
myapp-rc    3         3         3       19s

> kubectl get pods
NAME            READY   STATUS    RESTARTS   AGE
myapp-rc-4lvk9  1/1     Running   0          20s
myapp-rc-mc2mf  1/1     Running   0          20s
myapp-rc-px9pz  1/1     Running   0          20s
```

🛠️ إنشاء ReplicaSet (RS)

الاختلافات:

- **apiVersion : بدل v1 لازم نكتب apps/v1**
- **kind: ReplicaSet**
- **لازم نضيف selector يحدد أي Pods تتبع الـ ReplicaSet**

مثال:

📌 **هنا الـ ReplicaSet هيشوف أي Pod عنده label: app=myapp ويعتبره ضمنه.**

replicaset-definition.yml

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: myapp-replicaset
  labels:
      app: myapp
      type: front-end
spec:
  template:
    metadata:
      name: myapp-pod
      labels:
        app: myapp
        type: front-end
    spec:
      containers:
      - name: nginx-container
        image: nginx
  replicas: 3
  selector:
      matchLabels:
        type: front-end
```

pod-definition.yml

```yaml
apiVersion: v1
kind: Pod
```

```
> kubectl create -f replicaset-definition.yml
replicaset "myapp-replicaset" deleted
> kubectl get replicaset
NAME            DESIRED   CURRENT   READY   AGE
myapp-replicaset   3         3         3      19s
```

⚙️ أوامر kubectl المهمة

- **إنشاء RC أو RS : kubectl create -f file.yaml**
- **عرض الـ ReplicaSets : kubectl get rs**
- **عرض الـ ReplicationControllers : kubectl get rc**
- **عرض الـ Pods : kubectl get pods**
- **حذف ReplicaSet : kubectl delete rs myapp-rs**
- **تعديل ReplicaSet (Replace) : kubectl replace -f file.yaml**
- **توسيع ReplicaSet (Scale) لعدد معين من الـ Pods : kubectl scale rs myapp-rs --replicas=6**

📌 الخلاصة

- **ReplicationController قديم بس لسه بيفيد في فهم الأساسيات.**
- **ReplicaSet هو البديل الحديث والموصى به.**
- **بيستخدم selectors + labels علشان يربط الـ Pods**
- **تقدر تعمل Scale يدوي أو أوتوماتيك (HPA هنتكلم عنه لاحقًا)**

Deployments

🎬 مقدمة

هانتكلم عن Deployments في Kubernetes
وقبل ما يدخل في التفاصيل، بيقول تعالوا ننسى شوية الـ Pods و ReplicaSets ونفكر كأننا في Production Environment :

- عندنا تطبيق (مثلاً Web Server محتاج يشتغل بنسخ متعددة)
- عايزين نعمل تحديثات تدريجية (Rolling Updates) بدل ما نوقف كل حاجة مرة واحدة.
- عايزين لو حصل خطأ نقدر نعمل Rollback
- عايزين نقدر نعمل كذا تغيير (Scaling + Update + Resource changes) ونطبقهم مع بعض بعد ما نجهز Pause/Resume

🎯 كل ده بيتحقق عن طريق Deployments

🧩 ترتيب الـ Objects في Kubernetes

- Pod : بيشغل Container واحد أو أكتر.
- ReplicaSet : بيضمن وجود عدد معين من الـPods.
- Deployment : بيقعد فوق ReplicaSet (layer أعلى) (ويوفر مزايا إضافية updates, rollback, pause)

📌 Deployment هو المستوى العملي الحقيقي اللي بيستخدم في الإنتاج.

⚡ إمكانيات Deployment

1. Rolling Updates : تحديث نسخة الـ App تدريجيًا بدون ما المستخدم يحس.
2. Rollback : ترجع آخر تحديث حصل لو سبب مشكلة.
3. Pause/Resume : توقف التحديثات شوية، تعمل كذا تعديل، وبعدين تكمل rollout مرة واحدة.
4. Scaling : تزوّد أو تقلل عدد الـ Pods بسهولة.

🛠️ إنشاء Deployment

زي أي Object في Kubernetes ، بنحتاج ملف YAML
بيحتوي على:

- apiVersion: apps/v1
- kind: Deployment
- metadata : الاسم + labels
- Spec :
  - Replicas : عدد النسخ المطلوبة
  - Selector : بيحدد الـ Pods بالـ labels
  - Template : تعريف الـ Pod (metadata + containers)

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
  labels:
    app: myapp
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - name: nginx-container
        image: nginx:latest
```

📌 خطوات التشغيل

1. إنشاء Deployment : kubectl create -f deployment.yaml
2. عرض الـ Deployments : kubectl get deployments
3. عرض الـ ReplicaSets اللي اتعملت تلقائيًا: kubectl get rs
4. عرض الـ Pods اللي اتعملت جوة الـ ReplicaSet : kubectl get pods
5. عرض كل الموارد مع بعض (Deployment + ReplicaSet + Pods) : kubectl get all

---

📌 الفرق بين ReplicaSet و Deployment

- ReplicaSet : يضمن عدد الـ Pods
- Deployment : بيستخدم ReplicaSet لكن بيضيف :
  - Rolling updates
  - Rollbacks
  - Pause/Resume

يعني الـ Deployment = ReplicaSet + مزايا إضافية لإدارة الـ Production بسهولة.

✅ الخلاصة:

- الـ Deploymentهو الأوبجكت الأساسي لإدارة التطبيقات في Kubernetes
- هو اللي هتستخدمه في الشغل الحقيقي عشان يديك المرونة في التحديثات والـ scaling

NameSpaces

🎬 المقدمة

هانشرح فكرة الـ Namespaces فيKubernetes ، وبيبدأ بمثال حياتي عشان يقرب المعنى:

- فيه اتنين اسمهم "Mark"
- جوه بيتهم (House) أهاليهم بينادوا عليهم بالاسم الأول بس.
- لكن لو حد من بيت تاني عايز ينادي عليهم .. لازم يستخدم الاسم الكامل (Mark Smith أو Mark Williams)
- كل بيت له قوانينه وموارده الخاصة.

📌 في Kubernetes

- البيوت : Namespaces
- كل Namespace ليه موارده + قوانينه الخاصة.

🧩 الـ Namespaces الافتراضية في Kubernetes وأنواعها :

لما الـ Cluster يتعمل Setup أول مرة:

1. default : اللي بنشتغل فيه طول الوقت من غير ما نحس.
2. kube-system : بيتحط فيه الـ Pods والخدمات الداخلية الخاصة بالـ Cluster (زي DNS, Networking)
3. kube-public : أي موارد مفتوحة متاحة للجميع) ممكن أي User يشوفها.)

⚡ ليه نستخدم Namespaces ؟

- لو شغال بتجربة بسيطة أو Cluster صغير .. ممكن تفضل في default ومش هتحتاج غيره.
- لكن في Production :
  - ممكن تستخدم Namespace للـ dev وأخرى للـ prod
  - ده يمنع إن حد يعدل بالخطأ في بيئة الإنتاج وهو فاكر نفسه في dev
- كل Namespace ممكن:
  - يبقى له Policies خاصة تحدد مين يقدر يعمل إيه.
  - يتحدد له Resource Quota (مثلاً dev عنده 10 Pods ، 5 CPU ، 10 GB Memory)

🌐 التواصل بين الموارد

- جوه Namespace واحد : الـ Pods/Services بيكلموا بعض بالاسم المباشر.
  - مثال webapp Pod : يقدر يوصل للـ db-service بمجرد كتابة اسمه.
- بين Namespaces مختلفة: لازم تكتب الاسم بالكامل (FQDN) :
- <service-name>.<namespace>.svc.cluster.local

مثال:

db-service.dev.svc.cluster.local

```
mysql.connect("db-service.dev.svc.cluster.local")
```

🛠️ أوامر التعامل مع Namespaces

1. عرض الـ Pods في الـ default : kubectl get pods
2. عرض Pods في Namespace تاني (مثلاً kube-system): kubectl get pods --namespace=kube-system
3. إنشاء Pod جوه Namespace معين :
   - إمّا بالـ option : kubectl create -f pod.yaml --namespace=dev
   - أو تضيف الـ Namespace جوه ملف الـ YAML تحت metadata :

```
metadata:
  name: mypod
  namespace: dev
```

🏗️ إنشاء Namespace جديد

- بملف YAML :

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  namespace: dev
  labels:
    app: myapp
    type: front-end
spec:
  containers:
  - name: nginx-container
    image: nginx
```

علشان أعمل create للـ Pod في namespace معينة

- وتشغّل:

kubectl create -f dev-namespace.yaml

- أو مباشر:
- kubectl create namespace dev

🔄 التبديل بين الـ Namespaces

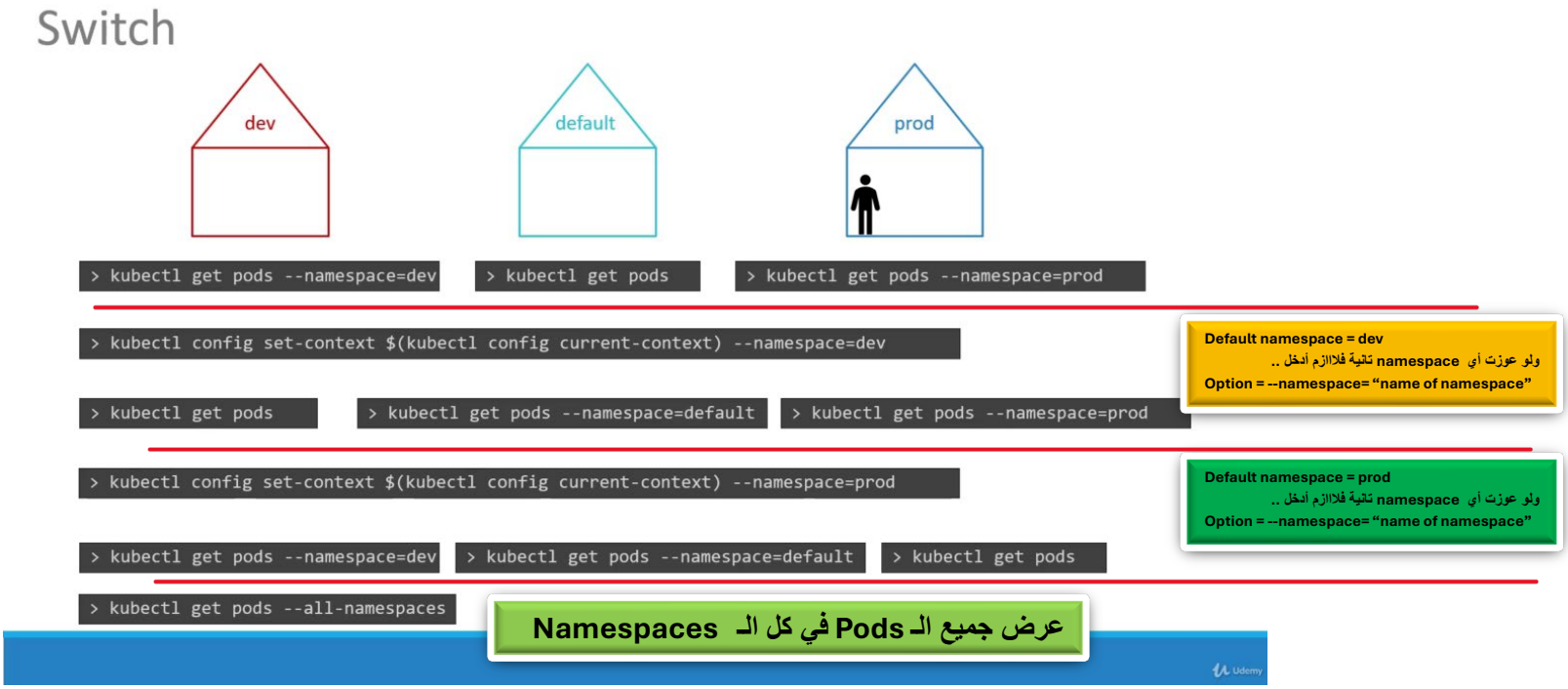
- بشكل افتراضي إحنا جوه default
- لو عايز تفضل دايمًا جوه dev من غير ما تكتب --namespace:

kubectl config set-context --current --namespace=dev

- بعد كده أي kubectl get pods هيعرض من dev على طول.

```
> kubectl create -f pod-definition.yml
pod/myapp-pod created
```

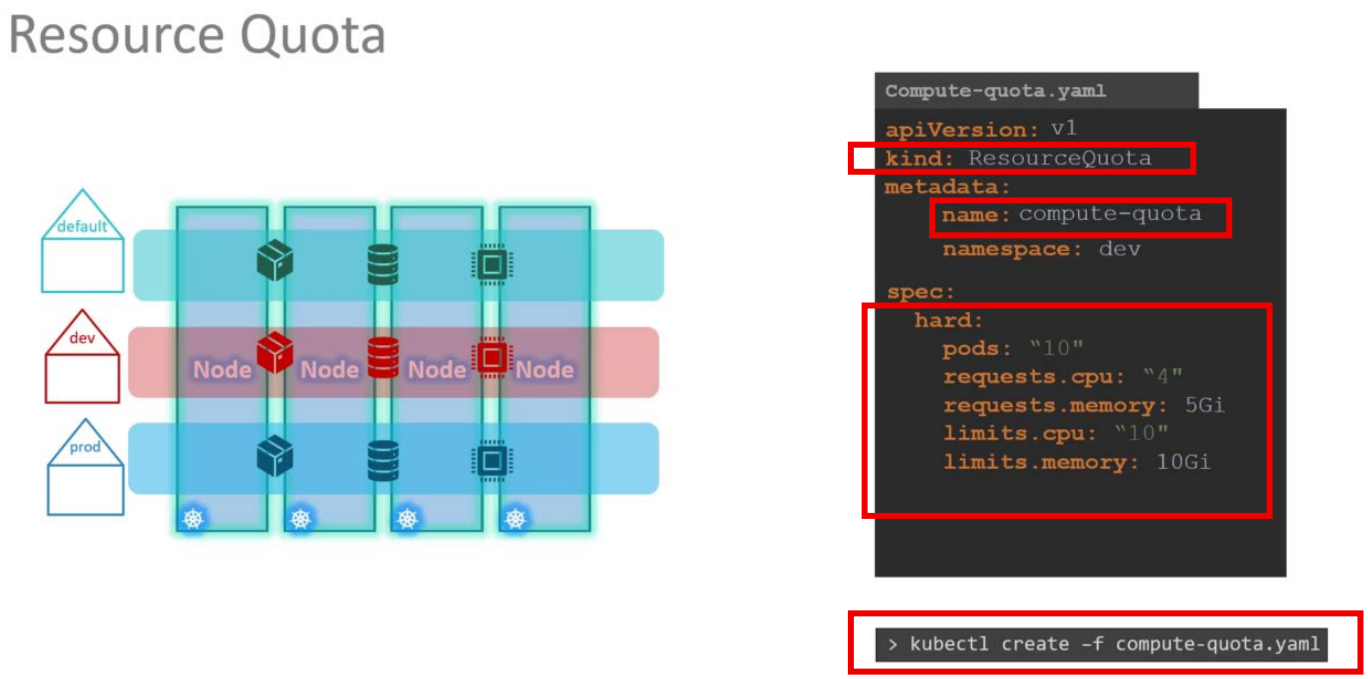
```
> kubectl create -f pod-definition.yml
pod/myapp-pod created
```



Resource Quotas 📌

لو عايز تحدد موارد لكل Namespace :

- تعمل ملف ResourceQuota :



- الهدف: تضمن إن كل Namespace مايستهلكش أكتر من حصته.

الخلاصة ✅

- Namespace : بيت يفصل الموارد + القوانين.
- مهم جدًا في Production لعزل البيئات (dev, test, prod)
- يسهّل إدارة الصلاحيات + الكوتا.
- الموارد تتكلم ببساطة بالاسم جوه نفس Namespace ، وبـ FQDN بين الـ Namespaces

Services

🎬 المقدمة

- Kubernetes Services هدفها إنها تعمل تواصل بين المكونات المختلفة جوه التطبيق أو مع العالم الخارجي.
- بتوصل الـ Pods ببعض أو مع مستخدمين خارجيين أو مع قواعد بيانات خارجية.
- وبالتالي هي اللي بتخلي الـ Microservices يشتغلوا مع بعض بشكل Loose Coupling (يعني مش مربوطين ببعض بشكل جامد)

📦 مثال للتوضيح

- عندنا 3 مجموعات من الـ Pods :
  1. Frontend : يخدم المستخدمين.
  2. Backend : يعالج البيانات.
  3. Database Connector : يوصل بقاعدة بيانات خارجية.
- Services هي اللي بتخلي:
  - الـ Frontend يكلّم المستخدمين.
  - الـ Frontend يتواصل مع Backend
  - الـ Backend يوصل للـ Database

🌍 مشكلة الوصول من الخارج

- Node عنده IP: 192.168.1.2
- Laptop عنده IP: 192.168.1.10
- Pod عنده IP داخلي : 10.244.0.2

🔴 وبالتالي هتلاقي ان الـ Laptop ما يقدرش يوصل مباشرًا للـ Pod لأنه في شبكة مختلفة.

الحل :

- ينفع من جوه الـ Node (بـ SSH) يعني نفتح المتصفح أو نعمل curl http://10.244.0.2 .. فيشتغل.
- لكن الهدف إني أوصل من اللابتوب مباشرةً بدون SSH

🛠️ هنا ييجي دور Service

- Service : Object زي Pod, ReplicaSet, Deployment
- واحدة من وظائفها: تسمع على Port في الـ Node وتنقل الترافيك للـ Pod
- النوع ده اسمه NodePort Service

🏷️ أنواع الـ Services

1. NodePort : يفتح Port على الـ Node ويحوّل الطلب للـ Pod
2. ClusterIP : يعمل IP داخلي (Virtual) يسهّل التواصل بين الـ Pods (مثلاً Frontend → Backend)
3. LoadBalancer : يطلب Load Balancer من Cloud Provider لتوزيع الترافيك بين عدة Pods

---

🔎 NodePort Service بالتفصيل

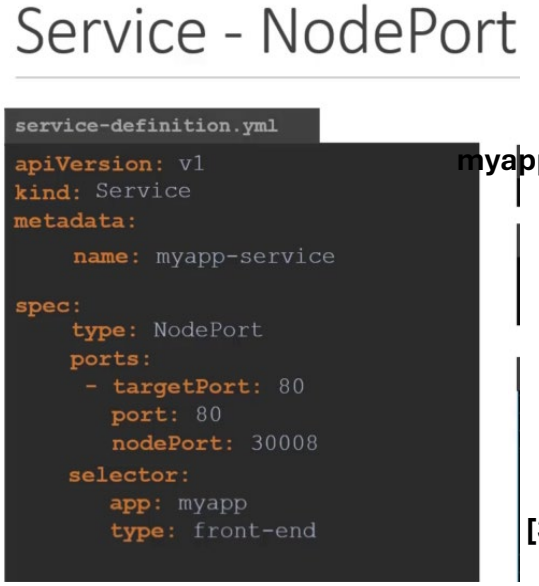
- فيه 3 Ports :
  1. targetPort = Port جوه الـ Pod (مثلاً 80 للـ Webserver)
  2. port = Port جوه الـ Service نفسه (دايمًا بيشاور على targetPort)
  3. nodePort = Port على الـ Node (بين 30000 → 32767)

مثال:

وبالتالي هقدر أفتح: http://192.168.1.2:30008 واوصل للـ Web App

---

📝 إنشاء Service (YAML)



❖ ملاحظات :

- Selector :
  - app: myapp يربط الـ Service بالـ Pods اللي عندهم label app = myapp
- ports :
  - port: 80 = دا الـ Port اللي علي الـ Service
  - targetPort: 80 = دا الـ Port اللي جوه الـ Pod
  - nodePort: 30008 = دا الـ Port اللي على الـ Node

📌 ملاحظات:

- port إجباري.
- لو ما كتبتش targetPort : بيتعامل إنه نفس قيمة port
- لو ما كتبتش nodePort : بيختار واحد عشوائي في المدى.[30000–32767]
- ports عبارة عن Array : ممكن تعمل كذا Mapping في نفس الـ Service

```
> kubectl create -f service-definition.yml
service "myapp-service" created
```

```
> kubectl get services
NAME            TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
kubernetes      ClusterIP   10.96.0.1        <none>        443/TCP        16d
myapp-service   NodePort    10.106.127.123   <none>        80:30008/TCP   5m
```

```
> curl http://192.168.1.2:30008
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
```

🏷️ ربط الـ Service بالـ Pods

- الخدمة لازم تعرف تكلّم أنهي Pods
- بنستخدم Labels + Selectors :
  - الـ Pods عندها Label: app=myapp
  - نحط نفس الـ Selector في الـ Service
- النتيجة : الـ Service تلقائيًا يلاقي كل الـ Pods اللي عندهم نفس الـ Label ويوصلهم .

⚖️ Load Balancing

- لو عندي أكتر من Pod بنفس الـLabel:
  - Service بيشاور عليهم كلهم.
  - بيستخدم خوارزمية Random لتوزيع الترافيك بينهم.
- بالتالي Service = Built-in Load Balancer

🌐 توزيع على أكتر من Node

- لو Pods متوزعة على كذا Node :
  - Service بيعمل نفس الـ NodePort على كل Nodes (مثلاً 30008)
  - أي IP من أي Node + نفس Port (يوصل للتطبيق)

✅ الخلاصة

- Service : جسر بين Pods وبين بعض أو مع الخارج.
- أهم الأنواع:
  - ClusterIP : داخل الكلاستر
  - NodePort : فتح Port من Node للخارج
  - LoadBalancer : في Cloud
- بتستخدم Labels + Selectors عشان تحدد أنهي Pods يستقبلوا الترافيك.
- فيها Load Balancing أوتوماتيك + Updates ديناميكية (لو Pods زادت أو قلت)

Service ClusterIP

🎬 المقدمة

- عندك تطبيق Web كامل (Full Stack) متقسم إلى طبقات (Layers) :
  - Frontend (الـ Web UI)
  - Backend (Business Logic)
  - Redis (Key-Value Store)
  - MySQL (Database)

كل طبقة عبارة عن Pods كتيرة.

مثال : عندك 3 Pods Backend كل واحد له IP مختلف.

❌ المشكلة

1. كل Pod له IPمتغير (مش ثابت). لو Pod وقع وجاء بداله Pod جديد : IP هيتغير.
2. الـ Frontend محتاج يوصل للـ Backend : بس أي Pod يختار؟ مين يقرر؟
3. كمان Backend محتاج يوصل للـ Redis والـ Database : برضه نفس المشكلة.

✅ الحل Kubernetes Service

- Service بيعمل:
  - يجمع مجموعة Pods تحت عنوان واحد (Single Access Point)
  - يديهم اسم + ClusterIP ثابت.
  - يعمل Load Balancing (يوزع الترافيك بين الـ Pods بالـ Random)

مثال:

- Service اسمه backend بيربط 3 Pods Backend
- أي Pod تاني (زي الـ Frontend يستخدم اسم backend : Kubernetes يحوّل الطلب لواحد من Pods Backend تلقائيًا)

🛠️ فوائد ClusterIP

- بيخلي كل Layer في التطبيق مستقلة عن التانية.
- تقدر تعمل Scaling (تزود/تنقصPods) بدون ما تكسر الاتصال.
- الـ Pods تتخاطب بالـ Service Name بدل الـ Pod IP

📌 ClusterIP = النوع الافتراضي للـ Services في Kubernetes

📝 تعريف ClusterIP Service (YAML)

service-definition.yml

```yaml
apiVersion: v1
kind: Service
metadata:
    name: back-end

spec:
    type: ClusterIP
    ports:
     - targetPort: 80
       port: 80

    selector:
       app: myapp
       type: back-end
```

- ملاحظات :
  - Spec :
    - default type: ClusterIP (حتى لو ما كتبتها)
  - selector :
    - app: backend (بيربط الـ Pods اللي عندها label app=backend)
  - ports :
    - port: 80 - Port على الـService
    - targetPort: 80 Port جوه الـPods

ملاحظات:

- port = المنفذ اللي الـ Service بيسمع عليه.
- targetPort = المنفذ اللي جوه الـ Pod (مكان التطبيق الحقيقي)
- selector = يحدد أنهي Pods هتكون جزء من الخدمة.

🔎 أوامر عملية

- إنشاء الـ Service : kubectl create -f backend-service.yaml
- مشاهدة كل الـ Services : kubectl get services
- النتيجة هتوريك حاجة زي:

```
NAME        TYPE        CLUSTER-IP     PORT(S)   AGE
backend     ClusterIP   10.96.0.15     80/TCP    1m
```

- أي Pod تاني يقدر يوصل للـ Backend عن طريق:
  - ClusterIP: 10.96.0.15
  - أو بالاسم http://backend

✅ الخلاصة

- ClusterIP = خدمة داخلية تربط مجموعة Pods بعنوان واحد.
- بتحل مشكلتين:
  1. IPsالمتغيرة للـ Pods
  2. Load Balancing بين مجموعة Pods
- بتستخدم جوه الكلاستر فقط (مش متاحة للمستخدم الخارجي).
- الافتراضي لأي Service في Kubernetes

Imperative vs. declarative

ليه imperative vs. declarative؟

- Declarative (الطريقة المُفضلة) : بتكتب ملف YAML يعبّر عن "الحالة المطلوبة"، وتعمل kubectl apply -f file.yaml سهل تعيد تشغيله وتحتفظ به في Git
- Imperative : أوامر مباشرة زي kubectl run/create/expose/scale مفيد للمهام لمرة واحدة أو لتوليد قالب YAML بسرعة.

❖ خيارين ذهبيين لتوفير وقتك

- --dry-run=client
  يجرّب الأمر من غير ما ينشئ أي مورد (resource) وبالتالي (يُفيد للتأكد إن الأمر صحيح)
- -o yaml
  يطبع تعريف المورد كـ YAML على الشاشة بدل ما ينفّذه.
- التركيبة السحرية --dry-run=client -o yaml > file.yaml : تولّد لك ملف YAML جاهز تعدّله وتستعمله بـ kubectl apply

---

الـ Pod

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
```

❖ إنشاء بود NGINX مباشرة : kubectl run nginx --image=nginx

- ينشئ Pod اسمه nginx بحاوية صورتها nginx

❖ توليد YAML للبود بدون إنشاءه : kubectl run nginx --image=nginx --dry-run=client -o yaml

- يطلع لك YAML شبيه بـ :
- عدّل لو عايز، واحفظه :
  - kubectl run nginx --image=nginx --dry-run=client -o yaml > nginx-pod.yaml
  - kubectl apply -f nginx-pod.yaml

الـ Deployment

❖ إنشاء Deployment سريع : kubectl create deployment nginx --image=nginx

- ملاحظة مهمة :: بعض إصدارات kubectl create deployment لا تدعم replicas
  - وبالتالي فالحل : أنشئه ثم كبّره : kubectl scale deployment nginx --replicas=3

❖ توليد YAML للديبلويمنت : kubectl create deployment nginx --image=nginx --dry-run=client -o yaml

- أو للحفظ :

kubectl create deployment nginx --image=nginx --dry-run=client -o yaml > nginx-deploy.yaml

- افتح الملف وعدّل أي حقول (مثلاً أضف spec.replicas، أو resources, أو env ....) ثم:

kubectl apply -f nginx-deploy.yaml

الـ Service (ClusterIP / NodePort) و الفرق بين expose و create service

❖ الفكرة الأساسية : الـ Service يختار البودز عبر selectors (ولازم يطابق labels الموجودة على البود/الديبلويمنت)

1. ClusterIP يخدم بود Redis على 6379 :

a) طريقة kubectl expose (دا تلقائياً بياخد الـ Labels بتاع الـ Pod كـ selectors)

- ممتاز لما label بتاع الـ pod تكون مظبوطة لأنّه "ينسخها" للـ selector

```
kubectl expose pod redis \
  --port=6379 \
  --name=redis-service \
  --dry-run=client -o yaml
```

b) kubectl create service clusterip : يفترض (selector=app=redis) :

- تحذير : هذه الطريقة لا تسمح بتمرير –selector
  وهتحتاج تولّد الملف وبعدين تعدّل spec.selector يدويًا
  ليلائم لابلز البود الحقيقية
  (مثلاً app: my-redis بدل app: redis)

```
kubectl create service clusterip redis \
  --tcp=6379:6379 \
  --dry-run=client -o yaml
```

2. NodePort لسيرفس NGINX على 80 ويُفتح 30080 على النودز :

a) kubectl expose (يستخدم labels بتاع الـ pod تلقائيًا لكن لا يقبل --node-port)

```
kubectl expose pod nginx \
  --port=80 \
  --name=nginx-service \
  --type=NodePort \
  --dry-run=client -o yaml
```

- بعدها عدّل الملف يدويًا وأضف nodePort: 30080 تحت spec.ports[0]
- مدى NodePort الافتراضي 30000-32767

b) kubectl create service nodeport (يسمح بتحديد node-port لكن لا ينسخ labels بتاع الـ pod) :

```
kubectl create service nodeport nginx \
  --tcp=80:80 \
  --node-port=30080 \
  --dry-run=client -o yaml
```

- لكن لازم تعدّل spec.selector يدويًا ليتوافق مع الـ labels بتاع البود/الديبلويمنت.

توصية عملية : استخدم kubectl expose لتضمن selectors صحيحة، ولو محتاج nodePort محدد .. فالأفضل تولّد الملف بـ -o yaml ثم أضِف الحقل يدويًا.

مثال تعديل Selector و NodePort في YAML

تخيل ولّدت

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  type: NodePort
  selector:
    app: nginx    # <-- جاي من لابلز البود
  ports:
  - port: 80
    targetPort: 80
    protocol: TCP
```

لو محتاج NodePort معيّن :

```yaml
  ports:
  - port: 80
    targetPort: 80
    nodePort: 30080    # <-- أضف هذا
```

ولو كانت اللابلز على البود مختلفة (مثلاً app: web و tier: frontend)

```yaml
  selector:
    app: web
    tier: frontend
```

ثم : kubectl apply -f nginx-service.yaml

⏱️ نصائح سريعة للامتحان

- دايمًا جرّب الأمر بـ --dry-run=client -o yaml وخزّنه في ملف لأن التعديل على ملف أسهل من حفظ كل الفلاغز.
- تأكد من تطابق labels/selectors بين الـ Service والـ Pod/Deployment لأن دا أكتر سبب يخلي السيرفس مش شايف البودز
- لـ NodePort مخصّص، اشتغل بملف YAML وعدّل nodePort يدويًا.
- لو إصدار kubectl عندك ما بيدعم --replicas في create deployment ..
  استخدم kubectl scale أو عدّل YAML وأضف spec.replicas

Manual Scheduling

في محاضراتنا اللي فاتت، كنا بنشوف إزاي الـ Kubernetes Scheduler هو اللي بيقوم بكل العمليات الصعبة عشان يحدد أفضل Node لكل Pod .. لكن إيه اللي بيحصل لو أنت كمُهندس عايز تتحكم في الموضوع بنفسك وتحدد الـ Node اللي هيشتغل عليها الـ Pod ؟

- هنا بيجي دور الجدولة اليدوية (Manual Scheduling) ، وهو موضوع محاضرتنا النهاردة.

المقدمة :

تتناول هذه المحاضرة مفهوم الجدولة اليدوية (Manual Scheduling) للـ Pods ، وهي طريقة لتحديد الـ Node التي سيتم تشغيل الـ Pod عليها بشكل مباشر، دون الاعتماد على خوارزمية الـ Kubernetes Scheduler

---

الفكرة الأساسية :

في العادة، لما بتنشئ Pod ، بيكون دور الـ Scheduler إنه يجدوله على Node معينة بناءً على موارده المتاحة وظروفه. لكن في بعض الأحيان، قد تحتاج إلى تجاوز هذه العملية.

- مثلاً : لو كنت بتختبر Node معينة، أو لو عندك Pod مهم جداً لازم يشتغل على Node معينة لسبب ما.

ففي الحالات دي، أنت بتتصرف كـ "Scheduler" شخصي، وبتحدد بنفسك مكان الـ Pod

---

تطبيق الفكرة فيKubernetes :

❖ يمكنك جدولة الـ Pods يدوياً من خلال طريقتين:

- الطريقة الأولى: تحديد nodeName في ملف التعريف (تحديدا وانا لسا بعمل create للـ pod) :

الـ Pods اللي بتم إنشاؤها بدون nodeName بيتم جدولتها بواسطة الـ Scheduler لكن لو أنت حددت اسم Node معينة في حقل nodeName داخل ملف تعريف الـ Pod ، فإن الـ Scheduler هيتجاهل الـ Pod ده وهيقوم الـ Kubelet على الـ Node المحددة بإنشاء الـ Pod مباشرة.

مثال توضيحي:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-manual-pod
spec:
  nodeName: node01
  containers:
  - name: my-container
    image: nginx
```

في المثال ده، الـ Pod اللي اسمه `my-manual-pod` هيشتغل مباشرة على الـ Node اللي اسمها `node01` .

- الطريقة الثانية : التحديث اليدوي لـ nodeName (ودا في حالة ان الـ pod اصلا معموله creation)

الفكرة الأساسية هي استخدام الـ Binding Object

- الـ Binding Object هو ببساطة وسيلة لإنشاء علاقة مباشرة بين Pod موجود بالفعل و Node معينة. باستخدام هذا الكائن، أنت بتقلّد (mimic) بالضبط اللي بيعمله الـ Scheduler لما بيختار Node ويجدول عليها Pod
- الهدف :ربط Pod موجود مسبقاً بـ Node معينة.
- العملية :يتم إرسال POST request إلى (API endpoint) معينة في الـ API Server اسمها pod binding API

خطوات تطبيق الفكرة

❖ العملية دي بتتم في خطوتين أساسيتين:

1. إنشاء الـ Binding Object:
   - بتقوم بإنشاء كائن ربط binding object.
   - داخل هذا الكائن، بتحدد اسم الـ Node المستهدفة في حقل target node.
   - لازم تحوّل ملف الـ YAML لـ JSON، وده لأن الـ API بيستقبل البيانات بصيغة JSON.
2. إرسال الـ POST Request:
   - بتبعت الـ POST request للـ Kubernetes API Server، وتحديداً لنقطة النهاية الخاصة بالـ Pod bindings.
   - بتحط البيانات الخاصة بالـ binding object اللي أنشأته في الخطوة الأولى داخل الـ Request.

```
curl --header "Content-Type:application/json" --request POST --data '{"apiVersion":"v1", "kind": "Binding" …. }
http://$SERVER/api/v1/namespaces/default/pods/$PODNAME/binding/
```

الخلاصة وما يحدث عند الفشل

- **لماذا يبقى الـ Pod معلقاً(Pending) ؟** إذا قمت بتحديد nodeName في ملف تعريف الـ Pod وكانت الـ Node دي مش موجودة أو غير متاحة، فإن الـ Pod هيفضل في حالة "Pending"
- كيف يمكنك معرفة السبب؟ يمكنك استخدام أمر kubectl describe pod <pod-name> عشان تشوف الأحداث (Events)الخاصة بالـ Pod حيث هيتم توضيح سبب عدم جدولة الـ Pod

Labels and Selectors

المقدمة

- Labels و Selectors هما طريقة قياسية في Kubernetes علشان:
  - نعمل Grouping (تجميع) للأشياء.
  - نقدر نعمل Filtering (تصفية) ونحدد العناصر اللي عايزينها.

زي ما بنعمل Tags على فيديوهات YouTube أو منتجات في Online Store : نقدر نفلتر بالـ Tags (مثلاً اللون = أخضر، أو النوع = ملابس ... وهكذا)

📌 Labels

- Label = خاصية (Key=Value) بتتضاف لأي Object في Kubernetes
- مثال:
- تقدر تحط أكتر من Label على نفس الـ Pod أو Deployment

```
metadata:
  labels:
    app: myapp
    tier: frontend
```

📌 Selectors

- Selector = شرط بتستخدمه علشان تجيب Objects معينة ليها Labels معينة.
- مثال عملي: kubectl get pods --selector app=myapp
  - ده هيعرض الـ Pods اللي ليها Label app=myapp

تطبيق الفكرة في Kubernetes

- عندك أنواع موارد كتير (Pods, Services, ReplicaSets, Deployments...) ومع الوقت عددهم بيكبر جدًا.
- نستخدم Labels لوسم الموارد بـ معاني مفيدة (app, function, env, tier...) وبعدين نستخدم Selectors لاختيار مجموعة محددة مثل app=App1

إزاي نكتب الـ Labels في YAML

- في تعريف الـ Pod تحت metadata.labels بتحط مفاتيح وقيم زي: (خلي باالك دي وسوم للـ ReplicaSet نفسها)

```
metadata:
  labels:
    app: App1
    tier: frontend
    env: production
```

- تقدر تضيف عدد كبير من الـ Labels .. للاختيار:

kubectl get pods -l app=App1

الاستخدام الداخلي لربط الموارد (ReplicaSet & Service)

❖ Kubernetes بيربط الموارد ببعض باستخدام Labels/Selectors

1 ReplicaSet

- بيحتاج يربط نفسه بالـ Pods اللي هيشغلها.
- ده بيتعمل عن طريق:
  - Labels على الـ Pods
  - Selector جوه ReplicaSet يطابق نفس الـ Labels

🔑 ملاحظة مهمة:

- في ملف ReplicaSet فيه Labels في مكانين:
  1. Labels على الـ ReplicaSet نفسه
  2. Labels تحت template : ودي اللي بتتطبق على الـ Pods

2 Service

- Service بيربط نفسه بالـ Pods اللي هيقدمهم للمستخدم.
- ده بيتم عن طريق Selector في تعريف الـ Service : يطابق الـ Labels بتاعت الـ Pods

```yaml
spec:
  selector:
    app: myapp
    tier: backend
```

Annotations ملاحظات غير قابلة للتصفية

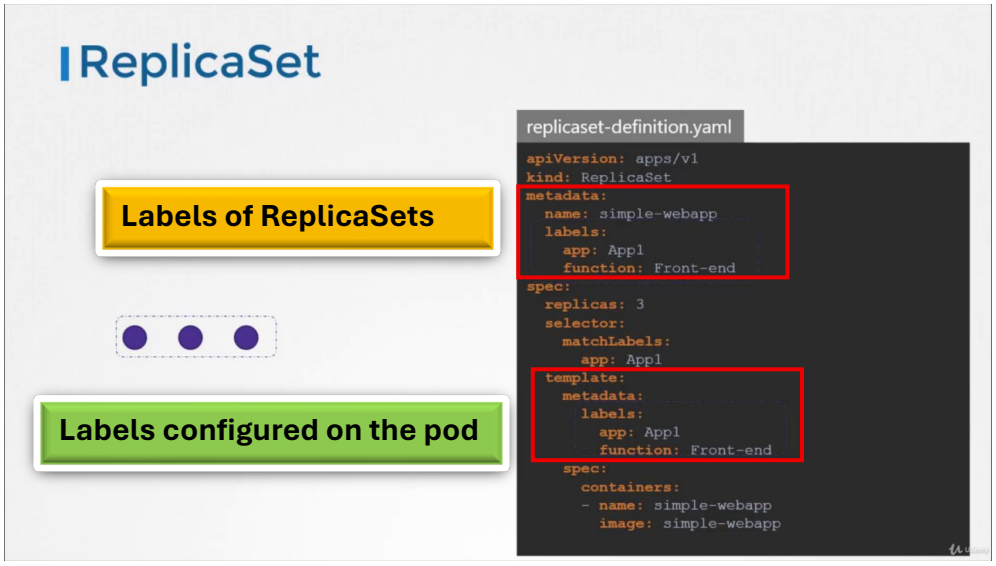
- زي الـ Labels لكنها مش للتجميع أو التصفية.
- بتُستخدم علشان تخزن معلومات إضافية (Metadata) زي:
- Version, Build Info
- Owner, Email, Contact Info
- Integration data لأدوات خارجية .

```yaml
metadata:
  annotations:
    buildVersion: "1.0.3"
    owner: "dev-team@example.com"
```

الخلاصة

- Labels : خصائص (Key:Value) على الكائنات.
- Selectors : شروط لاختيار مجموعات من الكائنات تستخدمها kubectl و Service و ReplicaSet ...
- ReplicaSet : ركّز إن الـ selector يطابق Labels البودات داخل template
- Service : يختار البودات بـ selector
- Annotations : معلومات إضافية غير قابلة للاختيار.

```
kubectl get pods --selector app=App1
NAME              READY   STATUS    RESTARTS   AGE
simple-webapp     0/1     Completed   0        1d
```



📌 الفائدة من Labels & Selectors

- تنظيم الموارد (خصوصًا في مشاريع كبيرة).
- إمكانية الفصل بين بيئات مختلفة:
  - env=dev
  - env=staging
  - env=production
- التحكم في الـ Services :
  - الـ Service يقدر يوجّه الترافيك ل Podsمعينة عن طريق الـ Selector
- المرونة : ممكن تضيف أكتر من Label للـ Object وتفلتر بأي طريقة.

📌 أمثلة عملية

1. تجميع حسب النوع:
   - Label: tier=frontend / tier=backend / tier=database.
   - Selector : هاتلي كل اللي "tier=backend "
2. تجميع حسب البيئة:
   - Label: env=dev / env=prod.
   - Selector : هاتلي كل اللي "env=prod "
3. تجميع معقد (Multiple Criteria) :
   - Label: color=green, type=bird.
   - Selector : هاتلي كل الـ Pods اللي لونها أخضر "وكمان طيور".

📍 الخلاصة:

- Labels : وسوم/خصائص بتحطها على الـ Objects
- Selectors : وسيلة لاختيار مجموعة من الـ Objects بناءً على الـ Labels
- من غير Labels و Selectors : إدارة الكلاستر هتبقى صعبة جدًا.

Taints and Tolerations

المقدمة والمفهوم الأساسي تُستخدم Taints (الشوائب) و Tolerations (التسامحات) لتقييد الـ pods التي يمكن وضعها على nodes معينة

- ❖ Taints = بنحطها على الـNode.
- ❖ Tolerations = بنضيفها على الـ Pod
- ❖ الهدف : تقييد (Restrict) مكان جدولة الـ Pods علي الـ Node (مش متعلق بالأمان أو Security)

علشان تفهم أكتر فهاديلك تشبيه بسيط

- **التشبيه :** يمكن تشبيه الـ node بشخص، والـ pods بالحشرات . لمنع الحشرات من الهبوط على الشخص، يتم رشّ الشخص بمادة طاردة أو "Taint" لكن ممكن بعض الحشرات لا تتحمل هذه الرائحة، وبالتالي لا تهبط على الشخص .
  بينما بعض الحشرات الأخرى " تتسامح Tolerations " مع الرائحة، وبالتالي تهبط على الشخص.
- **خلاصة التشبيه :**

  Node = الشخص

  Pod = الحشرة

  Taint = البخاخ

  Toleration = قدرة الحشرة تتحمل الريحة

كيفية عمل Taints وTolerations

- ❖ Taint بتحط على Node : تمنع أي Pod يشتغل عليه إلا لو عنده Toleration
- ❖ Toleration بتحط على Pod : تسمح له يشتغل على Nodes معينة عليها نفس الـ Taint

📌 مثال عملي

❖ الحالة العادية

- عندنا 3 Nodes (node1, node2, node3) و 3 Pods (A, B, C, D)
- الـ Scheduler بيوزع الـ Pods بالتساوي بدون أي قيود.

❖ إضافة Taint

- نفترض عايزين Node1 يبقى مخصص لتطبيق معين.
- بنحط Taint على Node1: app=blue:NoSchedule
- النتيجة : أي Pod عادي مش هيقدر يشتغل على Node1

❖ إضافة Toleration لـ Pod

- لو Pod D هو الخاص بالتطبيق ده .. بنضيف Toleration جوه تعريفه.
- النتيجة : Pod D الوحيد اللي يقدر يشتغل على Node1

أوامر وخصائص Taints وTolerations

❖ Tainting a Node :

- app=blue = Key/Value للـ Taint.
- NoSchedule = نوع التأثير (Effect).

```
kubectl taint nodes node-name key=value:taint-effect
```

```
kubectl taint nodes node1 app=blue:NoSchedule
```

pod-definition.yml

```yaml
apiVersion:
kind: Pod
metadata:
  name: myapp-pod
spec:
  containers:
  - name: nginx-container
    image: nginx

  tolerations:
  - key:"app"
    operator:"Equal"
    value: blue
    effect: NoSchedule
```

❖ Taint Effects

- يحدد الـ Taint effect ما سيحدث للـ pods التي لا تتحمل الـ Taint .. هناك ثلاثة تأثيرات:
  - NoSchedule : لا يتم جدولة أي pods جديدة على الـ node إذا لم تكن تتسامح مع الـ Taint
  - PreferNoSchedule : يحاول النظام تجنب وضع الـ pods على الـ node ، لكن هذا غير مضمون.
  - NoExecute : لا يتم جدولة أي pods جديدة على الـ node ، والـ pods الموجودة حاليًا على الـ node والتي لا تتسامح مع الـ Taint .. سيتم "إجلائها (evicted) "أو قتلها .
    ولاحظ انه يمكن أن تكون هذه الـ pods قد جُدولت على الـ node قبل تطبيق الـ Taint

❖ إضافة Toleration إلى Pod

- تُضاف الـ Tolerations إلى الـ pods
- يتم إضافة قسم يُسمى tolerations في قسم spec بملف تعريف الـ pod
- يجب أن تتطابق قيم الـ Toleration مع قيم الـ Taint الموجود على الـ node من حيث (المفتاح، القيمة، والتأثير)

نقاط هامة يجب تذكرها

- Taints بتحط على الـ Nodes.
- Tolerations بتحط على الـ Pods.
- وجود Toleration مش معناه Pod لازم يروح Node ده، لأ .. بس معناها إنه مسموح له يشتغل هناك.
- الـ Taints تمنع، ولا تجبر : الـ Taints و Tolerations تمنع الـ nodes من قبول pods معينة
  لكن لا يضمن الـ Taint أن الـ pod سيُوضع على node معين؛ بل يمنع الـ nodes الأخرى من قبول هذا ال pod

واذا كنت تريد تقييد pod على nodes معينة، فإن هذا يتم من خلال مفهوم آخر يُسمى Node Affinity

- الـ Master Node : Master node هو في الأساس node آخر يمكنه استضافة pods . ..
  ولكن بشكل افتراضي، لا يقوم الـ Scheduler بجدولة أي pods على الـ master node
  والسبب هو أنه يتم وضع Taint عليه تلقائيًا عند إعداد الـ cluster

هذه ممارسة مُفضَّلة لعدم نشر أحمال العمل على خوادم الـ master . ويمكنك التحقق من هذا الـ Taint باستخدام الأمر :

❖ kubectl describe node <master-node-name>

Node Selectors

الفكرة الأساسية :

Node Selectors هي طريقة بسيطة ومباشرة للتحكم في أي (node) سيتم وضع Pod معين عليها .يتم ذلك عن طريق مطابقة الـ Labels المحددة على الـ Pod مع الـ Labels الموجودة على الـ Node

- **لماذا نحتاجها؟ في نظام Kubernetes ، قد تحتوي الـ Cluster على أنواع مختلفة من الـNodes ، مثل Nodes صغيرة ذات موارد منخفضة و Nodes كبيرة ذات موارد أعلى .**
- **قد تحتاج إلى تشغيل أحمال عمل معينة (مثل معالجة البيانات التي تتطلب موارد عالية) على الـ Nodes الأكبر فقط .**
- **بشكل افتراضي، يقوم Scheduler بوضع أي Pod على أي Node متاحة**

ولحل هذه المشكلة، يمكن استخدام Node Selectors لتقييد الـ Pods بحيث لا تعمل إلا على Nodes محددة.

📌 المثال الأساسي

- **عندنا Cluster فيه 3 Nodes :**
  - **2 Nodes صغيرين (موارد قليلة).**
  - **1 Node كبير (موارد عالية).**
- **عايزين نضمن إن Data Processing Pod (اللي محتاج CPU/RAM كتير) يشتغل على الـ Node الكبير بس.**

⚙️ Node Selector و إزاي نستخدمه؟

❖ **تعتمد الـ Node Selectors على نظام الـ Labels والـ Selectors الذي تم شرحه سابقًا، ولكنه يُطبَّق على الـ Nodes بدلاً من الـ Pods**

1. **إضافة Label للـ Node**

   لازم الأول نحدد الـ Node الكبير وندي له Label : kubectl label nodes node1 size=large

   - **هنا عملنا Label اسمه size وقيمته large**

2. **تعديل تعريف الـ Pod**

   في ملف الـ Pod definition (YAML) :

   - **بعد تسمية الـ Node ، يتم إضافة خاصية جديدة تُسمى nodeSelector إلى قسم spec في ملف تعريف الـ Pod**
   - **تُستخدم نفس الـ Labels (بصيغة مفتاح-قيمة) التي تم إضافتها إلى الـ Node**

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: data-processor
spec:
  containers:
  - name: processor
    image: data-processor:latest
  nodeSelector:
    size: large
```

النتيجة : Pod هيتـSchedule على Node1 (الكبير) فقط.

مقارنة بـ Taints وTolerations

- **Taints و Tolerations تمنع الـ Node من قبول الـ Pods التي لا تتسامح مع الـ Taint بمعني انها عملية "طرد" أو "إقصاء" للـ Pods من الـ Node**
- **Node Selectors هي عملية "جذب" للـ Pods إلى Node معينة. فهي تجبر الـ Pod على أن يتم وضعه على Node معينة بناءً على الـ Labels**

قيود Node Selectors

- Node Selector بسيط جدًا : بيشتغل بـ Key/Value واحد.
- لكن لو عندك شروط أعقد:
  - مثال : "شغّل على large أو medium"
  - أو: "شغّل على أي Node مش Small"

➡️ ده مش ممكن بـ Node Selector.

علشان كده Kubernetes قدّمت Node Affinity & Anti-Affinity اللي بتديك مرونة أكتر (هنشوفها في الدرس الجاي).

✅ الخلاصة

- Node Selector = وسيلة بسيطة تربط Pod بـ Node عن طريق Label.
- لازم الأول تعمل Label للـ Node وبعدين تستدعيه في Pod definition.
- مناسب للحالات البسيطة.
- للحالات المعقدة → استخدم Node Affinity.

Node Affinity

الفكرة الأساسية:

Node Affinity هي ميزة متقدمة في Kubernetes تضمن أن الـ pods يتم استضافتها على nodes محددة.
تُعد Node Affinity أكثر تطورًا من Node Selectors لأنها تسمح باستخدام تعابير (expressions) أكثر تعقيدًا مثل "OR" أو "NOT" .

📌 **الهدف من Node Affinity**

- الغرض: نضمن إن Pods معينة تشتغل على Nodesمحددة.
- مثال : Pod ضخم لمعالجة البيانات (Data Processing) لازم يشتغل على Node فيها موارد كبيرة (large node)

كيفية عمل Node Affinity :

تستخدم Node Affinity بنية YAML أكثر تعقيدًا من Node Selectors:

- يتم تحديدها في قسم spec الخاص بملف تعريف الـ Pod.
- تحت spec ، تُضاف خاصية affinity ، وتحتها nodeAffinity.
- هناك أنواع مختلفة من Node Affinity تحدد سلوك الـ Scheduler.

Node Affinity

pod-definition.yml

```yaml
apiVersion:
kind: Pod
metadata:
  name: myapp-pod
spec:
  containers:
  - name: data-processor
    image: data-processor
  nodeSelector:
    size: Large
```

pod-definition.yml

```yaml
apiVersion:
kind:
metadata:
  name: myapp-pod
spec:
  containers:
  - name: data-processor
    image: data-processor
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecut
        nodeSelectorTerms:
        - matchExpressions:
          - key: size
            operator: In
            values:
            - Large
```

affinity : جواها الـ Node Affinity

requiredDuringSchedulingIgnoredDuringE xecution ده نوع الـ Affinity (هنشرحه كمان شوية)

matchExpressions : هنا بنحدد الشروط:
key : اسم الـ Label (مثلاً size)
Operator : نوع العملية (In, NotIn, Exists)
Values : القيم المطلوبة (مثلاً large)

أنواع Node Affinity يوجد نوعان أساسيان من Node Affinity:

❖ فيه حالتين مهمة :أثناء الـ Scheduling و أثناء الـ:Execution

1. أثناء Scheduling (إنشاء الـ Pod)
   - requiredDuringSchedulingIgnoredDuringExecution
     - إجباري. لو مفيش Node ينطبق عليها الشرط يبقي Pod مش هيتـ.Schedule
     - مناسب لما مكان الـ Pod مهم جدًا.
   - preferredDuringSchedulingIgnoredDuringExecution
     - مفضل. يحاول يلاقي Node مناسبة.
     - لو ملقاش .. يشغل الـ Pod على أي Node
     - مناسب لو المهم إن الـ Pod يشتغل حتى لو مش على المكان المثالي.
2. أثناء Execution (الـ Pod شغال بالفعل)

- الوضع الحالي = دايمًا IgnoredDuringExecution .. يعني أي تغيير في Labels بعد تشغيل الـ Pod مش بيأثر.
- مستقبلًا: هيضاف نوع اسمه requiredDuringExecution → لو الـ Label اتشال، Pod هيتـEvict (يتقتل).

(Advanced Expressions) :

تتيح Node Affinity استخدام عوامل تشغيل (operators) أكثر تعقيدًا من مجرد المساواة (equals).

يتم تحديد هذه العوامل داخل"node Selector Terms "

- **In : لجدولة الـ Pod على Node يكون فيها الـ size Label له قيمة "large" أو "medium".**
- **NotIn : لجدولة الـ Pod على Node ليس لديها قيمة small للـ size Label.**
- **Exists : للتحقق ببساطة من وجود Label معين على الـ Node، بغض النظر عن قيمته.**

📝 **مثال عملي**

لو عايز Pod يشتغل على أي Node حجمها كبير أو متوسط:

```
matchExpressions:
- key: size
  operator: In
  values:
  - large
  - medium
```

ولو عايزه يتجنب Nodes صغيرة:

```
matchExpressions:
- key: size
  operator: NotIn
  values:
  - small
```

مقارنة بـ Node Selectors

- **Node Selectors هي طريقة بسيطة للجدولة على أساس التطابق التام للمفاتيح والقيم.**
- **Node Affinity هي ميزة أكثر قوة وتوفر تحكمًا أكبر في الجدولة.**

Taints & Tolerations vs. Node Affinity

🎬 المقدمة

- قبل كده اتعلمنا عن:
  - Taints & Tolerations : بتخلي Node ترفض أي Pod إلا لو عنده Toleration
  - Node Affinity : بتجبر Pod إنه يروح ل Nodes معينة (بناءً على Labels)
- في المحاضرة دي هنشوف إزاي نستخدم الاتنين مع بعض علشان نخصص Nodes بالكامل لتطبيقات محددة.

📌 السيناريو

- عندنا 3 Nodes (أزرق – أحمر – أخضر)
- وعندنا 3 Pods (أزرق – أحمر – أخضر)
- الهدف:
  - Pod الأزرق → يشتغل على Node الأزرق.
  - Pod الأحمر → يشتغل على Node الأحمر.
  - Pod الأخضر → يشتغل على Node الأخضر.
- كمان فيه فرق تانية بتشاركنا نفس الـ Cluster
  - مش عايزين Pods بتاعتهم تتسكن على الـ Nodes بتاعتنا.
  - ومش عايزين Pods بتاعتنا تتسكن على الـ Nodes بتاعتهم.

🛠️ المحاولة الأولى Taints & Tolerations :

1. نعمل Taint على كل Node مثلًا: kubectl label nodes node1 color=blue
   - color=blue:NoSchedule على Node الأزرق.
   - color=red:NoSchedule على Node الأحمر.
   - color=green:NoSchedule على Node الأخضر
2. ندي لكل Pod Toleration لنفس اللون.

📍 النتيجة:

- Pod الأزرق بيتسكن على Node الأزرق. ✅
- Pod الأخضر بيتسكن على Node الأخضر. ✅
- لكن Pod الأحمر ممكن يروح على Node تاني (لو مش معموله Taint). ❌

➡️ السبب Taints & Tolerations: تمنع الـ Pods الغلط من دخول Node ، لكنها مش بتجبر Pod يروح Node معين.

🛠️ المحاولة الثانية Node Affinity :

1. ندي لكل Node Label مثلًا.(color=blue) :
2. ندي لكل Pod Node Affinity يحدد اللون المطلوب.

📍 النتيجة:

- Pod الأزرق بيروح على Node الأزرق. ✅
- Pod الأحمر بيروح على Node الأحمر. ✅
- Pod الأخضر بيروح على Node الأخضر. ✅

➡️ لكن: مفيش منع يخلي Pods تانية (من فرق تانية) تدخل على نفس الـ Nodes. ❌

🛠️ **الحل النهائي : الدمج بين الاتنين**

- نستخدم Taints & Tolerations : علشان نمنع أي Pods غريبة من إنها تتسكن على الـ Nodes بتاعتنا.
- نستخدم Node Affinity : علشان نضمن إن Pods بتاعتنا تتسكن على الـ Nodes الصح.

📍 النتيجة:

- الـ Pods بتاعتنا تروح على الـ Nodes المخصصة ليها.
- الـ Nodes بتاعتنا مش بتقبل أي Pods تانية من خارج تطبيقنا.

✅ **الخلاصة**

- Taints/Tolerations : أداة دفاعية → تمنع Pods معينة من دخول Node.
- Node Affinity : أداة توجيهية → بتجبر Pod يروح Node معين.
- الاتنين مع بعض = سيطرة كاملة : Node مخصص لتطبيق واحد، والـ Pod بتاع التطبيق بيروح على الـ Node ده بالظبط.

Resource Requirements and Limits

تتحدث المحاضرة عن Resource Requests & Limits(متطلبات وحدود الموارد) في Kubernetes ، وهما خاصيتان مهمتان للغاية لضمان عمل الـ Pods بكفاءة وفعالية. تساعد هذه الخصائص في إدارة الموارد على الـ Nodes وتمنع استهلاكها بشكل عشوائي.

الفكرة ولماذا نحتاجها (تشبيه بالموارد)

تخيل أن كل Node في مجموعة Kubernetes هي قطعة أرض واسعة، وتحتوي على موارد محدودة مثل الماء (Memory) والكهرباء (CPU) عندما تضع Podعلى هذه الأرض، فإن هذا الـ Pod يستهلك جزءًا من هذه الموارد.

الهدف : إدارة Resources المتاحة على كل Node بفعالية، وضمان أن كل Pod يحصل على ما يكفيه للعمل، وفي نفس الوقت لا يستهلك أكثر من حقه.

- عشان ده يشتغل :بنحدد لكل Pod كمية الموارد التي "يطلبها" وكمية الموارد التي "يُحد" بها.
- ماذا يحدث؟
  - (Scheduler) يستخدم هذه المعلومات ليقرر أي Node لديها موارد كافية لاستضافة الـ Pod
  - إدارة الموارد : تمنع الـ Pod من أن "يخرج عن السيطرة" ويستهلك كل الموارد المتاحة على الـNode ، مما قد يؤدي إلى تعطيل الـ Pods الأخرى.

تطبيق الفكرة فيKubernetes

عندك أنواع موارد أساسية هي الـ CPU (وحدة المعالجة المركزية) و Memory (الذاكرة). نستخدم خاصيتين للتحكم في استهلاك هذه الموارد:

- **Resource Requests :**
  - هي الحد الأدنى من الموارد التي يطلبها الـ Pod
  - يستخدمها Scheduler في عملية (Scheduling) للتأكد من أن الـ Pod سيُوضع على Node بها موارد كافية.
  - إذا لم تتوفر الموارد المطلوبة، يظل الـ Pod في حالة Pending معلّق
    يمكنك رؤية سبب هذا التعليق في الأحداث (Events)، والذي قد يكون "insufficient cpu" (وحدة معالجة مركزية غير كافية) أو "insufficient memory" (ذاكرة غير كافية).
  - القيمة الافتراضية: في الإعدادات الافتراضية، يفترض Kubernetes أن الـ Pod يطلب CPU 0.5 و 256 MiB من الذاكرة.
- **Resource Limits :**
  - هي الحد الأقصى من الموارد التي يُسمح للـ Pod باستهلاكها.
  - تمنع الـ Pod من تجاوز هذا الحد.

إزاي نكتبها فيYAML

في تعريف الـPod ، بنحط الـ requests و limits تحت قسم resources داخل (container)

```yaml
spec:
  containers:
  - name: my-container
    image: my-image
    resources:
      requests:
        memory: "1Gi"
        cpu: "1"
      limits:
        memory: "2Gi"
        cpu: "2"
```

- الـ CPU : تُقاس بالـ "core" أو "m" ملي كور . "1" تعني " 1 core " كامل ، و 500m تعني نصف core
- الـ Memory : تُقاس بالـ "Mi" (ميجابايت) أو "Gi" جيجابايت

ماذا يحدث عند تجاوز الحدود

- الـ Requests لا يتم تجاوزها، لأن الـ Pod لن يُوضع أصلاً على Node لا تتوفر بها الموارد المطلوبة .
- لكن الـ Limits يمكن تجاوزها، وعندها يتصرف Kubernetes بناءً على نوع المورد:
  - عند تجاوز حد الـ CPU : يقوم Kubernetes بعملية "throttling"(التقييد)، حيث يقلل من سرعة الـ Pod لضمان عدم تجاوز الحد المسموح به يعني الـ Pod هايستمر في العمل، ولكن بكفاءة أقل.
  - عند تجاوز حد الـ Memory : يتم إنهاء (terminate) الـ Pod بشكل فوري. وهذا هو السلوك الافتراضي لحماية الـ Node والـ Pods الأخرى من الانهيار.

الخلاصة :

- Requests:
  - تُستخدم للـ (Scheduling)
  - تحدد الحد الأدنى من الموارد.
  - إذا لم تتوفر الموارد، يظل الـ Pod في حالة Pending
- Limits:
  - تُستخدم للتحكم في استهلاك الموارد.
  - تحدد الحد الأقصى المسموح به.
  - إذا تم تجاوز الحد، يحدث throttling للـ CPU أو termination للـ Memory

```
NAME        READY     STATUS    RESTARTS   AGE
Nginx       0/1       Pending   0          7m

Events:
  Reason                    Message
  ------                    -------
  FailedScheduling  No nodes are available that match all of the following predicates:: Insufficient cpu (3).
```

How to Assign Memory Resources to Containers and Pods :

https://kubernetes.io/docs/tasks/configure-pod-container/assign-memory-resource/

How to Assign CPU Resources to Containers and Pods :

https://kubernetes.io/docs/tasks/configure-pod-container/assign-cpu-resource/

DaemonSets

المقدمة

تتناول هذه المحاضرة مفهوم DaemonSets في Kubernetes حيث يتم التركيز على كيفية نشر أنواع معينة من الـ Pods في Kubernetes Cluster

---

الفكرة الأساسية

لقد قمنا بنشر العديد من الـ Pods على Nodes المختلفة في الـ Cluster الخاصة بنا . وبمساعدة ReplicaSetsو Deployments، قمنا بضمان أن تتوفر نسخ متعددة من تطبيقاتنا على مختلف العقد العاملة.

الـ DaemonSet يشبه الـ ReplicaSet في أنه يساعدك على نشر نسخ متعددة من الـ Pod
ولكن الفارق الرئيسي هو أن الـ DaemonSet يقوم بتشغيل نسخة واحدة من الـ Pod على كل Node في الـ Cluster

So DaemonSets helps you deploy Multiple Instances of Pod But, runs One copy of your pod on each node in your cluster

- متى يتم إضافة Pod ؟
  - عند إضافة Node جديدة إلى الـCluster ، تتم إضافة نسخة من الـ Pod تلقائيًا إلى تلك الـ Node
- متى يتم حذف Pod ؟
  - عند إزالة Node ، يتم حذف الـ Pod تلقائيًا معها.

وبالتالي فال DaemonSet هذا يضمن أن نسخة واحدة من الـ Pod تكون موجودة دائمًا على جميع الـ Nodes في الـ Cluster

---

تطبيق الفكرة في Kubernetes

تُستخدم الـ DaemonSets في حالات استخدام محددة، مثل نشر أداة مراقبة أو أداة تجميع سجلات (log collector) على كل Node في الـ Cluster . هذا يساعد على مراقبة الـ Cluster بشكل أفضل.

- أمثلة على حالات الاستخدام:
  - أداة مراقبة :إذا كنت تريد نشر أداة لمراقبة كلNode ، فإن الـ DaemonSet هو الأداة المثالية لذلك.
  - تجميع السجلات :يمكن استخدام DaemonSet لنشر Pod يقوم بجمع السجلات من كل.Node

---

كيفية كتابة الـ DaemonSet في YAML

يتم استخدام نفس فكرة الـ Pod template الموجودة في الـ ReplicaSets و Deployments

- يمكنك كتابة ملف تعريف الـ DaemonSet ، والذي يحدد الـ Pod الذي تريد تشغيله. ثم يقوم الـ DaemonSet بإنشاء نسخة من هذا الـ Pod على كل Node في الـ Cluster
- يمكنك استخدام أمر kubectl get daemonset لرؤية الـ DaemonSet
  وأمر kubectl describe daemonSet للحصول على مزيد من التفاصيل.

Use Cases :

Kube-Proxy, Networking, Monitoring Solution, Log Viewer

الخلاصة

- **الهدف :ضمان وجود نسخة واحدة من الـ Pod على كل Node في الـ Cluster**
- **العملية:**
  - **عند إضافة Node جديدة، يقوم الـ DaemonSet تلقائيًا بإضافة Pod جديد إليها.**
  - **عند إزالة Node ، يقوم الـ DaemonSet تلقائيًا بإزالة الـ Pod منها.**
- **الاستخدام :يُعد الحل المثالي لمهام مثل المراقبة وتجميع السجلات التي يجب أن تعمل على كل Node**

Note >>> Main Difference between ReplicaSets and DaemonSets :

الفرق الرئيسي هو في الهدف من التوزيع (Distribution Goal):

- **DaemonSet : يضمن تشغيل نسخة واحدة من الـ Pod على كل Node في الـCluster و الهدف هو أن تكون الخدمة متاحة على كلNode ويستخدم غالبًا لتشغيل خدمات أساسية على مستوى الـ Node مثل أدوات المراقبة أو تسجيل الدخول.**

- **ReplicaSet : يضمن تشغيل عدد محدد من نُسخ الـ Pods على مستوى الـ Cluster ككل، بغض النظر عن عدد الـ Nodes.الهدف هو الحفاظ على التوافر العالي للخدمة (High Availability) وتوزيع الحمل. الـ Pods دي ممكن تكون على Nodeواحدة أو تتوزع على Nodes مختلفة، وده بيعتمد على الموارد المتاحة وحالة الـCluster ، لكن الـ ReplicaSet بيحاول يوزعها قدر الإمكان.**

Kubectl get ds –all-namespaces

Static Pods

في محاضراتنا السابقة، كنا دائمًا نؤكد على أن كل شيء في Kubernetes يتم عن طريق Kubernetes API Server فمثلاً، عشان نعمل Pod ، بنبعت ملف تعريف الـ Pod لـ API Server ، وبعدين الـ Scheduler بيشتغل ويقرر فين هيتحط الـ Pod ده، وبعدها الـ Kubelet بينفذ التعليمات.

النهاردة، هنتعلم عن مفهوم مختلف تماماً اسمه Static Pods وهو بيعتبر استثناء للقاعدة دي.

المقدمة:

- السؤال: "لو مفيشAPI server ، ولاScheduler ، ولا Cluster أساسًا، هل الـ kubelet يقدر يشتغل لوحده؟"
- الإجابة: آه 👍 عن طريق Static Pods

الفكرة الأساسية:

تخيلوا إن الـ Kubelet هو الموظف المسؤول عن تشغيل الـ Pods على Node معينة.
في الوضع الطبيعي، الـ Kubelet بيستنى الأوامر من الـ API Server عشان يعرف إيه الـ Pods اللي المفروض يشغلها.
لكن، في حالة Static Pods، الـ Kubelet بيشتغل لوحده من غير ما يرجع للـ API Server

الفكرة ببساطة هي إن الـ Kubelet بيقرأ ملف تعريف الـ Pod مباشرة من مكان محدد (ملف محلي أو رابط ويب) ويبدأ في تشغيله.

- تشبيه بسيط :اعتبر الـ Kubelet زي طباخ عنده دفتر وصفات. في الطريقة العادية، صاحب المطعم (API Server) بيقوله "اطبخ الطبق الفلاني "شغّل الـ Pod ده .
  في حالة Static Pods ، الطباخ بيلاقي وصفة موجودة في المطبخ (ملف محلي) فبيقرر إنه يطبخها من نفسه، من غير ما يرجع لصاحب المطعم.

تطبيق الفكرة في Kubernetes

تُستخدم الـ Static Pods في حالات مهمة جداً، وأهمها هو تشغيل مكونات Control Plane(مستوى التحكم) الخاصة بـ Kubernetes نفسها، زي API Server و Scheduler و Controller Manager .

- ليه بنستخدمها هنا؟ لو فكرنا، لو الـ API Server نفسه كان محتاج API Server عشان يشتغل، كنا هنقع في مشكلة "البيضة ولا الفرخة".
  عشان كده، بنستخدم Static Pods عشان نشغل المكونات الأساسية للنظام مباشرة على الـ Master Node ، وده بيضمن إن الـ Cluster يفضل شغال حتى لو حصلت مشكلة في الـ API Server

- كيف يعمل Kubelet ؟ الـ Kubelet بيراقب (Path) معين على الجهاز مثل /etc/kubernetes/manifests
  يعني أي ملف تعريف للـ Pod (اللي هو ملف YAML) بيتم وضعه في هذا المسار،
  يقوم الـ Kubelet تلقائياً بإنشاء وتشغيل الـ Pod الخاص به. وإذا تم حذف الملف، يتم إنهاء الـ Pod

- الـ Mirror Pod : بما أن الـ Static Pods لا يتم إدارتها من خلال الـ API Server ، فإن الـ Kubelet يقوم تلقائياً بإنشاء ما يسمى بـ Mirror Pod (حاوية معكوسة) في الـ API Server
  هذا الـ Mirror Pod هو مجرد تمثيل للـ Static Pod ، وليس الـ Pod الحقيقي، ووجوده يسمح لك برؤية الـ Static Pod باستخدام أمر kubectl get pods

---

الفكرة الأساسية : طرق تحديد مسار الـ Static Pods :

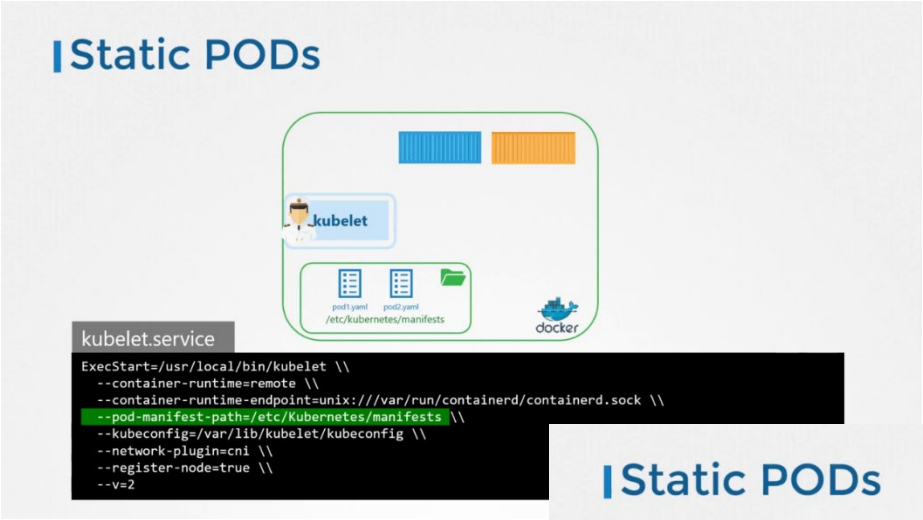
- **في الأساس، هناك طريقتان رئيسيتان لإخبار الـ Kubelet بمكان ملفات الـ Static Pods**
  - **الطريقة المباشرة : (Directly in the kubelet.service file) اللي موجود في /etc/kubernetes/manifests وهي أن تقوم بتعديل ملف الـ Kubelet.service مباشرةً وإضافة خيار -- staticPodPath إليه.**
  - **الطريقة البديلة : (Using a separate config file) يعني بدلاً من تعديل ملف الـ Kubelet.service مباشرةً، يمكنك أن تُشير إلى ملف إعدادات (config file) خارجي باستخدام خيار –config**

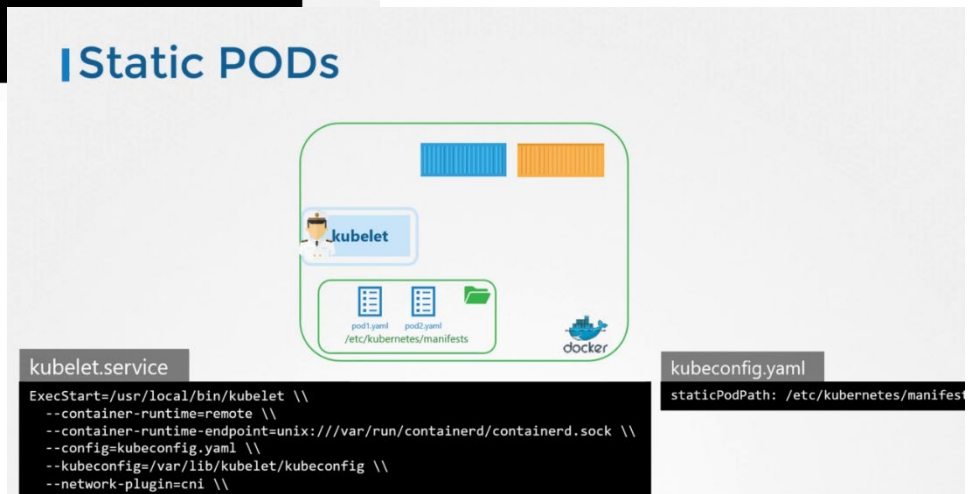
ببساطة : بدلاً من وضع الإعدادات مباشرةً في ملف واحد، تقوم بوضعها في ملف منفصل وتقول للـ" Kubelet اذهب إلى هذا الملف واقرأ إعداداتك منه". ده بيخلي الإعدادات أكثر تنظيماً وسهلة الإدارة. (ودي تعتبر نفس الفكرة في اللينكس لما تيجي تعدل في config معين فالأفضل انك تاخد نسخة وتعدل في النسخة وتسيب الاصل زي ماهو)

- **تطبيق الفكرة عملياً**

الطريقة الثانية دي هي اللي بتستخدم بشكل أساسي في أدوات إعداد الكلاسترز المشهورة زي kubeadm وده بيخليها الطريقة القياسية والمنتشرة.

- **ليه بيستخدموا الطريقة دي؟ لأنها بتخلي إعدادات الـ Kubelet أكثر مرونة وسهلة التغيير بدون المساس بملف الخدمة الأساسي.**
- **ما أهمية ذلك بالنسبة لك ؟ لو أنت بتفحص كلاستر موجود بالفعل، ومش عارف فين مكان ملفات الـ Static Pods ، يبقى أفضل حاجة تعملها إنك تدور على خيار --config في إعدادات الـ Kubelet عشان تعرف مسار ملف الإعدادات. وداخل هذا الملف، هتلاقي متغير اسمه staticPodPath بيحدد لك المسار الصحيح للملفات. وبمجرد ما تعرف المسار ده، هتعرف تحط ملف تعريف الـ Static Pod (ملف الـ YAML) فيه وبتعرف الليلة دي من خلال cd /etc/system/system/kubelet.service.d/**
- **وعلشان تشوفهم فبتستخدم أمر docker ps**





لماذا نستخدم أمر docker ps وليس kubectl؟

تخيل إن الـ kubectl هو الطريقة اللي بتتكلم بيها مع الـ Kubernetes API Server فلو مفيش API Server زي في حالة إنك لسه بتبني الـCluster ، يبقى مفيش وسيلة لـ kubectl إنه يشتغل.

في الحالة دي، بنستخدم أمر docker ps أو أي أمر خاص بالـ Container Runtime اللي بتستخدمه (عشان نشوف الحاويات اللي بتشتغل مباشرةً على الـ Node . وده بيثبت إن الـ Pods موجودة وبتشتغل فعلاً، حتى لو الـ Kubernetes Cluster لسه مكتملش.

كيف يعمل الـ Kubelet مع الـ Static Pods والـ Pods العادية؟

الـ Kubelet ذكي جداً، بيقدر يستقبل أوامر تشغيل الـ Pods من مصدرين مختلفين في نفس الوقت:

- من ملفات YAML : بيراقب مسار معين على الـNode ، وأي ملف YAML لتعريف Pod بيتحط في المسار ده فالـ Kubelet بيشغله على طول. ودي هي طريقة الـ Static Pods
- من الـ API Server :بيستقبل الأوامر عبر HTTP API من الـ Kubernetes API Server، وده اللي بيحصل مع الـ Pods العادية.

الـ Kubelet بيقدر يشغل الـ Static Pods والـ Pods العادية في نفس الوقت.

ما هو الـ Mirror Pod بالتفصيل ؟

- بما إن الـ Static Pod بيتم تشغيله بمعرفة الـ Kubelet مباشرةً ومش عن طريق الـAPI Server ، ده ممكن يخلي الـ API Serverغير مدرك لوجود الـ Pod ده.
- عشان يحل المشكلة دي، الـ Kubelet بيعمل حاجة اسمها Mirror Pod في الـ API Server والـ Mirror Pod ده هو مجرد "صورة طبق الأصل" للـStatic Pod ، وبيظهر لك لما بتستخدم أمر kubectl get pods
- ملاحظات هامة:
  - الـ Mirror Pod هو مجرد تمثيل للقراءة فقط (read-only)
  - ماتقدرش تعدّل أو تحذف الـ Pod باستخدام kubectl وعشان تحذفه، لازم تحذف ملف الـ YAML الأساسي من المسار المحدد على الـ Node
  - اسم الـ Static Pod بيتم إضافة اسم الـ Node تلقائياً إليه مثالpod-name-node01
- ---

الخلاصة بين Static Pods vs. DaemonSets :

البعض قد يخلط بين Static Pods و DaemonSets لأن كلاهما يقوم بتشغيل Pods على Nodes محددة. لكن هناك فروقات جوهرية:

الميزة	Static Pods	DaemonSets
التحكم والإدارة	يتم إنشاؤها وإدارتها **مباشرة** بواسطة الـ Kubelet.	يتم إنشاؤها وإدارتها بواسطة **DaemonSet Controller** عبر الـ API Server.
الهدف	تشغيل Pods على **Node محددة** (عادةً الـ Master Node).	ضمان تشغيل نسخة واحدة من الـ Pod على **كل Node** في الـ Cluster (أو مجموعة محددة).
المرونة	أقل مرونة، لأنها مرتبطة بـ Node واحدة فقط.	أكثر مرونة، حيث يمكنك التحكم في مجموعة الـ Nodes التي يتم وضع الـ Pods عليها.

Static PODs	DaemonSets
Created by the Kubelet	Created by Kube-API server (DaemonSet Controller)
Deploy Control Plane components as Static Pods	Deploy Monitoring Agents, Logging Agents on nodes
Ignored by the Kube-Scheduler	

Multiple Schedulers

في محاضراتنا السابقة، كنا بنعتمد على الـ Default Scheduler عشان يحدد الـ Node المناسبة لأي Pod .
الـ Default Scheduler بيستخدم (Algorithm) عشان يوزع الـ Pods بشكل متساوي، وكمان بياخد في اعتباره شروط زي الـ Taints & Tolerations و Node Affinity

لكن السؤال المهم هو: "إيه اللي ممكن يحصل لو خوارزمية الـ Default Scheduler ما كانتش مناسبة لاحتياجاتك الخاصة؟"

الفكرة الأساسية: إضافة Scheduler خاص بك :

تخيل إن عندك تطبيق معين بيحتاج شروط إضافية ومختلفة عشان يتحط على Node ، مش موجودة في خوارزمية الـ Default Scheduler هنا بتيجي قوة Kubernetes في إنه نظام مرن جداً(highly extensible) ، وبيسمحلك تكتب الـ Scheduler الخاص بك.

- الهدف : أن يكون لديك أكثر من Scheduler في نفس الـ Cluster
- العملية:
  - يمكنك استخدام الـ kube-scheduler الأساسي، أو تبني الـ Scheduler الخاص بك من الصفر.
  - تقوم بنشره كـ Scheduler إضافي بجانب الـ Default Scheduler
  - التطبيقات العادية هتستخدم الـ Default Scheduler ، لكن التطبيق اللي محتاج شروطك الخاصة هيستخدم الـ Scheduler الجديد.

كيفية إعداد Scheduler مخصص :

تستخدم أداة kubeadm الـ Scheduler كـ Static Pod حيث ان ملف التعريف الخاص به بيكون موجود في مجلد الـ manifests

❖ خطوات الإعداد:

1. تاخد نسخة من ملف تعريف الـ kube-scheduler وبتبدأ من امر cd /etc/system/system/kubelet.service.d/ وبعدين cat "o/p" وبعدين grep -l static "o/p" وبعدين cd "o/p" وبعدين ls وبعدين cp
2. تغيّر اسم الـ Pod لاسم مخصص مثل my-custom-scheduler
3. في قسم command، بتضيف خيار جديد عشان تحدد اسم الـ Scheduler الجديد. هذا الاسم مهم جداً، لأنه الاسم اللي هتستخدمه عشان تخلي أي Pod يستخدم الـ Scheduler ده.

❖ **خيار leader-elect :**

الـ leader-elect هو خيار مهم جداً بيمنع حدوث أي تعارض لو عندك أكتر من نسخة من الـ Scheduler شغالين في نفس الوقت .... عشان نفهم الموضوع ده بشكل أوضح، خلينا نقسّمه لسيناريوهين:

- الحالة الأولى : لما يكون عندك Master Node واحدة بس
  - في الحالة دي، بما إن عندك نسخة واحدة من الـKubernetes Master ، فانت مش محتاج تعمل عملية اختيار قائد (leader election)ببساطة لان الـ Scheduler اللي عندك هو القائد الوحيد . وعشان كده لازم تخلي قيمة leader-elect بـ False .. وبالتبعية ده بيخلي الـ Scheduler يتجاهل عملية اختيار القائد تماماً، وده بيناسب إعداد الـ Cluster اللي مش مبني على خاصية الـ High Availability
- الحالة الثانية: لما يكون عندك أكتر منMaster Node
  - هنا الوضع بيختلف، لأن عندك أكتر منMaster Node ، وكل واحدة منهم ممكن يكون عليها نسخة من الـ Scheduler و لو كلهم كانوا شغالين في نفس الوقت، هيحصل تعارض ومشاكل في عملية جدولة الـ Pods وعشان كدا بتخلي قيمة leader-elect بـ True

وبالتبعية هذا الخيار بيسمح للـ Schedulers إنهم يختاروا "قائد" واحد منهم بس يكون هو المسؤول عن الجدولة في أي وقت. و كمان لازم تضيف مُعامل إضافي اسمه **lock object name**

وظيفة الـ lock object name هو اللي بيميز الـ Scheduler المخصص (Custom Scheduler) بتاعك عن الـ Scheduler الافتراضي(Default Scheduler)

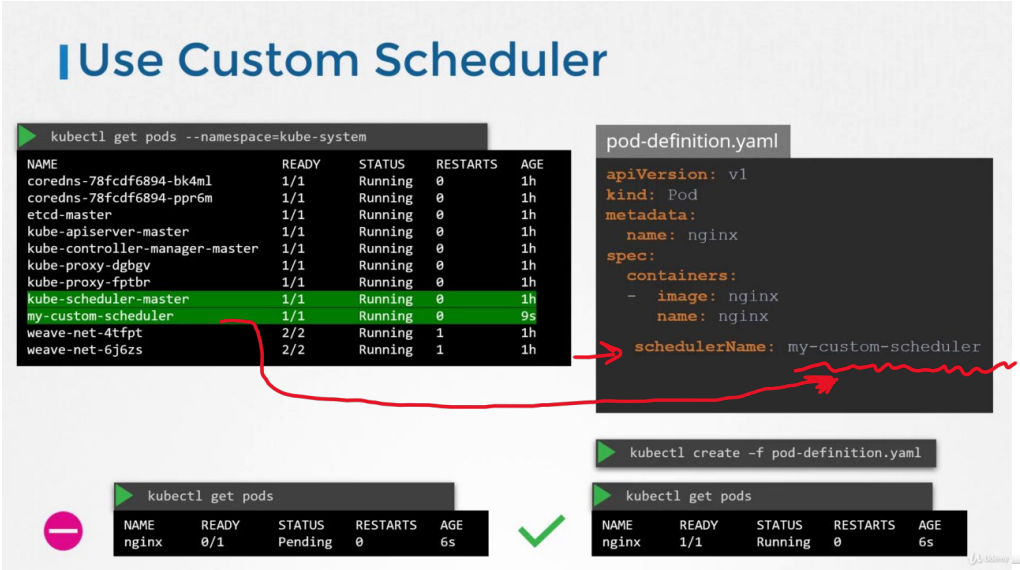
- وبالتالي ده بيضمن إن كل مجموعة من الـ Schedulers (الافتراضي والمخصص) بتعمل عملية اختيار القائد بشكل منفصل، عشان مايحصلش أي لخبطة في الـ Cluster

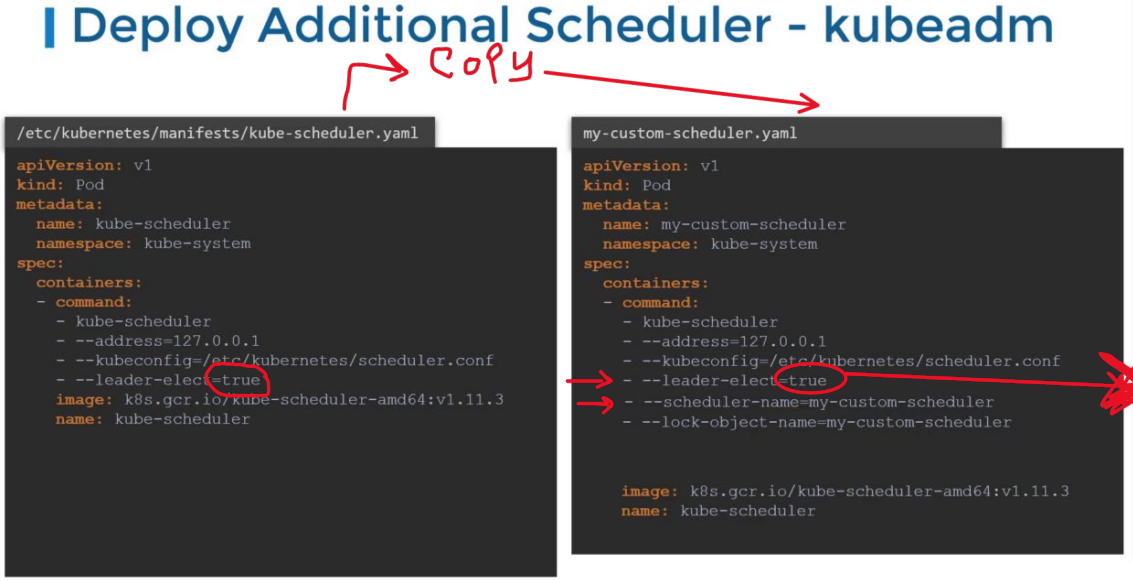
---

- **كيفية استخدام الـ Scheduler المخصص**

بمجرد ما الـ Scheduler الجديد يشتغل ويكون حالته Running، كل اللي عليك إنك تضيفه لملف تعريف الـ Pod الخاص بك.

- الخطوات:
  1. في ملف تعريف الـ Pod (ملف YAML) ، بتضيف حقل اسمه schedulerName
  2. بتحط قيمة الحقل ده باسم الـ Scheduler اللي أنشأته.
  3. لما بتنشئ الـPod ، الـ Scheduler الخاص بك هو اللي هيقوم بمهمة جدولته.
- التحقق من عمل الـ Scheduler
  - لو الـ Scheduler مش متظبط صح، الـ Pod هيفضل في حالة Pending
  - عشان تتأكد إن الـ Scheduler الجديد هو اللي قام بجدولة الـPod ، ممكن تستخدم أمر kubectl get events هتلاقي في الـ Events إن مصدر الحدث هو الـ Scheduler اللي أنشأته.
  - كمان، تقدر تشوف سجلات (logs) الـ Scheduler باستخدام أمر kubectl logs





Remember: To get multiple schedulers working, you must either set the "leader-elect" option to False and this in case where you don't have multiple masters. But in case you have multiple masters, you can set it as True and set an additional parameter to set a "look object name" and this to differentiate the new custom scheduler from the default during the leader election process.

Monitor Cluster Components

النهاردة، هنتعمق أكتر في مراقبة (Monitoring) الـ Kubernetes Cluster، وهنشوف إزاي نعرف حالة الموارد واستهلاكها على مستوى الـ Nodes والـ Pods.

المقدمة :

تتناول هذه المحاضرة مفهوم مراقبة الـ Kubernetes Cluster، مع التركيز على أهم أنواع البيانات التي يجب مراقبتها، والحلول المتاحة لذلك.

الأفكار الأساسية :

- ماذا يجب أن نراقب؟
  - Metrics على مستوى الـ Nodes : مثل عدد الـ Nodes في الـ Cluster، وحالتها (صحية أم لا)، بالإضافة إلى Metrics الأداء مثل استهلاك وحدة المعالجة المركزية (CPU)، والذاكرة (Memory)، واستخدام الشبكة (Network)، والقرص (Disk).
  - Metrics على مستوى الـ Pods : مثل عدد الـ Pods، و Metrics of performance الخاصة بكل Pod مثل استهلاك الـ CPU والـ Memory.
- هل يأتي Kubernetes بحل مراقبة جاهز؟
  - لا، Kubernetes لا يأتي بحل مراقبة متكامل وجاهز للاستخدام مباشرة.
  - ولكن هناك العديد من الحلول مفتوحة المصدر مثل Metrics-Server، و Prometheus، و Elastic Stack.
  - كان هناك مشروع قديم اسمه Heapster للمراقبة، ولكنه أصبح مهملاً (Deprecated) الآن، وظهرت بدلاً منه نسخة أبسط تُعرف باسم Metrics-Server.

تطبيق الفكرة Metrics-Server :

الـ Metrics-Server هو النسخة المُبسطة والبديلة لـ Heapster.

- وظيفته: يقوم Metrics-Server بسحب الMetrics من كل الـ Nodes والـ Pods في الـ Cluster، ويجمعها ويخزنها في الذاكرة (in-memory).
- ملاحظة هامة: بما أن البيانات تُخزن في الذاكرة، فإن الـ Metrics-Server لا يحتفظ بالبيانات التاريخية (historical data). إذا كنت تحتاج بيانات تاريخية، يجب عليك استخدام حلول مراقبة متقدمة أخرى مثل Prometheus.

كيف يتم توليد الـ Metrics ؟

- كل Node في الـ Kubernetes Cluster لديها (agent) اسمه Kubelet.
- الـ Kubelet يحتوي على مكون فرعي اسمه (Container Advisor) cAdvisor.
- وظيفة cAdvisor هي الحصول على Metrics of performance من الـ Pods الموجودة على الـ Node، وتقديمها من خلال واجهة الـ API الخاصة بالـ Kubelet، لتكون متاحة لـ Metrics-Server.

كيفية نشر Metrics-Server وعرض الـ Metrics ؟

- **نشر الـ Metrics-Server:**
  - **على Minikube: استخدم الأمر minikube addons enable metrics-server**
  - **في البيئات الأخرى: قم باستنساخ (cloning) ملفات النشر من مستودع Metrics-Server على GitHub وذلك من خلال امر git clone ، ثم قم بنشرها باستخدام الأمر kubectl create . هذا الأمر يقوم بنشر مجموعة من الـ Pods والـ Services والـ Roles اللي بتخلي Metrics-Server قادر على سحب Metrics الأداء من الـ Nodes في الـ Cluster.**

- **عرض الـ Metrics :**
  - **بعد نشر الـ Metrics-Server وإعطائه بعض الوقت لجمع البيانات، يمكنك عرض الأداء.**
  - **لـ Metrics الـ Nodes: استخدم الأمر kubectl top node هذا الأمر سيوفر استهلاك الـ CPU والذاكرة لكل Nodes.**
  - **لـ Metrics الـ Pods : استخدم الأمر kubectl top pod**

فحص الحالة واللوجز

أ) فحص الحالة

```
kubectl get pods -n kube-system
kubectl describe pod <pod-name> -n kube-system
```

- الأمر `describe` بيوضح الـ Events وأسباب الأعطال (زي ImagePullError أو مشاكل جدولة).

ب) فحص اللوجز

```
kubectl logs <pod-name> -n kube-system
```

- مهم جدًا خصوصًا مع `kube-apiserver`, `etcd`, `scheduler`, `controller-manager`.

⚠️ ملحوظة: فيه أمر قديم اسمه:

Edit Copy bash

```bash
kubectl get componentstatuses
```

لكن بقى **Deprecated**. الأفضل تعتمد على البودات نفسها أو أدوات مراقبة زي Prometheus/Grafana.

فحص مكوّنات الـ Control Plane :

أ) kube-apiserver

- هو نقطة الدخول لكل أوامر kubectl
- لو وقف ← الكلاستر كله بيتأثر.
- افحصه

```
kubectl -n kube-system get pods | grep apiserver
kubectl -n kube-system logs <apiserver-pod>
```

ب) etcd

- بيمثل قاعدة البيانات الأساسية.
- المشاكل فيه = انهيار الكلاستر.
- افحصه:

```
kubectl -n kube-system get pods | grep etcd
kubectl -n kube-system logs <etcd-pod>
```

ج) kube-controller-manager

- مسؤول عن إن الحالة الفعلية = الحالة المطلوبة.
- لو فيه مشكلة ← الـ Pods مش بتتطابق مع العدد المطلوب.
- افحصه

```
kubectl -n kube-system logs <controller-manager-pod>
```

د) Kube- Scheduler

- مسؤول عن توزيع الـ Pods
- لو فيه مشكلة ← هتلاقي Pods في حالة Pending.
- افحصه

```
kubectl -n kube-system logs <scheduler-pod>
```

فحص المكوّنات على مستوى النود :

أ) kubelet

- بيشتغل على كل Node ويراقب الحاويات.
- لو النود مش Ready ← شوف السبب

```
kubectl get nodes
kubectl describe node <node-name>
```

ب) kube-proxy

- مسؤول عن الشبكة والتوجيه.
- مشاكل الشبكة غالبًا مرتبطة بيه.
- افحصه

```
kubectl -n kube-system logs <kube-proxy-pod>
```

أخطاء شائعة :

- Pods Pending : غالبًا مشكلة Scheduler أو موارد النود قليلة.
- إعادة تشغيل متكرر :راجع الـ Logs + Probes + Resource Limits
- مشاكل شبكة :راجع kube-proxy أو الـ CNI (flannel/calico)
- API Server مش بيرد : تحقق من etcd أو الـ Certificates
- etcd فيه مشاكل : راجع مساحة القرص أو البطء في التخزين.

أدوات إضافية للمراقبة :

- **metrics-server** : بيديك أوامر **kubectl top** عشان تشوف الاستهلاك.

```
kubectl top nodes
kubectl top pods -A
```

- **Events**: عشان تعرف ليه حصل خطأ:

```
kubectl get events -A --sort-by=.metadata.creationTimestamp
```

- **Probes**
- **livenessProbe** : لو البود وقف داخليًا ← يعمل **Restart**.
- **readinessProbe** : يحدد لو البود جاهز يستقبل ترافيك ولا لأ.

أوامر سريعة للمراجعة

```
# نظرة عامة
kubectl get nodes
kubectl get pods -A
kubectl get pods -n kube-system

# تشخيص
kubectl describe node <node>
kubectl describe pod <pod> -n kube-system

# لوجز
kubectl logs <pod> -n kube-system

# موارد
kubectl top nodes
kubectl top pods -A

# الأحداث
kubectl get events -A --sort-by=.metadata.creationTimestamp
```

Managing Application Logs

النهاردة، هنتعمق في موضوع مهم جداً، وهو آليات التسجيل (Logging mechanisms) في Kubernetes. عشان نفهم الموضوع بشكل أفضل، خلينا نبدأ الأول بكيفية عمل الـ Logging في Docker.

المقدمة Logging في Docker :

- هانبدأ بمثال عشوائي على Docker Container اسمه event-simulator، وده دوره إنه بيعمل أحداث عشوائية (random events) زي اللي ممكن تحصل على الويب سيرفر. (بكرر ان دا مجرد مثال عشان تفهم بيه الليلة دي)
- الـ Container ده بيعرض كل الأحداث دي على الـ Standard Output .
- لما بتشغل الـ Docker container في الخلفية باستخدام خيار -d (detached mode)، فمش هتشوف الـ Logs مباشرة.
- عشان تشوف الـ Logs دي، بتستخدم أمر docker logs متبوعاً بالـ Container (ID). ولو عايز تشوف الـ Logs لحظة بلحظة (Live trail)، بتستخدم خيار -f (follow).

Logging في Kubernetes

- بعد ما فهمنا الأساس في Docker، بننتقل لـ Kubernetes.
- بتقوم بإنشاء Pod بيستخدم نفس صورة الـ Docker اللي استخدمناها في المثال السابق.
- بمجرد ما الـ Pod يشتغل، يمكنك تشوف الـ Logs بتاعته باستخدام أمر kubectl logs متبوعاً باسم الـ Pod.
- زي ما عملنا مع Docker، يمكنك تستخدم خيار -f عشان تشوف الـ Logs لحظة بلحظة.
- بس خلي بالك ان الـ Logs دي خاصة بالـ Container اللي بيشتغل جوه الـ Pod.

حالة الـ Pods متعددة الـ Containers

- في بعض الأحيان، الـ Pod الواحد ممكن يكون فيه أكتر من Container. في المثال ده، تم تعديل ملف تعريف الـ Pod عشان يضيف Container تاني اسمه image-processor.
- في الحالة دي، لو حاولت تستخدم أمر kubectl logs باسم الـ Pod بس، هيحصل خطأ.
- السبب هو إن Kubernetes مش هيعرف يحدد أي Container فيهم اللي أنت عايز تشوف الـ Logs بتاعته.
- الحل هو إنك لازم تحدد اسم الـ Container بشكل صريح في الأمر.
- يعني هتكتب الأمر بالشكل ده kubectl logs <pod-name> c- <container-name>
- لإحضار لوجات البودات القديمة (بعد Restart) kubectl logs <podname> --previous

---

ملاحظات :

اللوجات مرتبطة بعمر الـ Pod/Container :

- لو البود اتوقف أو اتمسح → اللوجات بتختفي.
- عشان كده لازم تستخدم أدوات مركزية لو عاوز تحتفظ باللوجات لفترة طويلة.

تحديات اللوجات في Kubernetes

- اللوجات بتكون موزعة على كل.Node
- لما البود يتنقل من نود لأخرى أو يتعمله → Restart اللوجات القديمة بتروح.
- صعب تجمع لوجات من آلاف البودات باستخدام kubectl logs بس.

الحل Centralized Logging

- **الأفضل تستخدم حلول مركزية تجمع اللوجات كلها في مكان واحد:**
- EFK Stack (Elasticsearch + Fluentd + Kibana).
- ELK Stack (Elasticsearch + Logstash + Kibana).
- Loki + Promtail + Grafana.

❖ **المبدأ:**

1. الـ Pods تطبع اللوجات لـ stdout/stderr
2. الـ Agent زي Fluentd أو Promtail على كل Node بيجمع اللوجات.
3. ثم يرسلها لسيرفر مركزي زي Elasticsearch أو Loki
4. ممكن تبحث وتفلتر وتعرضها في واجهة زي Kibana أو.Grafana

نصائح عملية

- دايمًا خليك ملتزم إن التطبيق يطبع اللوجات لـ stdout/stderr بدل ما يكتبها في ملفات جوه الحاوية → كده تبقى متوافقة مع Kubernetes.
- راقب حجم اللوجات عشان ما تملأش مساحة النود.
- استخدم أدوات Monitoring مع Logging زي Prometheus + Grafana عشان تشوف الـ Metrics جنب اللوجات.

Rolling Updates and Rollbacks

النهاردة، هنتعمق في موضوع مهم جداً وهو التحديثات (Updates) و التراجع (Rollbacks) في الـ Deploymentsالخاصة بـ Kubernetes.

المقدمة: مفهوم الـ Rollout والـ Revision :

- أول ما تنشئ Deployment جديد .. Kubernetes بيعمل حاجة اسمها Rollout.
- كل Rollout بيتسجل كـ Revision
- أول مرة تعمل Deployment .. ده اسمه Revision 1.
- لو غيّرت نسخة الـ Container (مثلاً من v1 إلى v2) .. يتعمل Revision 2.

الهدف:

- متابعة كل التغييرات اللي اتعملت.
- إمكانية الرجوع لنسخة أقدم بسهولة لو حصل خطأ.

يمكنك رؤية حالة الـ rollout باستخدام الأمر kubectl rollout status متبوعاً باسم الـ Deployment

ولرؤية سجل وتاريخ الـ rollouts ، يمكنك استخدام الأمر kubectl rollout history

---

استراتيجيات النشر(Deployment Strategies)

❖ هناك نوعان رئيسيان من استراتيجيات النشر في Kubernetes :

أ) استراتيجية Recreate

- في هذه الاستراتيجية، إذا كان لديك 5 نسخ (replicas) من التطبيق وتريد تحديثها، فإنك تقوم أولاً بحذف كل النسخ الخمسة القديمة مرة واحدة ، ثم تقوم بإنشاء 5 نسخ جديدة بالنسخة المحدثة من التطبيق.
- المشكلة : كما يمكنك أن تتخيل، خلال الفترة التي تكون فيها النسخ القديمة قد توقفت ولم يتم إنشاء أي نسخ جديدة بعد، فإن التطبيق سيكون متوقفاً وغير متاح للمستخدمين .
- لحسن الحظ، هذه ليست الاستراتيجية الافتراضية في Kubernetes

ب) استراتيجية Rolling Update (التحديث التدريجي)

- هذه هي الاستراتيجية الافتراضية في Kubernetes
- في هذه الاستراتيجية، بدلاً من حذف كل النسخ القديمة دفعة واحدة، يتم حذف نسخة قديمة وإنشاء نسخة جديدة واحدة تلو الأخرى.
- وبكدا يظل التطبيق متاحاً للمستخدمين طوال عملية التحديث وميحصلش Downtime، وتكون العملية سلسة (seamless)

كيفية تحديث الـDeployment

- التحديث يمكن أن يعني عدة أشياء، مثل: تحديث نسخة الـDocker container ، أو تحديث الـ Labels ، أو تحديث عدد النسخ (replicas)
- الطريقة الأولى (الموصى بها) :يمكنك تعديل ملف تعريف الـ Deployment (definition file)

وبعد إجراء التغييرات اللازمة، يمكنك تطبيقها باستخدام الأمر kubectl apply -f deployment.yaml حيث ان هذا الأمر سيقوم بتشغيل rollout جديد، وسيتم إنشاء revision جديد.

- الطريقة الثانية : يمكنك استخدام الأمر kubectl set image لتحديث صورة الـ Container مباشرة. لكن يجب أن تكون حذراً، لأن هذا التغيير لن ينعكس تلقائياً في ملف تعريف الـDeployment ، مما قد يسبب مشاكل في التغييرات المستقبلية.

فهم العملية الداخلية للتحديث

- يمكنك رؤية الفرق بين استراتيجيتي Recreate و Rolling Update بشكل مفصّل باستخدام الأمر kubectl describe deployment
- عند استخدام استراتيجية Recreate ، ستلاحظ في (events) أن Replica Set القديم تم تقليصه (scaled down) إلى صفر نسخة أولاً، ثم تم تكبير Replica Set الجديد إلى العدد المطلوب (مثلاً، 5 نسخ).
- أما عند استخدام استراتيجية Rolling Update ، فسترى أن Replica Set القديم يتم تقليصه نسخة واحدة تلو الأخرى، وفي نفس الوقت، يتم تكبير Replica Set الجديد نسخة واحدة تلو الأخرى.
- عندما تقوم بتحديث التطبيق، فإن Kubernetes يقوم بإنشاء Replica Set جديد تحت الغطاء (under the hood) ويبدأ في نشر الـ Containers الجديدة هناك، بينما يقوم في نفس الوقت بإزالة الـ Pods من Replica Set القديم باتباع استراتيجية Rolling Update
- يمكنك رؤية هذه الـ Replica Sets باستخدام الأمر kubectl get replicasets

---

التراجع (Rollback)

- إذا قمت بتحديث تطبيقك واكتشفت أن النسخة الجديدة بها مشكلة، فإن Kubernetes يسمح لك بالعودة إلى إصدار (revision)سابق.
- للقيام بذلك، استخدم الأمر kubectl rollout undo متبوعاً باسم الـ Deployment
- سيقوم الـ Deployment بحذف الـ Pods الموجودة في Replica Set الجديد وإعادة تشغيل الـ Pods القديمة من Replica Setالسابق.
- ستعود نسخة تطبيقك إلى حالتها السابقة.

ملاحظة هامة : kubectl run vs Deployment :

- قد تكون استخدمت سابقاً أمر kubectl run لإنشاء Pod
- ولكن في الحقيقة، هذا الأمر لا يقوم فقط بإنشاء Pod ، بل يقوم بإنشاء Deployment كاملاً.
- يمكنك ملاحظة ذلك من خلال رسالة الإخراج التي تقول Deployment "nginx" created
- هذا يعتبر طريقة أخرى لإنشاء Deployment فقط بتحديد اسم الصورة، دون الحاجة لملف تعريف. ولكن، يفضل استخدام ملفات التعريف (definition files) لأنها قابلة للحفظ في مستودع الكود وتسهل التعديلات المستقبلية.

ملخص الأوامر

- kubectl create لإنشاء الـ Deployment
- kubectl get deployment لعرض قائمة بالـ Deployments
- kubectl apply و kubectl set image لتحديث الـ Deployments
- Kubectl edit deployment.name لتعديل حاجة معينة في الـ Deployment
- kubectl rollout status لرؤية حالة عملية الـ rollout
- kubectl rollout undo للتراجع عن الـ Deployment

EntryPoint & CMD in Docker

المقدمة

- الموضوع مش مطلوب بشكل رسمي في امتحان Kubernetes CKA/CKAD ، لكن مهم جدًا لأنه بيبان تافه وأغلب الناس بتتجاهله.

أوامر في Docker Containers

- خلينا نفتكر الأول إزاي بيشتغل Command في Docker
- بعد كده هنربطه بالـ Pods في Kubernetes (next Lec.)
- هنركز على:
  - Command
  - Arguments
  - Entrypoint

مثال عمليUbuntu Container :

- لما تعمل docker run ubuntu يعني هايشغل ال container من صورة Ubuntu
  هتلاقي النتيجة : بيتشغّل لحظة وبعدين يخرج (Exit).
- ولو حبيت تتأكد وعملت docker ps -a هتلاقيه في حالة Exited.

ودا يأخدنا لنقطة ليه الـ Container بيخرج؟

- الـ Container مش زي الـ Virtual Machine
- هو معمول إنه يشغّل مهمة واحدة أو Process محدد زي:
  - Web server (NGINX)
  - Database server (MySQL)
  - عملية حسابية أو تحليل بيانات.
- القاعدة:
  - الـ Container يفضل عايش طول ما الـ Process اللي جواه شغال.
  - لو الـ Process وقف → الـ Container يوقف.

مين بيحدد الـ Process الأساسي؟

- في الـ Dockerfile الخاص بأي صورة:
  - فيه أمر اسمه CMD
- الـ CMD هو اللي بيحدد إيه اللي يشتغل جوه الـ Container

أمثلة:

- صورة NGINX → CMD = nginx
- صورة MySQL → CMD = mysqld
- صورة Ubuntu → CMD = bash

مشكلة Ubuntu Image

- Ubuntu الافتراضي بيشغّل bash
- الـ Bash محتاج Terminal عشان يشتغل
- Docker لما يشغّل Container → ما بيركبش Terminal افتراضيًا.
- **النتيجة:**
  - Bash يدوّر على Terminal → مش يلاقي → يخرج.
  - وبالتالي الـ Container يخرج معاه.

تغيير الـ Command مؤقتًا

- ممكن تضيف Command مع docker run عشان تتخطى الـ CMD الأصلي . مثال :

```
docker run ubuntu sleep 5
```

- هنا:
  - Ubuntu Container بيبدأ
  - بدل Bash هابيشغّل sleep 5
  - بينام 5 ثواني وبعدين يخرج.

تغيير الـ Command دائمًا

- لو عاوز الصورة نفسها دائمًا تشغّل sleep 5
  - تبني صورة جديدة من Ubuntu وتضيف فيها CMD جديد.

:Dockerfile

```dockerfile
FROM ubuntu
CMD ["sleep", "5"]
```

- بعد كده:

```bash
docker build -t ubuntu-sleeper .
docker run ubuntu-sleeper
```

- النتيجة: دائمًا ينام 5 ثواني ويخرج.

تمرير Arguments بدل CMD

- لو عايز تغيّر الوقت من 5 إلى 10 ثواني:
  - تكتب: docker run ubuntu-sleeper sleep 10 بس نوعا ما ده شكله مش منطقي لأن اسم الصورة نفسه بيوحي إنه بيعمل "sleep". (مجرد صدفة ومتخدهاش قاعدة)
    وعشان كدا الأفضل انك تكتب docker run ubuntu-sleeper 10 يعني تمرر الرقم بس

هنا ييجي دور ENTRYPOINT

- **ENTRYPOINTبيحدد البرنامج الأساسي اللي لازم يتشغّل.**
- **وأي حاجة تكتبها بعد اسم الـ Container ← تتضاف كـ Arguments للـENTRYPOINT.**

مثال:

- Dockerfile:

```dockerfile
FROM ubuntu
ENTRYPOINT ["sleep"]
```

- لما تشغّل:

```bash
docker run ubuntu-sleeper 10
```

- النتيجة: هيشغّل `sleep 10`.

الفرق بين CMD و ENTRYPOINT

- **CMD**
  - **إذا قمت بتمرير أمر جديد عبر سطر الأوامر مثل docker run ubuntu sleep 5 ، فإن هذا الأمر سيحل محل (overrides) الأمر الافتراضي الذي تم تعريفه في CMD داخل الصورة**
- **ENTRYPOINT**
  - **إذا قمت بتعريف ENTRYPOINT في الـ Dockerfile، فإن الأمر الذي تمرره من سطر الأوامر سيُضاف (appended) إلى الـ ENTRYPOINT بدلاً من أن يحل محله.**

مثال:

- **إذا كانت الـ Dockerfile تحتوي على ENTRYPOINT ["sleep"]**
- **وقمت بتشغيل الأمرdocker run ubuntu-sleeper 10**
- **فإن الأمر النهائي الذي سيتم تنفيذه هو sleep 10**

الجمع بين CMD + ENTRYPOINT

- لو عايز تحط **قيمة افتراضية** (Default) للـ Arguments

مثال:

```dockerfile
FROM ubuntu
ENTRYPOINT ["sleep"]
CMD ["5"]
```

- النتيجة:
  - لو مش كتبت حاجة → يعمل `sleep 5`.
  - لو كتبت `docker run ubuntu-sleeper 20` → يعمل `sleep 20`.

⚠️ شرط: جب دائمًا استخدام صيغة الـ JSON array عند تحديد ENTRYPOINT و CMD لضمان أن الأمر والوسائط يتم التعامل معها بشكل صحيح ومنفصل.

تعديل ENTRYPOINT وقت التشغيل

- لو حبيت تغيّر الـ ENTRYPOINT نفسه في وقت التشغيل:
  - تستخدم `docker run --entrypoint sleep2.0 ubuntu-sleeper 10`
- النتيجة : بدل ما يستخدم"sleep" ، هيشغّل "sleep2.0 10"

الخلاصة

- الـ CMD بيحدد البرنامج الافتراضي اللي يشتغل.
- الـ ENTRYPOINT بيحدد البرنامج الأساسي، والـ Arguments اللي تكتبها بتضاف له.
- الجمع بينهم بيخليك تحط قيمة افتراضية وتتخطاها لو عايز.
- تقدر كمان تغيّر الـ ENTRYPOINT في وقت التشغيل لو محتاج.

Commands & Args in K8S

المقدمة

- في المحاضرة اللي فاتت اتكلمنا عن CMD و ENTRYPOINT في Docker
- دلوقتي هنشوف إزاي نفس الفكرة بتتطبق على Pods في Kubernetes
- الهدف: نفهم إزاي نحدد الأوامر والمعاملات (Commands & Args) جوه تعريف الـ Pod

---

مراجعة سريعة منDocker

- عملنا Image اسمه ubuntu-sleeper
- الـ Image ده بيستخدم ENTRYPOINT = sleep + CMD = 5
- يعني:
  - لو شغّلت docker run ubuntu-sleeper هينام 5 ثواني ويخرج.
  - لو شغّلت docker run ubuntu-sleeper 10 هينام 10 ثواني ويخرج.

ربط الفكرة بالـ Pod

- لو عايز تعمل Pod من الصورة دي:
  - تبدأ بملف Pod Definition.
  - تكتب الاسم + تكتب اسم الصورة:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: ubuntu-sleeper-pod
spec:
  containers:
- name: ubuntu-sleeper
  image: ubuntu-sleeper
```

- النتيجة: الـ Pod بيشغّل Container ينام 5 ثواني (القيمة الافتراضية من CMD).

تغيير ال CMD (المكافئ لها في الـ POD هو args)

- و عايز تخليه ينام 10 ثواني بدل 5:
- في Docker كنا بنكتب: docker run ubuntu-sleeper 10
- لكن في Kubernetes بتكتب المعاملات (Arguments) في خاصية اسمها args

```yaml
spec:
  containers:
  - name: ubuntu-sleeper
    image: ubuntu-sleeper
    args: ["10"]
```

❖ يعني بإختصار ممكن تعتبر ان الـ args في Pod Definition تعادل CMD اللي موجود في Dockerfile

تغيير ال ENTRYPOINT (المكافي لها في الـ POD هو COMMAND)

- في Docker كنا نستخدم: **docker run --entrypoint sleep2.0 ubuntu-sleeper 10**
- لكن في Kubernetes نستخدم command داخل Pod Definition

```yaml
spec:
  containers:
  - name: ubuntu-sleeper
    image: ubuntu-sleeper
    command: ["sleep2.0"]
    args: ["10"]
```

❖ **يعني بإختصار ممكن تعتبر ان الـ Command في Pod Definition تعادل EntryPoint اللي موجود في Dockerfile**

Configure Environment Variables in Applications

المقدمة

- الهدف من المحاضرة: إزاي نضيف Environment Variables جوه Container في Kubernetes باستخدام Pod Definition File
- الفكرة قريبة جدًا من اللي كنا بنعمله في Docker باستخدام -e أو --env

---

الطريقة الأولى إضافة Environment Variable مباشرة كـ (Key-Value)

- جوه ملف الـ Pod Definition نستخدم خاصية اسمها env
- ولاحظ ان الـ env بتكون عبارة عن Array
- يعني كل متغير بيبدأ بشرطة - ويتكتب فيه name و value

مثال:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mycontainer
    image: ubuntu
    command: ["sleep", "3600"]
    env:
    - name: APP_MODE
      value: "production"
    - name: APP_VERSION
      value: "1.0"
```

OR

```
docker run -e APP_COLOR=pink simple-webapp-color
```

pod-definition.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: simple-webapp-color
spec:
  containers:
  - name: simple-webapp-color
    image: simple-webapp-color
    ports:
      - containerPort: 8080

    env:
      - name: APP_COLOR
        value: pink
```

📌 هنا:

- المتغير APP_MODE قيمته "production"
- المتغير APP_VERSION قيمته "1.0"
- دول بيبقوا متاحين جوه الـ Container زي أي var env عادي.

الطريقة التانية ConfigMaps و Secrets (هايتشرحوا بالتفصيل قدام)

- بدل ما تكتب القيمة بشكل مباشر(hard-coded value) ، ممكن تجيب القيمة من ConfigMap أو Secret
- في الحالة دي بدل ما تكتب : value هاتكتب valueFrom:
  يعني هاستدعي القيمة سواء من ConfigMapKeyRef or SecretKeyRef

❖ مثال مبسط (ConfigMap) :

```yaml
env:
- name: APP_COLOR
  valueFrom:
    configMapKeyRef:
      name: app-config
      key: color
```

📌 هنا:

- بدل ما أحط القيمة يدويًا، أنا بجيبها من ConfigMap اسمه app-config.
- وبالتحديد من المفتاح color.

الفرق بين الطريقتين

- **Direct Key-Value : تكتب القيم مباشرة في ملف الـ Pod سهل وسريع لكن مش مرن .**
- **ConfigMaps/Secrets : تخزن القيم في موارد منفصلة وتستدعيها وده مفيد عشان:**
  - **تقدر تغيّر القيم من غير ما تعدل ملف الـ Pod**
  - **تفصل الـ Configuration عن الكود.**
  - **تخلي الـ Secrets (زي الباسورد) مخزنة بأمان.**

Configuring ConfigMaps in Applications

◆ المقدمة

- في المحاضرة اللي فاتت شُفنا إزاي نحدد Environment Variables في ملف تعريف الـ Pod
- لكن لما يبقى عندك عدد كبير من ملفات الـ Pod ، هيبقى صعب جدًا تدير كل الـ environment data جوه الملفات نفسها.
- الحل: نستخدم ConfigMaps علشان نطلع الـ configuration من ملفات الـ Pod ونديرها مركزيًا.

◆ ما هي الـ ConfigMaps ؟

- أداة في Kubernetes بتُستخدم علشان تمرر بيانات التهيئة (Configuration Data) في شكل Key-Value pairs
- لما بيتعمل Pod ، الـ ConfigMap بيتحقن (Injected) جوه الـ Pod
- النتيجة: القيم دي بتتوفر كـ Environment Variables للتطبيق اللي شغال جوه الـ Container

---

◆ خطوات العمل مع ConfigMaps

فيه مرحلتين أساسيتين:

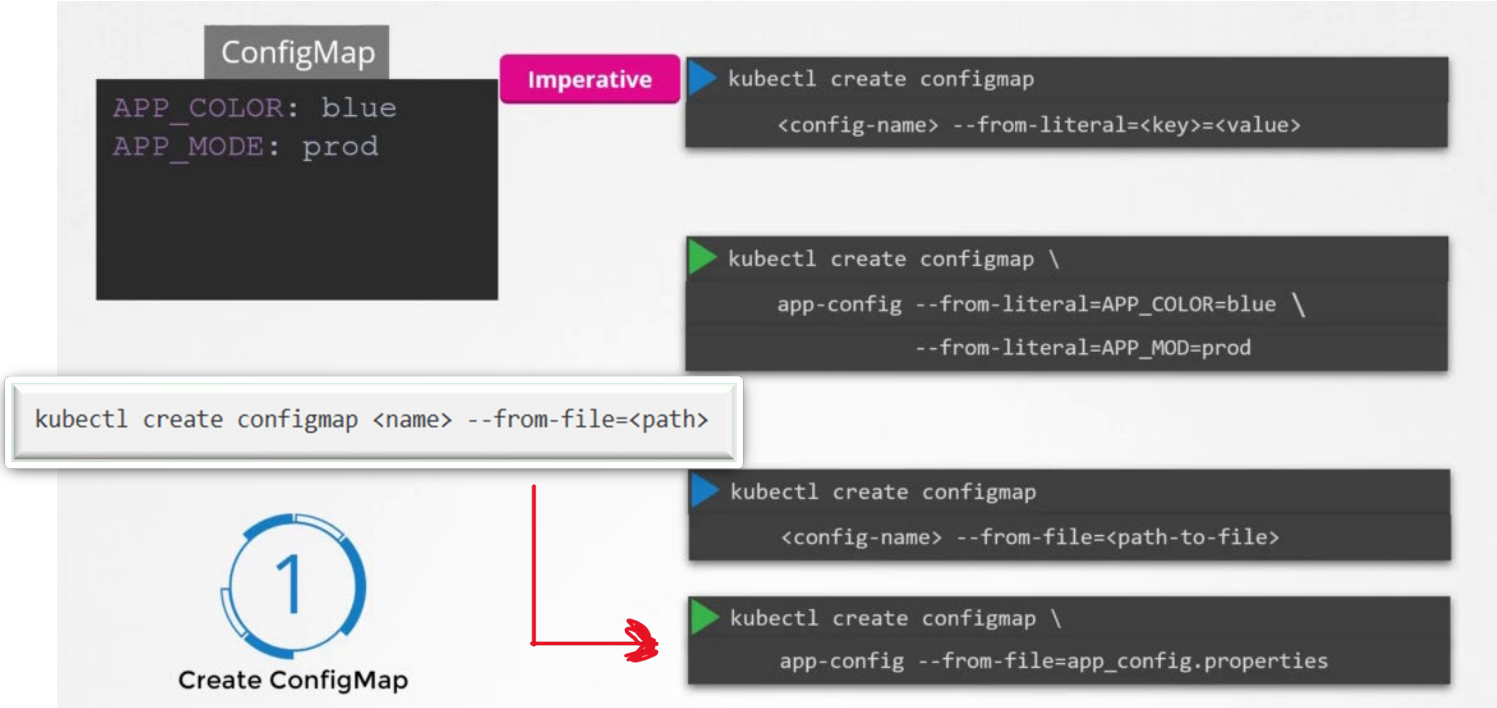
1. إنشاء الـ ConfigMap
2. حقنه (Inject) في الـ Pod

◆ طرق إنشاء ConfigMap

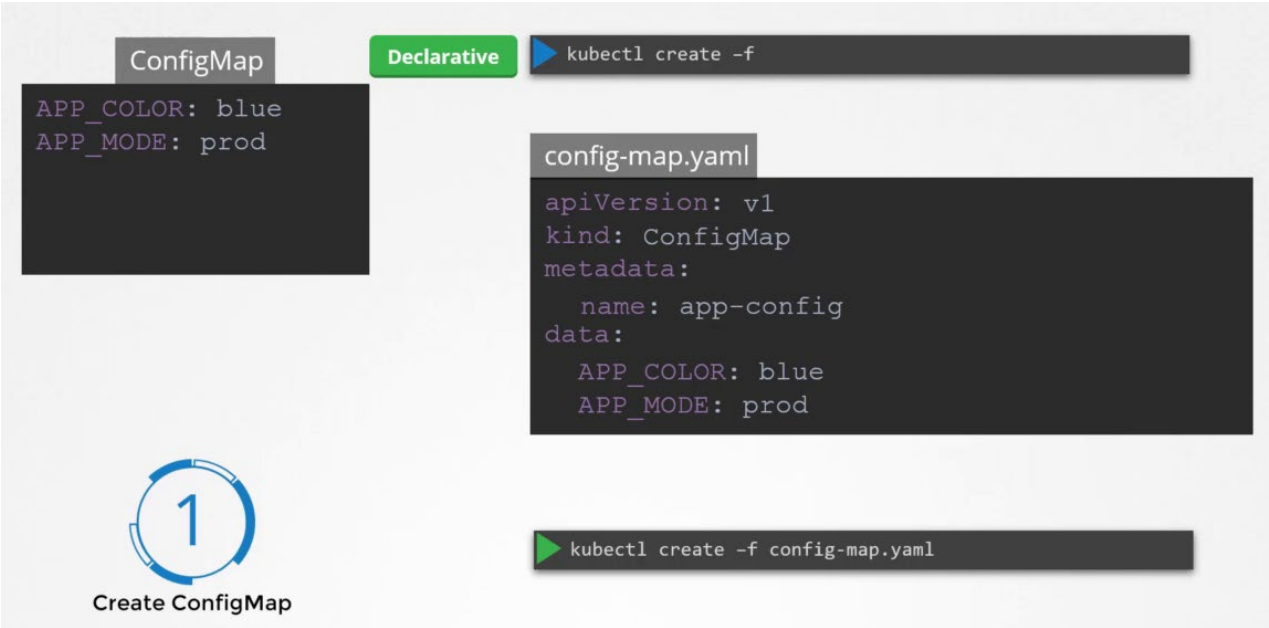
زي أي Object في Kubernetes ، عندك طريقتين:

1. "Imperative" (يعني أوامر مباشرة بدون definitionfile.yaml)

- باستخدام الأمر:
- لو عاوز تضيف أكتر من قيمة : تكرر from-literal -- كذا مرة.
- العيب: بيبقى معقد لو عندك قيم كتير.



2. " Declarative " (يعني من خلال ملف yaml)

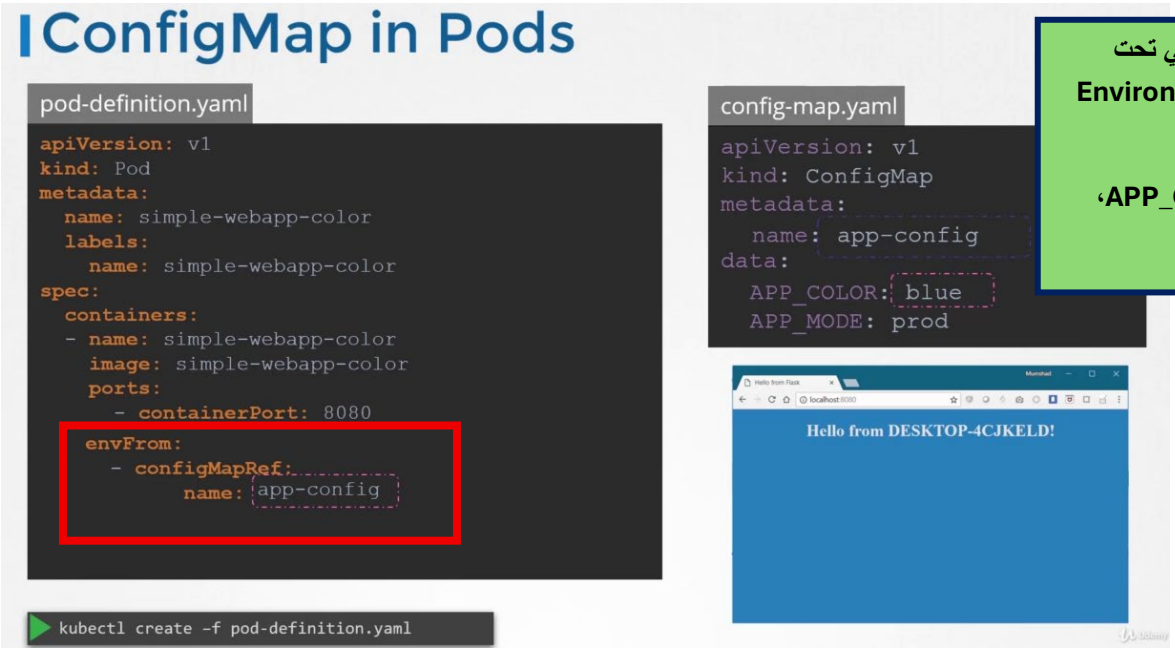


◆ إدارة الـ ConfigMaps

- ممكن تعمل أكتر من ConfigMap (مثال: واحد للتطبيق، واحد للـMySQL ، واحد للـ Redis)
- مهم جدًا تسميهم صح علشان هتحتاج تستدعيهم بعدين.
- لعرضهم:
  - kubectl get configmaps
- للتفاصيل :
  - kubectl describe configmap app-config

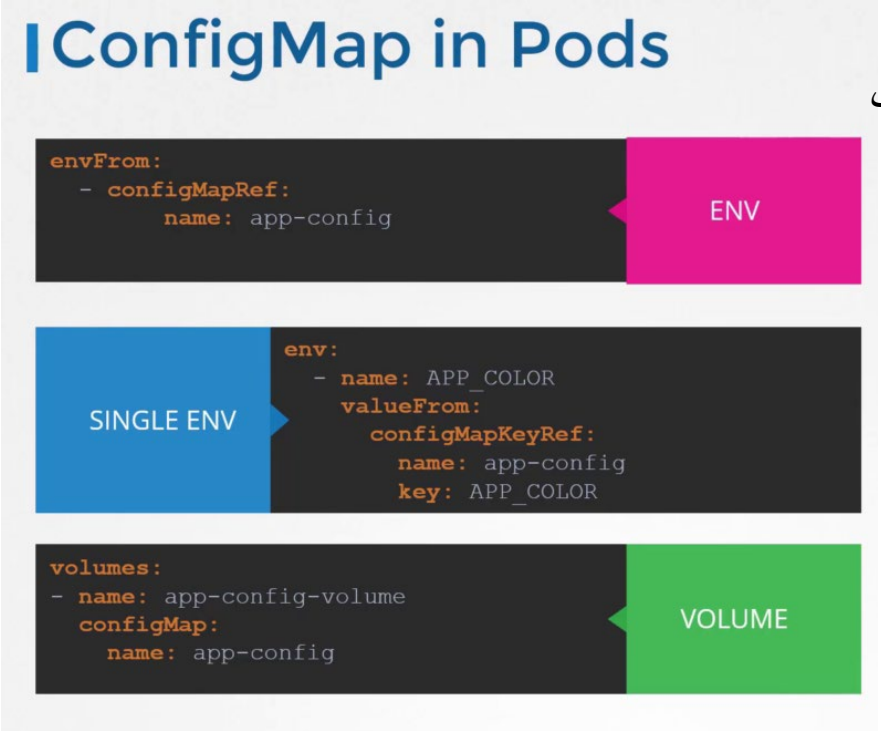
◆ حقن ConfigMap في Pod (ثلاث طرق) :

1. عندك Pod عادي، وعاوز تحقن فيه ConfigMap هاتيجي في ملف تعريف الـ Pod ، جوه الـ container ، بتضيف:



كده أي قيمة في app-config اللي تحت ConfigMap هاتتبعت كـ Environment Variable للـContainer

مثال: لو كان فيه APP_COLOR=blue، التطبيق هيشتغل بخلفية زرقاء.



2. كـ Environment Variables زي ما شرحنا فوق
3. كملف في Volume حيث ان
القيم تتحول لملفات جوه الـ Container

◆ الخلاصة

- ConfigMap بتفصل بين الكود وبيانات الإعدادات.
- سهلة الإدارة وإعادة الاستخدام.
- عندك 3 طرق إنشـاء:
  - من أمر مباشر (imperative)
  - من ملف (declarative)
  - من ملف بيانات خارجي.
- ليها أكتر من طريقة للحقن Env Variables أو Volumes

أمثلة :

1 إنشاء ConfigMap من ملف YAML

ملف: app-config.yaml

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  APP_COLOR: blue
  APP_MODE: production
```

إنشاء الـ ConfigMap:

```bash
kubectl apply -f app-config.yaml
```

عرضه:

```bash
kubectl get configmaps
kubectl describe configmap app-config
```

2 استخدام ConfigMap في Pod كـ Environment Variables

ملف: **pod-with-configmap.yaml**

Edit Copy yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
spec:
  containers:
  - name: myapp-container
    image: nginx
    envFrom:
    - configMapRef:
        name: app-config
```

نشرح:

- `envFrom` → معناه حقن كل القيم من الـ `app-config` ConfigMap.
- النتيجة: الـ Container هيشوف متغيرات:
  - `APP_COLOR=blue`
  - `APP_MODE=production`

تشغيل:

Edit Copy bash

```bash
kubectl apply -f pod-with-configmap.yaml
```

تجربة جوه الـ Container:

Edit Copy bash

```bash
kubectl exec -it myapp-pod -- env | grep APP_
```

3 استخدام ConfigMap كـ ملف Volume

ملف: pod-with-configmap-volume.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod-volume
spec:
  containers:
  - name: myapp-container
    image: nginx
    volumeMounts:
    - name: config-volume
      mountPath: /etc/config
  volumes:
  - name: config-volume
    configMap:
      name: app-config
```

نشرح:

- هنا القيم في `app-config` ConfigMap هتتبعت كـ **ملفات** جوه الـ Container.
- الملف `etc/config/APP_COLOR/` هيحتوي `blue`.
- الملف `etc/config/APP_MODE/` هيحتوي `production`.

تجربة:

```bash
kubectl exec -it myapp-pod-volume -- cat /etc/config/APP_COLOR
```

4 إنشاء ConfigMap Imperative (من CLI مباشرة)

```bash
kubectl create configmap app-config \
  --from-literal=APP_COLOR=red \
  --from-literal=APP_MODE=dev
```

◆ الخلاصة

- **عندك 3 طرق أساسية تربط ConfigMap بـ Pod**
  1. **envFrom : يحقن كل القيم كـ Environment Variables**
  2. **env : تختار قيم محددة من ConfigMap**
  3. **volumes تحقن القيم كملفات جوه الـ Container**

Configure Secrets in Applications

◆ المقدمة

- عندنا تطبيق ويب مكتوب بـ Python بيتصل بقاعدة بيانات MySQL
- الكود فيه الـ hostname + username + password مكتوبين بشكل hardcoded (مكتوبين جوه الكود مباشرة)
- دي مشكلة كبيرة لأنه خطر أمني.

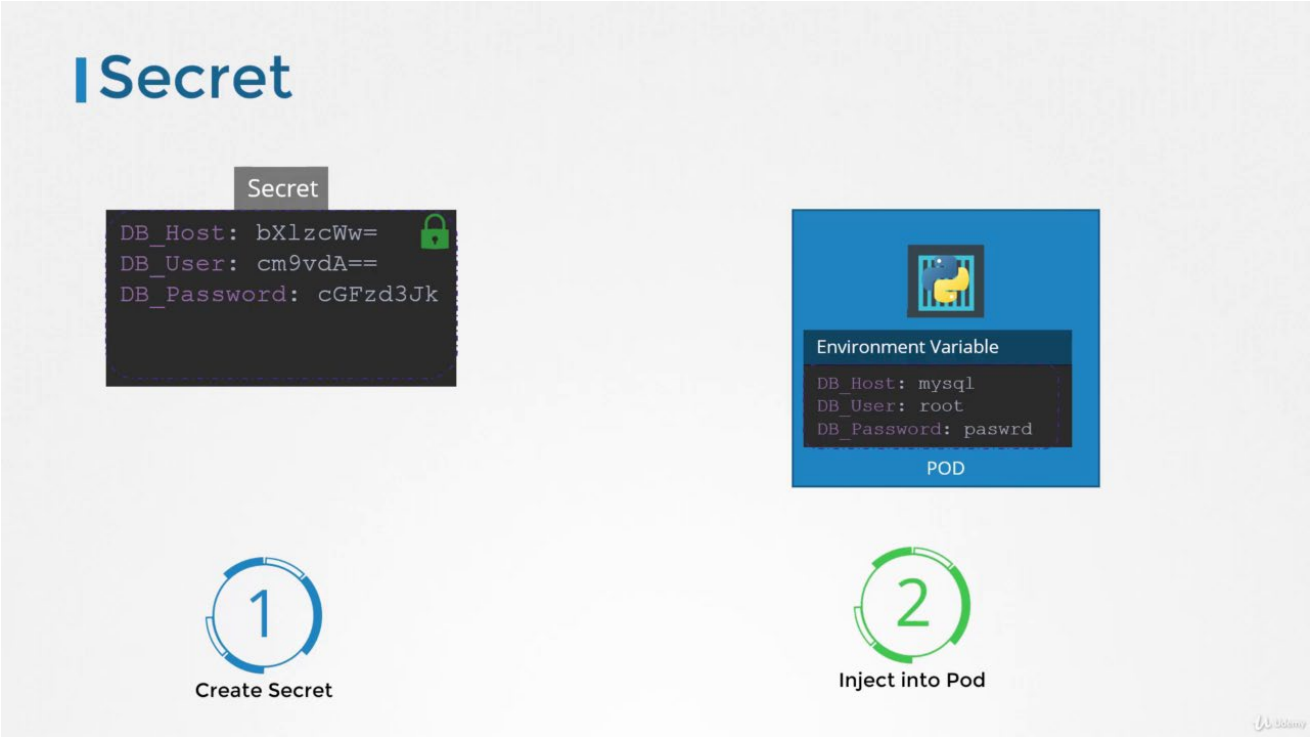
◆ المشكلة

- لو سيبنا بيانات حساسة زي Passwords أو Keys جوه الكود يبقي أي حد يقدر يوصلها.
- في المحاضرة اللي فاتت شُفنا إننا ممكن ننقل القيم دي لـ ConfigMap
- لكن:
  - الـ ConfigMap بيخزن البيانات كـ Plain text وبالتالي ينفع نخزن فيه حاجات عادية زي Hostname أو Username
  - لكن مش آمن نخزن فيه Passwords أو مفاتيح.

◆ الحل Secrets

- Secret هو نوع خاص من Kubernetes Object بيُستخدم لتخزين بيانات حساسة:
  - Passwords
  - Tokens
  - Keys (SSH, TLS, ...)
- يشبه ConfigMap ، لكن الاختلاف الأساسي:
  - البيانات بتتخزن بشكل Encoded (base64) علشان تبقى محمية أكتر. (فيه ملاحظة مجنونة جدا عالحوار ده قدام)

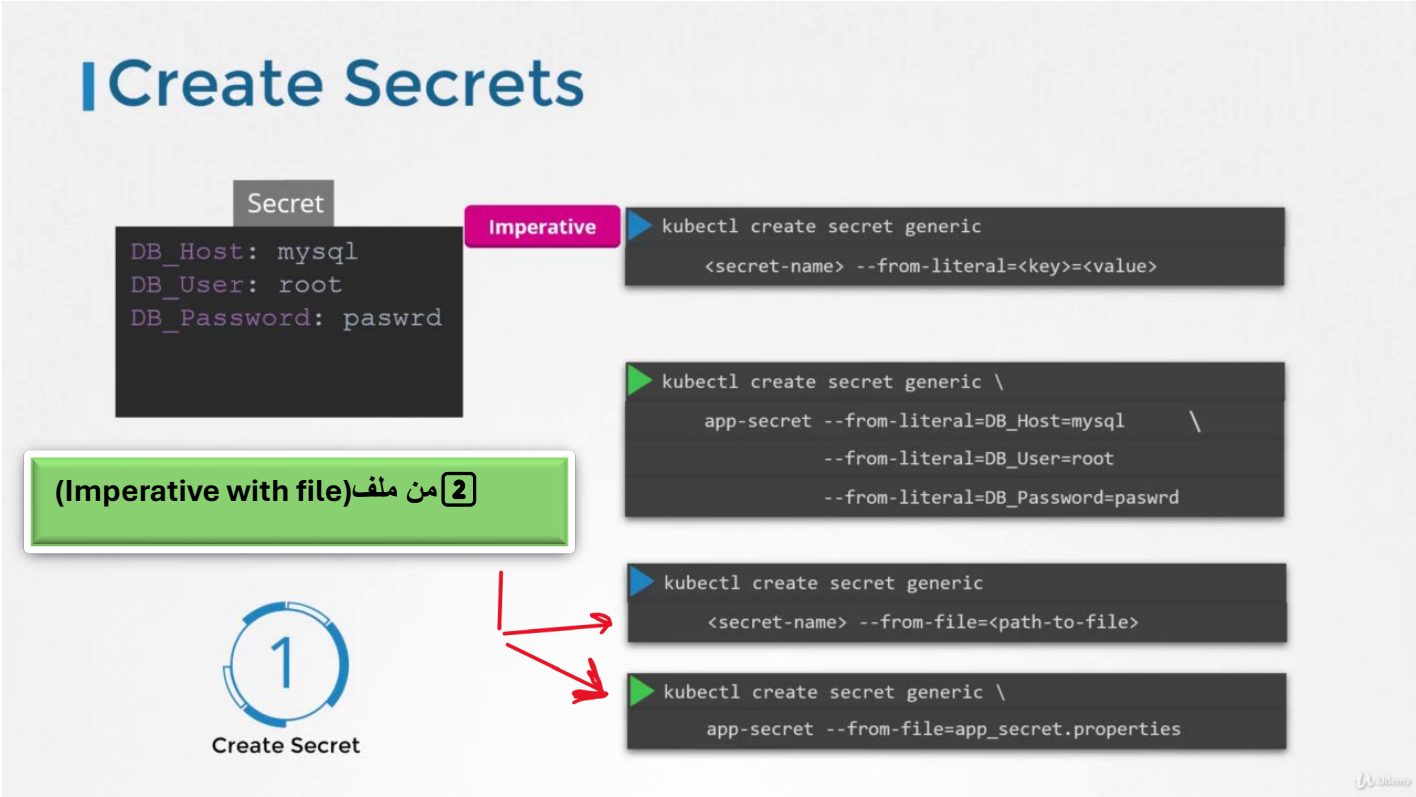
◆ خطوات استخدام Secrets



◆ طرق إنشاء Secrets

زي ConfigMap ، عندنا طريقتين:

1 Imperative (CLI)



- هنا:
- app-secret = اسم الـ Secret
- --from-literal = نحدد القيم Key-Value
- ممكن تكرر --from-literal أكتر من مرة لإضافة قيم جديدة.
- العيب: لو عندك Secrets كتيرة هيبقى معقد.

---

3 Declarative (YAML file)

- نكتب ملف زي ConfigMap ، لكن النوع Secret

Encode Secrets

Secret

```
DB_Host: mysql
DB_User: root
DB_Password: paswrd
```

Declarative

```
kubectl create -f
```

secret-data.yaml

```
DB_Host: bXlzcWw=
DB_User: cm9vdA==
DB_Password: cGFzd3Jk
```

```
apiVe
kind:
metac
  nar
data:
  DB_Host: mysql
  DB_User: root
  DB_Password: paswrd
```

1

Create Secret

```
kubectl create -f secret-data.yaml
```

زي ماتوقعت ياعزيزي برافو عليك .. داتا زي دي لازم تكون متشفرة ومعمولها Base64 Encoded (وخلي بالك ان ده له علاقة بالملاحظة المجنونة جدا اللي هاقولهالك قدام)

طاب بردو ازاي هاتعمل encoded و decoded

Encode Secrets

```
DB_Host: mysql
DB_User: root
DB_Password: paswrd
```

```
DB_Host: bXlzcWw=
DB_User: cm9vdA==
DB_Password: cGFzd3Jk
```

```
echo -n 'mysql' | base64
bXlzcWw=
```

```
echo -n 'root' | base64
cm9vdA==
```

```
echo -n 'paswrd' | base64
cGFzd3Jk
```

Decode Secrets

```
echo -n 'bXlzcWw=' | base64 --decode
mysql
```

```
echo -n 'cm9vdA==' | base64 --decode
root
```

```
echo -n 'cGFzd3Jk' | base64 --decode
paswrd
```

◆ إدارة Secrets :

- عرض Secrets: **kubectl get secrets**
- وصف Secret: **kubectl describe secret app-secret** (بيظهر الخصائص لكن مش القيم نفسها)
- لو عايز تشوف القيم (مشفرة): **kubectl get secret app-secret -o yaml**

```
kubectl get secrets
NAME                  TYPE                                  DATA      AGE
app-secret            Opaque                                3         10m
default-token-mvtkv   kubernetes.io/service-account-token   3         2h
```

```
kubectl describe secrets
Name:         app-secret
Namespace:    default
Labels:       <none>
Annotations:  <none>

Type:  Opaque

Data
====
DB_Host:      10 bytes
DB_Password:  6 bytes
DB_User:      4 bytes
```

```
kubectl get secret app-secret -o yaml
apiVersion: v1
data:
  DB_Host: bXlzcWw=
  DB_Password: cGFzd3Jk
  DB_User: cm9vdA==
kind: Secret
metadata:
  creationTimestamp: 2018-10-18T10:01:12Z
  labels:
    name: app-secret
  name: app-secret
  namespace: default
uid: be96e989-d2bc-11e8-a545-080027931072
type: Opaque
```

◆ حقن Secrets في Pod

1 كـ Environment Variables

pod-definition.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: simple-webapp-color
  labels:
    name: simple-webapp-color
spec:
  containers:
  - name: simple-webapp-color
    image: simple-webapp-color
    ports:
      - containerPort: 8080
    envFrom:
      - secretRef:
          name: app-secret
```

secret-data.yaml

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: app-secret
data:
  DB_Host: bXlzcWw=
  DB_User: cm9vdA==
  DB_Password: cGFzd3Jk
```

2 Inject into Pod

```
kubectl create -f pod-definition.yaml
```

- هنا: كل القيم في app-secret هتتبعت كـ Env Variables
  - DB_Host=mysql
  - DB_User=admin
  - DB_Password=paswrd

2 كـ Volume (ملفات)

Edit Copy

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod-volume
spec:
  containers:
  - name: myapp-container
    image: python-app
    volumeMounts:
    - name: secret-volume
      mountPath: /etc/secret
  volumes:
  - name: secret-volume
    secret:
      secretName: app-secret
```

ENV

```yaml
envFrom:
  - secretRef:
      name: app-config
```

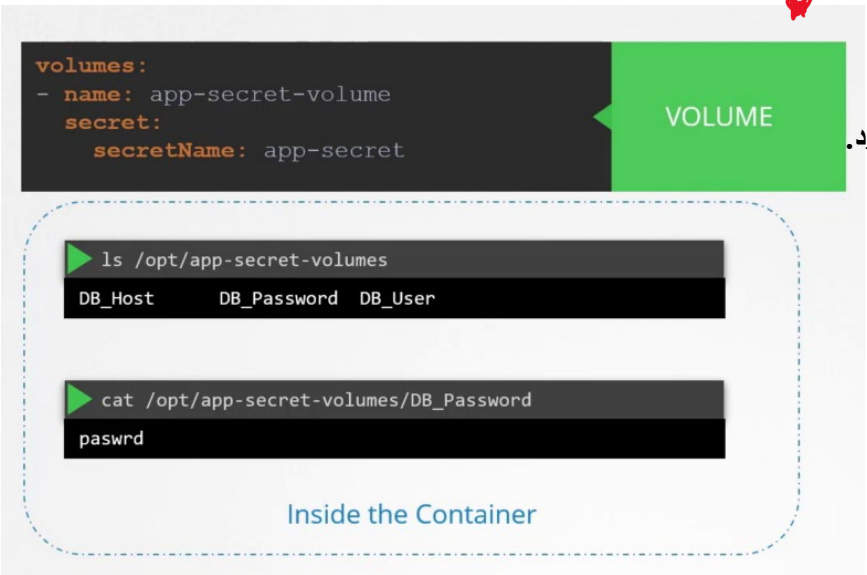
SINGLE ENV

```yaml
env:
  - name: DB_Password
    valueFrom:
      secretKeyRef:
        name: app-secret
        key: DB_Password
```

VOLUME

```yaml
volumes:
- name: app-secret-volume
  secret:
    secretName: app-secret
```

- هنا كل Key في Secret بيتحول لملف جوه /etc/secret
  - /etc/secret/DB_Host هيحتوي "mysql"
  - /etc/secret/DB_Password هيحتوي الباسورد.



📊 مقارنة بين ConfigMap و Secret في Kubernetes

العنصر	ConfigMap	Secret
📌 الاستخدام	لتخزين بيانات **غير حساسة** (إعدادات – Configurations) مثل: APP_COLOR=blue	لتخزين بيانات **حساسة** مثل: Passwords, Tokens, Certificates
🔐 التخزين	Plain text (نص عادي غير مشفر)	مشفر باستخدام Base64 (Encoding)
🗄️ المكان	etcd (قواعد بيانات الـ Kubernetes)	etcd (لكن مشفر Base64)
⚠️ الأمان	غير آمن لو فيه Passwords	آمن نسبيًا، مخصص للبيانات الحساسة
🛠️ إنشاؤه	`kubectl create configmap` أو YAML	`kubectl create secret` أو YAML
🧩 حقنه في Pod	- كـ Env Variables - كـ Volume (ملفات)	- كـ Env Variables - كـ Volume (ملفات)
🗂️ شكل البيانات	key-value (مثال: APP_COLOR=blue)	key-value لكن القيم لازم تكون Base64 (مثال: admin → YWRtaW4=)
🔎 عرض البيانات	`kubectl get configmap -o yaml` → القيم بتظهر واضحة	`kubectl get secret -o yaml` → القيم بتظهر مشفرة
🔑 أمثلة	DB_Host=mysql, APP_MODE=prod	DB_Password=12345, TLS_KEY=..., ...=API_TOKEN

الملاحظة المجنونة جدا اللي يفضل تكون عارفها

الـ Secrets في Kubernetes بتخزن البيانات الحساسة زي الباسوردات أو الـ tokens في صيغة base64

- **يعني لو عندك Secret فيه كلمة سر admin123، هيتم تخزينها كـ YWRtaW4xMjM وبعد ماتطبق Base64**
- **المشكلة :أي حد معاه الـ Base64 ده يقدر يفك تشفيره بسهولة ويشوف الباسورد الأصلي.**

وبالتالي هل Secrets آمنة فعلًا؟

- **الوثائق الرسمية (Documentation) والمدونات بتقول إن Secrets أأمن من تخزين البيانات كنص عادي (plain text)**
- **✅ السبب: بتقلل من خطر إنك تعرض الباسورد بالغلط مثلًا في YAML أو في ملفات الـ Pod مباشرة.**
- **❌ لكن: الـ Secrets نفسها مش مشفرة، فهي مش "آمنة" بمعنى التشفير الكامل.**
- **اللي بيخليها أأمن فعلًا هو الـ Best Practices**

الـ Best Practice مع الـ Secrets

- ما تحطش ملفات الـ Secrets في الـ Git أو أي Repository عشان متبقاش مكشوفة.
- شغّل Encryption at Rest في Kubernetes لأن ده بيخلي البيانات متخزنة مشفرة في قاعدة بيانات etcd
- اعتمد على طريقة Kubernetes في التعامل مع الـ Secrets :
  1. الـ Secret مش بيتبعت للـ Node إلا لو فيه Pod محتاجه.
  2. الـ Kubelet المسؤول عن تشغيل الـ Pods بيخزن الـ Secret في tmpfs (ذاكرة مؤقتة) .. يعني مش بنخزنه على الهارد ديسك.
  3. لما الـ Pod يتشال، الـ Kubelet بيمسح أي نسخة محلية للـ Secret

---

المخاطر والحلول المتقدمة

- بالرغم من إن الـ Secrets أحسن من الـ ConfigMap ، لكنها مش الحل الأقوى لتأمين كلمات السر.
- فيه أدوات متقدمة وأأمن ممكن تستخدمها:
  - Helm Secrets بتشفر الـ Secrets جوه ملفات Helm Charts
  - HashiCorp Vault نظام متخصص في (secrets management system) حيث بيأمّنها بشكل كامل.

🔑 الخلاصة

- الـ Secrets مش مشفرة افتراضيًا، بس أكثر أمانًا من كتابة الباسورد في ملفاتك كـ نص عادي.
- الأمان الحقيقي بيعتمد على أفضل الممارسات زي التشفير عند التخزين (Encryption at Rest) وعدم رفع ملفات Secrets في الـ Repos
- لو عايز مستوى أعلى من الأمان فالأفضل استخدم أدوات متخصصة زي Vault

Multi Container PODs & Init Containers

المقدمة

- في البداية بيقول إننا في القسم ده هنتكلم عن Multi-Container Pods البود اللي فيه أكتر من Container
- الفكرة الأساسية جاية من تصميم التطبيقات الحديثة:
  - بدل ما يبقى عندنا برنامج ضخم واحد Monolithic Application (كل حاجة فيه متجمعة)
  - بنقسمه لجزء صغير مستقل اسمه Microservice

ليه الـ Microservices مهمة؟

- كل Microservice بيكون كود صغير مستقل وبالتالي أسهل في التطوير والتعديل.
- نقدر نعمله Deploy بشكل منفصل من غير ما نأثر على باقي الأجزاء.
- كمان بيخلينا نقدر نعمل Scaling (زيادة أو تقليل عدد النسخ) لكل جزء لوحده بدل ما نعدل البرنامج كله.

في بعض الحالات محتاجين خدمتين يشتغلوا مع بعض

- Web Server (الخدمة الرئيسية) .
- Logging Service / Agent (خدمة تسجيل Logs)
- محتاجين كل Web Server يكون معاه نسخة من Logging Agent علشان يسجل نشاطه.

🛑 المشكلة: مش عايز أدمج الكود بتاع الاتنين في برنامج واحد لأن كل واحد ليه وظيفة مختلفة ومستقل.

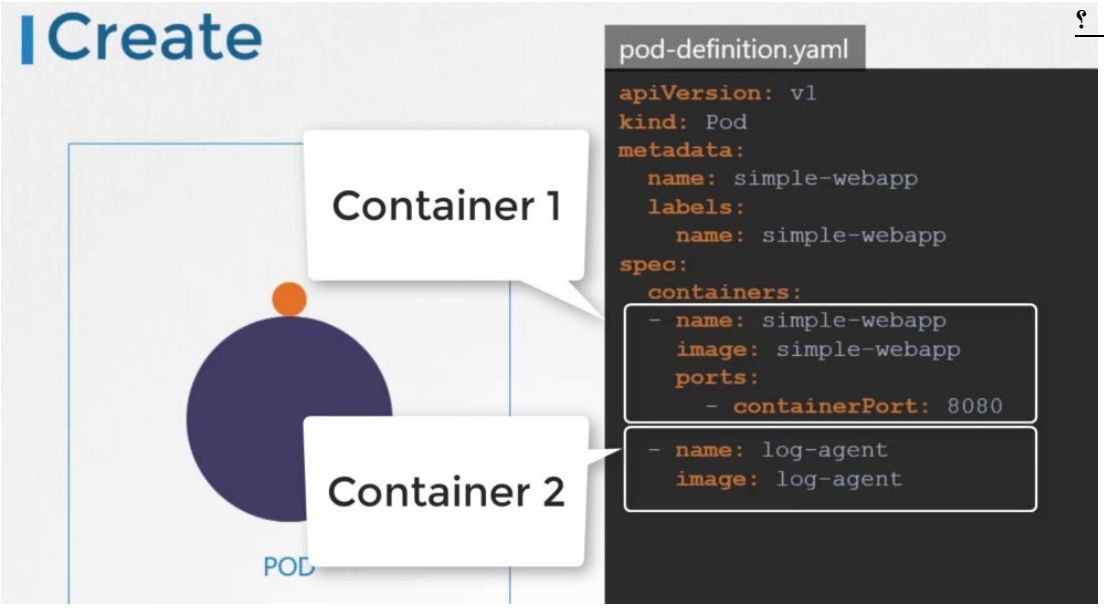
✅ الحل: أخليهم يتنشروا بشكل منفصل لكن يشتغلوا مع بعض .. طاب ازاي ؟؟؟؟!

هنا بييجي دور الـ Multi-Container Pods

- بدل ما أخلي كل خدمة في Pod مختلف، ممكن أحط أكتر من Container جوه Pod واحد.
- الميزة إنهم بيشاركوا حاجات مهمة:
  1. نفس دورة الحياة يعني يتخلقوا سوا ويتشالوا سوا.
  2. نفس الـ Network Space يقدروا يتكلموا مع بعض باستخدام localhost
  3. نفس الـ Volumes يعني يشاركوا الملفات / البيانات المخزنة في الـ Storage

➡️ ده معناه إني مش محتاج أعمل Service أو إعدادات معقدة علشان أخليهم يتواصلوا.

إزاي أكتب Multi-Container Pod ؟



الخلاصة

- Multi-Container Pods مفيدة لما يكون عندنا خدمتين محتاجين يشتغلوا مع بعض زي Web Server + Logger
- كل الحاويات (Containers) جوه نفس الـ Pod :
  - ليهم نفس الـ Lifecycle
  - بيتشاركوا في الشبكة (Network)
  - بيتشاركوا في التخزين (Volumes)
- بيتكتبوا في YAML تحت قسم containers لأنه Array

شرح: Init Containers في Kubernetes

1 الفكرة الأساسية

- في أي Pod فيه أكتر من Container زي ما شرحنا قبل كده عن multi-container pods
  - كل Container لازم يكون عنده process (عملية) شغالة طول ما البود عايش.
  - مثلًا:
    - عندك Web Application شغال.
    - ومعاه Log Agent بيسجل Logs
  - الاتنين لازم يفضلوا شغالين مع بعض لأن لو واحد فيهم وقع فالبود كله بيعمل Restart

---

2 المشكلة

- ساعات بيكون عندك process محتاج يتنفذ مرة واحدة بس عند بداية البود.
  - مثال 1 Container: يسحب كود أو ملفات من Repository (Git) قبل ما التطبيق يبدأ.
  - مثال 2 Container: يستنى لحد ما Database أو Service تانية تبقى جاهزة قبل ما التطبيق يشتغل.

👉 هنا مينفعش تخلي الـ Container الأساسي هو اللي يعمل ده، لأن وظيفته الرئيسية حاجة تانية.

---

3 الحل Init Containers

- دي Containers خاصة بتتكتب جوه Pod
- بتتحدد في قسم اسمه:
- initContainers:
- الفرق بينهم وبين الـ Containers العادية:
  - لازم يخلصوا تنفيذ العملية (run to completion)
  - مش بيشتغلوا طول الوقت.
  - لازم ينجحوا الأول قبل ما يبدأ أي Container أساسي في البود.

4️⃣ مثال 1: Container يسحب Repository

Edit Copy yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: busybox:1.28
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']

  initContainers:
  - name: init-myservice
    image: busybox
    command: ['sh', '-c', 'git clone <some-repository-that-will-be-used-by-application> ; done;']
```

🔎 **الشرح:**

- **البود عنده Container رئيسي (myapp-container) شغال رسالة + بينام ساعة.**
- **فيه initContainer اسمه init-myservice :**
  - **بيستخدم صورة busybox**
  - **بينفذ أمر git clone علشان يسحب الكود اللي التطبيق هيحتاجه.**
- **النتيجة: مش هيبدأ التطبيق غير لما العملية دي تخلص بنجاح.**

5️⃣ مثال 2: يستنى لحد ما Service أو Database تجهز

Edit Copy yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: busybox:1.28
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']

  initContainers:
  - name: init-myservice
    image: busybox:1.28
    command: ['sh', '-c', 'until nslookup myservice; do echo waiting for myservice; sleep 2; done;

  - name: init-mydb
    image: busybox:1.28
    command: ['sh', '-c', 'until nslookup mydb; do echo waiting for mydb; sleep 2; done;']
```

🔎 **الشرح:**

- **عندنا 2 initContainers ::**
  1. **init-myservice بيستنى لحد ما Service اسمها myservice ترد.**
  2. **init-mydb بيستنى لحد ما Database اسمها mydb تبقى جاهزة.**
- **كل واحد بيتنفذ بالترتيب (sequential order)**
- **لو أي واحد فيهم فشل فالـ Kubernetes هيعيد تشغيل البود كله لحد ما ينجح.**

6️⃣ خصائص Init Containers ::

- **بيتنفذوا قبل الـ Containers الأساسية.**
- **بيتنفذوا واحد واحد بالترتيب.**
- **لازم يكملوا بنجاح وإلا البود يفضل يعمل Restart**
- **بيتكتبوا زي أي Container عادي لكن في قسم initContainers**

🔑 **الخلاصة**

- **Multi-Container Pod = كل Container فيه لازم يفضل شغال طول الوقت.**
- **Init Container = Container مؤقت بيخلص مهمة محددة (زي تجهيز البيانات أو الانتظار لشيء خارجي) وبعدها التطبيق الأساسي يبدأ.**
- **تقدر تستخدم أكتر من Init Container ، وهيشتغلوا واحد ورا التاني.**
- **لو أي Init Container فشل فالبود كله هيفضل يعمل Restart لحد ما ينجح.** (خلي بالك سؤال انترفيوهات)

Drain & Cordon & Uncordon

مقدمة عن الموضوع

المحاضرة تناقش السيناريوهات التي تتطلب إيقاف تشغيل Node في Cluster لأغراض الصيانة، مثل:

- تحديث البرامج الأساسية (Base Software Upgrade)
- تطبيق تصحيحات أمنية (Security Patches)

في Kubernetes ، يتكون الـ Cluster من عدة Nodes ، وكل Node يحتوي على Pods تشغّل التطبيقات. عندما تحتاج إلى إيقاف Node للصيانة، يجب التأكد من إدارة الـ Pods بشكل صحيح لتجنب انقطاع الخدمة.

ماذا يحدث عند إيقاف Node ؟

لديك Cluster يحتوي على عدة Nodes و Pods تشغّل التطبيقات.

- عند إيقاف Node فال الـ Pods على هذا الـ Node تصبح غير متاحة (غير قابلة للوصول)

  و تأثير ذلك يعتمد على كيفية نشر الـ Pods ::

  - إذا كان لديك عدة replicas من Pod معين مثل Blue Pod لن يتأثر المستخدمون لأن الـ replicas الأخرى لنفس الـ pod على Nodes أخرى ستستمر في العمل.
  - إذا كان لديك Pod واحد فقط مثل Green Pod فإن المستخدمين الذين يستخدمون هذا التطبيق سيتأثرون لأنه لا توجد replicas أخرى.

كيف يتعامل Kubernetes مع توقف Node ؟

- إذا عاد الـ Node للعمل بسرعة :
  - إذا عاد الـ Node فوراً (قبل انتهاء المهلة الزمنية)، فإن عملية kubectl تعيد تشغيل الـ Pods تلقائياً، وتعود التطبيقات للعمل.
- إذا استمر توقف الـ Node لأكثر من 5 دقائق :
  - Kubernetes يعتبر الـ Pods على الـ Node انها (Dead)
  - إذا كانت الـ Pods جزءًا من ReplicaSet فإن Kubernetes يعيد إنشاء هذه الـ Pods على Node آخر متاح.
  - إذا لم تكن الـ Pods جزءًا من ReplicaSet فإنها تُفقد تماماً ولا يتم إعادة إنشائها تلقائياً.

شرح مصطلح Pod Eviction Timeout :

- هذا هو الوقت الذي ينتظره Kubernetes قبل اعتبار الـ Pods على Node متوقف "ميتة".
- القيمة الافتراضية هي 5 دقائق ، ويتم تعيينها في Controller Manager باستخدام الخيار --pod-eviction-timeout =300s هذا يعني أن Kubernetes سينتظر (5 دقائق) قبل إزالة الـ.Pods

---

ماذا يحدث عند عودة الـ Node بعد Pod Eviction Timeout ؟

- إذا عاد الـ Node للعمل بعد مرور أكثر من 5 دقائق، فإنه يعود "فارغًا(Blank) "، أي بدون أي Pods مجدولة عليه.
- الـ Pods التي كانت عليه :
  - إذا كانت جزءًا من ReplicaSet فسيتم إنشاء replicas جديدة على Node آخر.
  - إذا لم تكن جزءًا من ReplicaSet ، فإنها تُفقد نهائياً.

كيفية إجراء الصيانة بأمان؟

1. لو عايز تعمل صيانة للـ Node زي Upgrade أوReboot
   - لو متأكد إن الـ Node هترجع أقل من 5 دقائق:
     ممكن تعمل Upgrade/Reboot عادي.
   - لكن :عمرك ما تقدر تضمن إنها هترجع في 5 دقائق أو حتى ترجع أصلاً.

إذن الأفضل إنك تجهز الـ Node صح قبل ما توقفها . وبدلاً من الاعتماد على عودة الـ Node في غضون 5 دقائق. هي الاعتماد على أوامر Kubernetes التالية Drain , Cordon , Uncordon

2. أمر Drain

- قبل عمل صيانة هاستخدام أمر kubectl drain <node-name> --ignore-daemonsets --delete-local-data
- لاحظ ان امر –ignore-daemonsets بيتجاهل الـ Daemon-Sets .. يعني دي اللي هايحصلها استثناء وتفضل زي ماهي ودا لأنها عبارة عن Pods خاصة تعمل على كل Node (مثل أدوات المراقبة أو التسجيل). ولا يمكن نقلها عند الـ Drain

✅ اللي بيحصل:

- Kubernetes بيعمل تفريغ (drain) لكل الـ Pods على الـ Node.
- الـ Pods دي بتتـ terminate gracefully (تتقفل بشكل منظم) وتتعملها إعادة تشغيل على Nodes تانية.
- في نفس الوقت بيتعمل Cordon (Unschedulable) لل Node (يعني تتوسم إنها غير قابلة لجدولة Pods جديدة).

3. أمر Cordon (إيقاف الجدولة فقطط)

- لو مش عايز تنقل أو توقف الـ Pods الحالية، لكن عايز تمنع أي Pods جديدة تتشغل على الـ Node عن طريق أمر kubectl cordon <node-name

✅ اللي بيحصل:

- الـ Node بتبقى Unschedulable يعني ممنوع يتعمل عليها جدولة جديدة.
- الـ Pods القديمة اللي شغالة بتفضل زي ما هي.

4. Uncordon Node (إعادة تفعيل الـ Node)

- بعد ما تخلص الصيانة وتعيد تشغيل الـ Node، بتستخدم kubectl uncordon <node-name

✅ اللي بيحصل:

- الـ Node بترجع صالحة لجدولة Pods جديدة.
- لكن خليك عارف: الـ Pods اللي كانت اتنقلت لNodes تانية مش هترجع أوتوماتيكياً.
- إنما أي Pods جديدة أو Pods اتشالت ممكن تتوزع على الـ Node دي عادي.

الخلاصة

- للتحقق من حالة الـ Node قابل للجدولة أم لا : kubectl get nodes لو الـ Status كان : Ready,SchedulingDisabled فده معناه ان الـ Node في حالة Cordon

- أقل من 5 دقائق Down : الـ Pods ممكن ترجع عادي.
- أكتر من 5 دقائقDown : الـ Pods بتعتبرDead

الحل الأفضل للصيانة:

1. استخدم kubectl drain لتفريغ الـ Node
2. اعمل Upgrade أو Patch أو Reboot
3. بعد الرجوع، استخدم kubectl uncordon لتفعيلها من جديد.

Kubernetes Software Version

المقدمة

المحاضرة دي بتتناول موضوع الإصدارات (Releases) والإصدارات الفرعية (Versions) الخاصة بـ Kubernetes
الفكرة إن أي Cluster لـ Kubernetes بيشتغل بإصدار محدد، والإصدارات دي بتتجدد باستمرار سواء بإضافات جديدة أو بإصلاح مشاكل.

ما نعرفه عن الـ API Versions في Kubernetes

أول ما تنزل Kubernetes ، بتنزل نسخة معينة منه. تقدر تعرف النسخة دي من خلال الأمر:

kubectl get nodes

الأمر ده بيجيبلك معلومات عن الـ Nodes اللي عندك، ومنها تقدر تشوف نسخة Kubernetes
مثال: ممكن يطلعلك إن النسخة هي : **v1.11.3**

مكونات رقم الإصدار

إصدار Kubernetes بيتكون من 3أجزاء رئيسية:

- Major version الإصدار الرئيسي
- Minor version الإصدار الفرعي
- Patch version الإصلاحات/الترقيعات

📌 مثال:

v1.11.3

- 1 → Major
- 11 → Minor
- 3 → Patch

- الـ Minor versions بتنزل كل كام شهر علشان تضيف مميزات جديدة.
- أما الـ Patch versions بتنزل بشكل متكرر أكتر لإصلاح مشاكل أو Bugs حرجة.

أنواع الإصدارات

الإصدارات مش كلها Stable ، فيه كمان:

1. Alpha Release
   - بتتجرب فيها المميزات لأول مرة.
   - المميزات فيها مقفولة (disabled) بشكل افتراضي.
   - ممكن تكون مليانة Bugs
2. Beta Release
   - الكود فيها بيكون مجرب أكتر.
   - المميزات الجديدة مفتوحة افتراضيًا (enabled by default)
3. Stable Release
   - النسخة الرسمية اللي تتثبت وتشتغل في الإنتاج (Production)

📌 يعني: أي ميزة جديدة : الأول Alpha بعدين Beta وأخيرًا Stable

---

مكان تنزيل الإصدارات

تقدر تلاقي كل الإصدارات في:

📍 صفحة الإصدارات في GitHub الخاصة بـ Kubernetes

لما تنزل الملف: kubernetes.tar.gz وتفكه، هتلاقي جواه كل مكونات الـ Control Plane زي:

- kube-apiserver
- kube-controller-manager
- kube-scheduler
- kubelet
- kubectl

وكلهم بيبقوا بنفس الإصدار اللي نزلته.

ملاحظات مهمة عن الإصدارات الداخلية

في مكونات تانية شغالة جوا Kubernetes لكنها مشاريع منفصلة ولها إصدارات خاصة بيها زي:

- ETCD اللي بيخزن بيانات الـ Cluster
- CoreDNS اللي بيحل أسماء الـ Services

عشان كده، في Release Notes لأي إصدار جديد لـ Kubernetes هتلاقي مكتوب إيه الإصدارات المدعومة من ETCD و CoreDNS

أوامر عملية مفيدة

- لمعرفة نسخة Kubernetes اللي عندك:
- kubectl version --short
- لمعرفة إصدارات المكونات:
- kubectl get componentstatuses
- لتصفح الإصدارات على GitHub

👉 Kubernetes GitHub Releases

الخلاصة

- إصدارات Kubernetes بتتكون من Major.Minor.Patch
- كل ميزة جديدة تعدي بمراحل Alpha → Beta → Stable
- Control Plane كله بيكون بنفس النسخة، لكن مكونات زي ETCD/CoreDNS ليها نسخ مستقلة.
- دايمًا تابع Release Notes لأي نسخة علشان تفهم التغييرات والمتطلبات.

References :

https://kubernetes.io/docs/concepts/overview/kubernetes-api/

https://github.com/kubernetes/community/blob/master/contributors/devel/sig-architecture/api-conventions.md

https://github.com/kubernetes/community/blob/master/contributors/devel/sig-architecture/api_changes.md

Cluster Upgrade Process

المقدمة

في المحاضرة اللي فاتت شوفنا إزاي بيتم إدارة الإصدارات المختلفة للسوفت وير، وإزاي كل Component من مكونات Kubernetes بيكون له إصدار خاص.
لكن هنا هنركّز على Control Plane components زي الـ API Server, Controller Manager, Scheduler

---

توافق الإصدارات بين المكونات

- لازم المكونات الرئيسية تبقى متوافقة.
- الـ kube-apiserver هو المكوّن الأساسي اللي بيتواصل مع كل باقي المكونات.
- القاعدة:
  - ولا مكوّن تاني ينفع يكون أعلى من نسخة الـ API Server
  - لكن ينفع يكون أقل (Minor versions)

📌 مثال عملي:

- لو الـ API Server عنده نسخة 1.10 :
  - الـ Controller Manager & Scheduler يقدروا يبقوا 1.10 أو 1.9
  - الـ Kubelet & Kube-proxy يقدروا يوصلوا لحد 1.8
  - لكن ماينفعش يبقوا 1.11 أعلى من الـ API Server

إستثناء خاص بالـ kubectl

أداة kubectl (أداة التحكم) ممكن تكون:

- في نفس إصدار الـ API Server
- أو إصدار أعلى (حد نسخة واحدة فوق).
- أو إصدار أقل (حد نسخة واحدة تحت).

وده بيديك مرونة أثناء عمليات التحديث.

دعم الإصدارات في Kubernetes

Kubernetes بيدعم آخر (نسخ 3) Minor فقط.

📌 يعني:

- لو عندك إصدارات 1.10, 1.11, 1.12 : لسه مدعومة.
- أول ما تنزل نسخة 1.13 فأوتوماتيك الدعم هيبقى لـ 1.11, 1.12, 1.13

علشان كده : أفضل وقت تعمل Upgrade للـ Cluster هو قبل ما نسخة جديدة Major أو Minor تنزل.

طريقة الترقية (Upgrade Path)

ماينفعش تقفز مرة واحدة من 1.10 لـ 1.13

📌 لازم خطوة بخطوة:

- من 1.10 → 1.11
- بعدين 1.11 → 1.12
- بعدين 1.12 → 1.13

أنواع الـ Clusters والـUpgrade

- Managed Clusters (مدارة من Cloud Providers زي GKE, EKS, AKS)
  عملية التحديث سهلة جدًا وأحيانًا بتتم بزرارين في Dashboard
- Clusters معمولة بـ kubeadm
  عندك أوامر مخصصة تساعدك تخطط وتنفّذ التحديث. (في اخر الصفحة)
- Clusters معمولة يدويًا (from scratch)
  لازم تحدّث كل مكون (API Server, Scheduler, Controller Manager, Kubelet, Proxy) يدويًا.

خطوات الترقية الرئيسية

عملية الـ Upgrade بتتم على خطوتين كبار:

1 تحديث Master Nodes (Control Plane)

- أثناء التحديث ← الـ API Server والمكونات الإدارية (Scheduler, Controller Manager) بيبقوا Down لحظيًا.
- لكن الـ Worker Nodes (اللي فيها التطبيقات) بتكمل تشتغل عادي.
- خلال الفترة دي:
  - مش هتقدر تنفذ أوامر kubectl
  - مش هتقدر تنشر (Deploy) تطبيقات جديدة.
  - مش هيتعمل Reschedule للـ Pods لو حصل Fail
- بعد التحديث ← كل حاجة بترجع طبيعية.

2 تحديث Worker Nodes

- عندك 3 استراتيجيات:
  1. تحديث كل الـ Nodes مرة واحدة بس خلي بالك ان ده بيوقف التطبيقات فترة (Downtime)
  2. Rolling Update (واحد واحد) :
     - تعمل Drain لـ Node واحد (تنقل الشغل منه).
     - تحدّثه ← ترجعه.
     - تروح للتاني ← وهكذا.
     - ده بيخلي التطبيقات شغالة بدون توقف.

       أوامر مهمة:
     - kubectl drain <node-name> --ignore-daemonsets ده بيفضي الـ Pods من النود
     - kubectl uncordon <node-name> ده بيرجّع النود يشتغل عادي
  3. إضافة Nodes جديدة (بالإصدار الجديد) ونقل الشغل عليها، وبعدين إزالة القديمة (مناسب للـ Cloud)

```
kubectl drain node-1
kubectl uncordon node-1
kubectl drain node-2
kubectl uncordon node-2
kubectl drain node-3
kubectl uncordon node-3
```

```
apt-get upgrade -y kubeadm=1.12.0-00
apt-get upgrade -y kubelet=1.12.0-00
kubeadm upgrade node config --kubelet-version v1.12.0
systemctl restart kubelet
```

أوامر عملية للتحديث باستخدام kubeadm

1. تحديث kubeadm نفسه : apt-get upgrade -y kubeadm=1.x.x-00
2. التحقق من خطة التحديث : kubeadm upgrade plan هيعرض: النسخة الحالية والنسخ المتاحة وخطوات الترقية.
3. تنفيذ الترقية : kubeadm upgrade apply v1.x.x

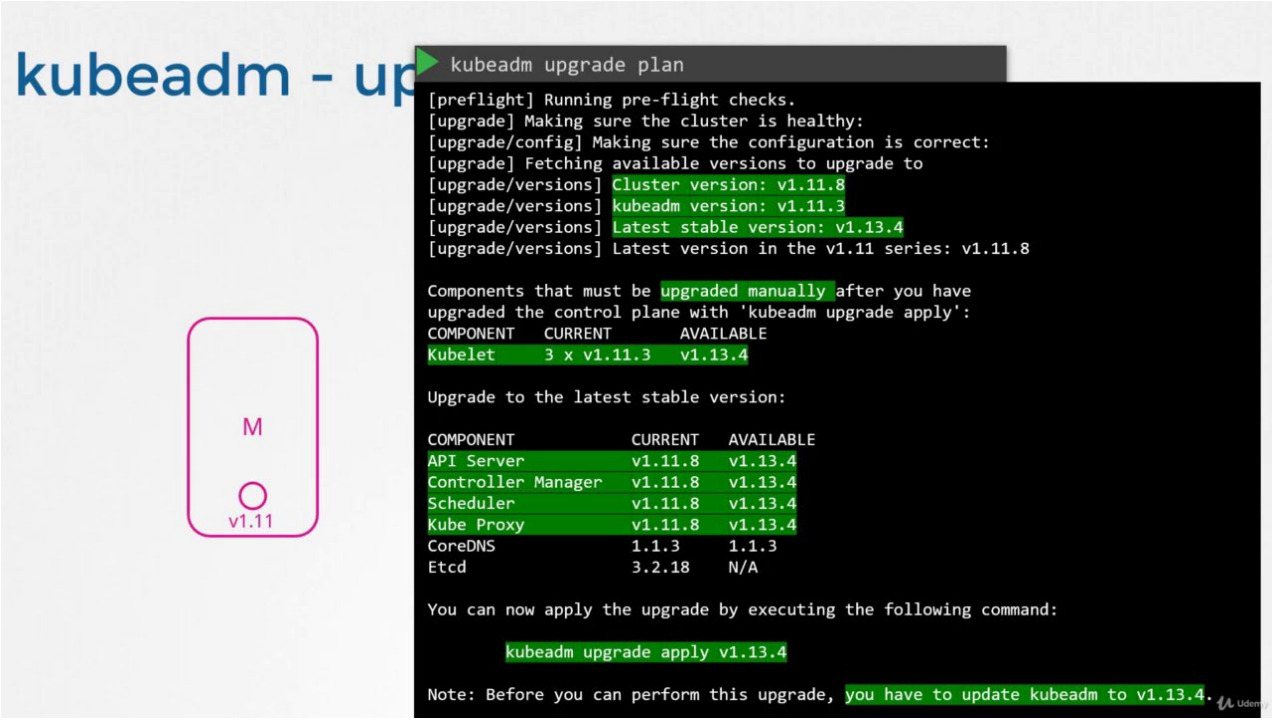
❖ تحديث الـ kubelet على كل Node :

apt-get upgrade -y kubelet=1.x.x-00 then systemctl restart kubelet

❖ يفضل تستخدم امر apt-mark hold kubeadm و apt-mark hold kubelet علشان تعمل "تجميد" (hold) للباكيدج kubelet or kubeadm ، بحيث ما يتعملوش تحديث تلقائي لو شغلت: apt-get upgrade

طيب ليه ؟؟ قالك ان الفكرة إنك في k8s لازم تعمل upgrade للـ components (kubeadm, kubelet, kubectl) بترتيب معين (أول حاجة kubeadm، بعدين kubelet، وأخيرًا kubectl)

علشان كده ساعات بنعمل hold للباكيدجات دي لحد ما نكون جاهزين نعمل الـ upgrade بطريقة منظمة.



ملاحظات مهمة

- أداة kubectl get nodes بتعرض إصدار الـ Kubelet مش إصدار الـ API Server
- علشان كده ممكن تشوف Master Node بإصدار قديم، لكن فعليًا الـ Control Plane متحدث.
- تحديث الـ Worker لازم يتعمل واحد واحد علشان تتجنب Downtime

الخلاصة

- التحديث في Kubernetes بيتم تدريجيًا Minor by Minor
- فيه قواعد توافق بين مكونات الـ Control Plane والـ Workers
- عندك 3 طرق لتحديث الـ Workers كلهم مع بعض، واحد واحد، أو إضافة Nodes جديدة.
- أوامر kubectl drain و uncordon و kubeadm upgrade هي الأدوات الأساسية اللي لازم تحفظها.

Backup and Restore Methods

المقدمة

هانتكلم عن طرق أخذ النسخ الاحتياطية (Backup) واستعادة البيانات (Restore) في Kubernetes

---

إيه اللي لازم أعمل له Backup ؟

- أي Kubernetes Cluster بيتخزن فيه حاجات أساسية:
  1. ETCD database : ده المكان اللي فيه كل cluster state زي الـ (pods, services, nodes, configmaps ... etc)
  2. Persistent Volumes (PV) : لو التطبيقات بتستخدم تخزين دائم.
  3. YAML definition files : ملفات التعريف اللي بتعمل بيها (Deployments, Services, ConfigMaps, Secrets ... etc)

طرق إنشاء الموارد

- في السابق استخدمنا طريقتين:
  1. Imperative : أوامر مباشرة زي kubectl create namespace أو kubectl expose
  2. Declarative (ملفات YAML) : زي kubectl apply -f app.yaml

📌 ملاحظة مهمة:

الـ Declarative أفضل عشان بيخلي كل Configurations محفوظة في ملفات. تقدر تشاركها أو تخزنها في GitHub

---

مشكلة الـ Imperative

❖ لو حد من التيم عمل حاجة بـ Imperative (أمر مباشر) ومكتبهاش في ملف فللأسف مش هتعرف تعيدها وقت المشكلة.

❖ الحل:

- تعمل Backup مباشر من API Server باستخدام kubectl get وتخزن كل الموارد بصيغة YAML

📌 مثال عملي لعمل Backup Script :

```
kubectl get all --all-namespaces -o yaml > all-resources-backup.yaml
```

- ممكن تضيف كمان:

```
kubectl get configmaps,secrets,pv,pvc,ingress --all-namespaces -o yaml > extra-resources.yaml
```

أدوات Backup جاهزة

- مش لازم تكتب Script بنفسك.
- فيه أدوات زي:
  - Velero (Heptio Ark) سابقًا : ووظيفتها إنها بتاخد Backup باستخدام الـ Kubernetes API
  - Velero كمان بتعمل Backup للـ Persistent Volumes (لو مستخدم معها Plugins خاصة بـ Storage)

Backup لـ ETCD

- ETCD هو قلب الـ Cluster ، بيخزن كل حاجة عن الـ state
- بيتخزن عادة في Master Nodes تحت:

  /var/lib/etcd
- ينفع تعمل Backup بـ 2 طرق:
  1. Backup للـ data directory (ملفاته) .
  2. Snapshot باستخدام etcdctl

📌 أوامر عملية:

❖ أخذ Snapshot

```
ETCDCTL_API=3 etcdctl snapshot save snapshot.db \
  --endpoints=https://127.0.0.1:2379 \
  --cacert=/etc/kubernetes/pki/etcd/ca.crt \
  --cert=/etc/kubernetes/pki/etcd/server.crt \
  --key=/etc/kubernetes/pki/etcd/server.key
```

بعرف كل الـ Paths بتاع الـ options دي من خلال :

```
kubectl -n kube-system describe po etcd-master
```

- هيعمل ملف اسمه snapshot.db
- تقدر تغير مكانه لو حبيت.
- ملحوظة مهمة : مع كل أوامر etcdctl لازم تحدد:
  - CA certificate
  - Server certificate
  - Server key
  - Endpoint
  - لو مش محددهم ← مش هيشتغل.

خلي بالك علشان متحفظش كتير ان امر
ETCDCTL_API=3 etcdctl snapshot save -h
هايجيبلي كل الاوبشنز المتاحة اللي المفروض تتحط مع الامر ده

طاب افرض بعد مانت كتبت الاوبشنز بال paths بتاعتها بس عاوز تعمل test هل كل حاجة صحيحة ولا لأ قبل ماتعمل run للأمر ؟؟
عن طريق ETCDCTL_API=3 etcdctl member list "options=paths"

❖ التأكد من حالة الـ Snapshot

```
ETCDCTL_API=3 etcdctl snapshot status snapshot.db
```

❖ Restore من Snapshot

- خطوات الاستعادة:

1. توقف الـ kube-apiserver (عشان بيعتمد على etcd)

```
systemctl stop kube-apiserver
```

2. اعمل Restore :

```
ETCDCTL_API=3 etcdctl snapshot restore snapshot.db \
  --data-dir=/var/lib/etcd-from-backup \
  --initial-cluster=master=https://127.0.0.1:2380 \
  --initial-advertise-peer-urls=https://127.0.0.1:2380 \
  --name=master \
  --initial-cluster-token=new-cluster-token
```

✅ هنا لازم تحدد token جديد و data-dir جديد (اللي هاتعمل عليه ال restore زي /var/lib/etcd-from-backup) ..
طيب ليه ؟؟! علشان ميدخلش بالخطأ في Cluster قديم.
لأنك لما بتعمل Restore فبيعمل new cluster configuration and configures the members of ETCD as new members to a new cluster

3. عدّل ملف Config لـ etcd من خلال (/etc/kubernetes/manifests/etcd.yaml) وتحديدا هاتعدل الـ data-dir والـ mountPath وال hostPath وكمان هاتضيف الـ initial-cluster-token وكل ده عشان يشير للمكان الجديد .
4. Reload & Restart :

```
systemctl daemon-reload
systemctl restart etcd
systemctl start kube-apiserver
```

✅ ✅ كده الـ Cluster بيرجع للحالة القديمة اللي كان عليها وقت الـ Backup

Managed Kubernetes

لو بتستخدم Managed Cluster (زي GKE, EKS, AKS)

- **ساعات مش بيكون عندك Access مباشر لـ ETCD**
- **الحل الوحيد هو تعمل Backup من خلال kube-apiserver يعني باستخدام (kubectl أو Velero)**

الخلاصة

- **عندك طريقتين أساسيين للـ Backup**
  1. **Backup via kube-apiserver عن طريق (kubectl أو Velero)**
  2. **Backup via ETCD snapshot**
- **الأفضل تستخدم الاتنين مع بعض.**
- **ولو Managed Cluster فغالبًا هتستخدم طريقة Backup via kube-apiserver فقط.**

📊 مقارنة بين طرق الـ Backup في Kubernetes

العنصر	Backup via API Server (kubectl / Velero)	Backup via ETCD Snapshot
نطاق البيانات	بيعمل Backup للـ Kubernetes Objects (Pods, Deployments, Services, ConfigMaps, Secrets ... إلخ).	بيعمل Backup للـ Cluster state كامل (بما في ذلك جميع الـ Objects، الـ Nodes info ...).
Persistent Volumes (PV)	محتاج أداة زي Velero + Plugin خاص بالـ Storage Provider.	مش بيحتوي الـ PV data نفسها (إلا لو بتخزنها في etcd، وده نادر).
سهولة الاستخدام	سهل مع `kubectl get -o yaml` أو Velero. مناسب للـ Automation.	محتاج أوامر etcdctl وصلاحيات على Master Node. أصعب شوية.
إعادة الاستخدام	ممكن تاخد الـ YAML Files وتعيد نشرها على Cluster مختلف.	Snapshot بيرجع الـ Cluster لحالته السابقة (مش مناسب لإعادة نشر App على Cluster جديد).
المرونة	تقدر تعمل Restore لـ App واحد أو Namespace واحد فقط.	Restore بيرجع كل حاجة زي ما كانت (كل الـ Cluster).
المسؤولية (Access)	ينفع حتى في Managed Kubernetes (GKE, EKS, AKS).	غالبًا مش متاح في Managed Clusters (لأنك مش بتشوف etcd).
الأمان	بيتعامل مع الـ RBAC (صلاحيات الـ kubectl).	محتاج Certificates (CA, cert, key) عشان تقدر تستخدم etcdctl.
الأداء والسرعة	أسرع في Backup/Restore للـ Apps الصغيرة.	أسرع وأضمن لو عايز Restore للـ Cluster كله.
أفضل استخدام	Backup للتطبيقات + Configurations. مناسب للـ Dev/Test.	Backup للـ Cluster State بالكامل. مناسب للـ Production Disaster Recovery.

References :

https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/#backing-up-an-etcd-cluster

https://github.com/etcd-io/etcd/blob/master/Documentation/op-guide/recovery.md

https://www.youtube.com/watch?v=qRPNuT080Hk

Kubernetes Security Primitives

المحاضرة دي بتتكلم عن مفاهيم الأمان (Security Primitives) في Kubernetes
بما إن Kubernetes بقى المنصة الأساسية لتشغيل الـ Production-grade applications، فـ الأمان يعتبر عنصر أساسي.

---

1 أمان الـ Hosts او الـ Nodes نفسها

قبل ما ندخل في تفاصيل Kubernetes ، لازم الأول تأمّن الـ Nodes (السيرفرات/الماكينات اللي عليها الـ Cluster)

- ✅ امنع Root Access المباشر.
- ✅ عطّل Password Authentication
- ✅ استخدم فقط SSH Key Authentication
- ✅ طبق إجراءات إضافية لتأمين البنية التحتية سواء Physical أو Virtual Machines

📌 لأن: لو أي Node (Host) اتعرض للاختراق → الكل بيتعرض للخطر.

---

2 الـ API Server (خط الدفاع الأول)

- الـ kube-apiserver هو المكون الأساسي في Kubernetes
- أي عملية بتحصل في الـ Cluster بتمر من خلاله (سواء باستخدام kubectl أو API مباشر)
- عشان كده لازم أول حاجة نأمّنه.

---

3 Authentication & Authorization

🔑 Authentication (مين يقدر يدخل الـ Cluster ؟؟)

- بيتحدد عن طريق طرق مختلفة:
  - Static File فيه User IDs + Passwords
  - Tokens (Bearer Tokens)
  - Certificates (x509 Client Certs)
  - External Providers زي LDAP, OIDC
- بالنسبة للـ machines/applications : بنستخدم Service Accounts

📌 أوامر مهمة:

- لإنشاء Service Account عن طريق kubectl create serviceaccount my-app-sa
- لعرض Service Accounts عن طريق kubectl get serviceaccounts

🔄 Authorization (اللي دخل يقدر يعمل إيه ؟؟؟)

- بيتحدد باستخدام: RBAC (Role-Based Access Control)
  - Users/Groups بيرتبطوا بـ Roles/ClusterRoles
  - Roles بتحدد الـ Permissions : (زي Get, List, Create .... etc)
- فيه كمان طرق تانية زي :
  - ABAC (Attribute-Based Access Control)
  - Node Authorizers
  - Webhooks

```yaml
# role.yaml
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list"]
```

Edit Copy

```
kubectl apply -f role.yaml
```

4️⃣ TLS Encryption بين المكونات

كل مكونات الـ Control Plane والـ Worker Nodes بتتواصل مع بعض باستخدام TLS Certificates

- ETCD ↔ API Server
- Controller Manager ↔ API Server
- Scheduler ↔ API Server
- Kubelet ↔ API Server
- Kube-proxy ↔ API Server

📌 هنتعلم إزاي نولّد Certificates باستخدام openssl أو cfssl ونعمل Trust بين المكونات.

5️⃣ Network Policies (بين الـ Pods)

- افتراضيًا : كل Pod يقدر يكلم أي Pod تاني داخل الـ Cluster
- ده مش آمن في الـ Production
- ممكن تحدد مين يكلم مين باستخدام Network Policies

💡 هنا بس الـ Pods اللي عندها access=true تقدر تدخل على الـ Nginx Pod

```yaml
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: allow-nginx
  namespace: default
spec:
  podSelector:
    matchLabels:
      app: nginx
  ingress:
  - from:
    - podSelector:
        matchLabels:
          access: "true"
```

📝 الخلاصة

- أمان الـ Hosts أساس كل حاجة.
- الـ API Serverهو خط الدفاع الأول و لازم يتأمن كويس.
- عندنا مستويين:
  - Authentication (مين يدخل)
  - Authorization (يعمل إيه)
- كل التواصل بين مكونات Kubernetes مؤمّن بـ TLS
- تقدر تتحكم في تواصل الـ Pods مع بعض باستخدام Network Policies

أهم خمس خطوات ::

🔒 قفل الـ Hosts وتأمين الدخول بـ SSH Keys

🔑 تفعيل RBAC + Authentication قوي.

📜 تأمين الاتصال بين المكونات بـ TLS

🌐 تطبيق Network Policies للتحكم بين الـ Pods

🔑 استخدام Secrets + Policies لمنع الحاويات من الدخول كـ Root.

Authentication in Kubernetes

أنواع الـ Users اللي بتتعامل مع الـ Cluster

- Admins / Developers (بشر) : بيستخدموا kubectl أو API عشان يعملوا Deploy أو Configure
- End Users (مستخدمين للتطبيقات) : دول بيتعاملوا مع التطبيقات نفسها مش مع الـ cluster (مش موضوعنا هنا.)
- Robots / Services (غير بشر) : زي الـ CI/CD pipelines ، monitoring systems أو أي integration مع الـ cluster

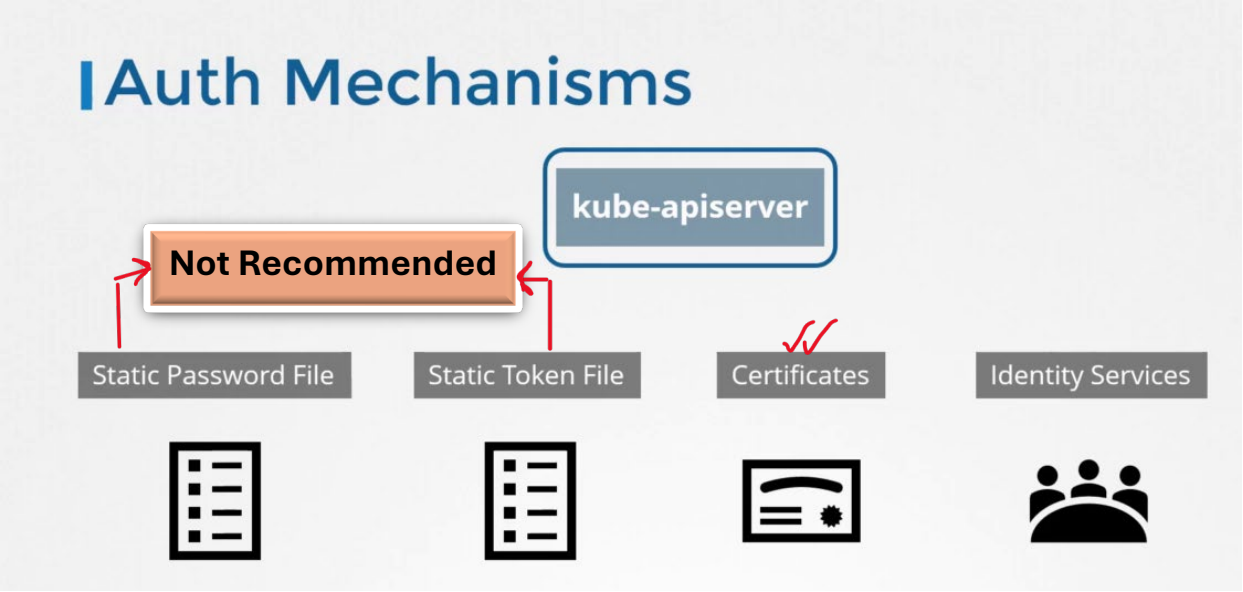
📌 الخلاصة: عندنا Human Users + Service Accounts

Note : Service Accounts are not part of CKA curriculum, instead, they are part of CKAD. So it's explanation are available in the CKAD course on kodekloud platform.

مين بيدير الـ Authentication ؟

- كل حاجة بتمر عبر kube-apiserver
- هو اللي بيقرر مين يدخل.
- Kubernetes نفسه ماعندوش Database users جواه (يعني مش زي Linux) .. ولكنه بيعتمد على :
  1. Static files (Passwords / Tokens)
  2. Certificates
  3. External Auth (LDAP, Kerberos, OIDC...)
- لكن Service Accounts بيديرها Kubernetes نفسه.

طاب ايه هي ال Authentication Mechanism ؟؟



Static Password File

📂 الملف بيكون CSV بالشكل ده:

```
password,username,uid,"group"
mypwd123,admin,1001,"system:masters"
```

⚙️ تفعيل الملف:

❖ تضيفه كـ Option في الـ kube-apiserver

```
spec:
  containers:
  - name: kube-apiserver
    command:
    - kube-apiserver
    - --basic-auth-file=/etc/kubernetes/auth/users.csv
```

❖ بعد كده تعمل:

```
kubectl -n kube-system edit pod kube-apiserver-master
# أو لو kubeadm:
sudo vi /etc/kubernetes/manifests/kube-apiserver.yaml
```

Kubeadm هيعمل Restart للـ API Server أوتوماتيك.

اختبار الدخول بالـ User/Pass

curl -v -k https://localhost:6443/api/v1/pods --user "admin:mypwd123"

Static Token File

❖ بدل الباسوورد:

```
token,username,uid,"group"
ab1234567890xyz,dev-user,1002,"dev-team"
```

❖ تضيف الخيار للـ API Server :

- --token-auth-file=/etc/kubernetes/auth/tokens.csv

❖ وتجرب الطلب بالـ Token :

curl -v -k https://localhost:6443/api/v1/pods --header "Authorization: Bearer ab1234567890xyz

---

ملاحظات مهمة ⚠️

- عيب كبير وهو ان الملفات دي بتتخزن Clear Text (مش مشفرة).
- وبالتالي فهو مش مناسب Production بس مفيد في التعلم والفهم.
- وفي الحقيقة الشركات بتستخدم:
  - Certificates (TLS).
  - OIDC (OpenID Connect) مع Google, Azure, Okta , ...
  - LDAP / Kerberos

- خلي بالك لو مستخدم Kubeadm لازم تضيف Volume Mount للـ Static Files عشان توصلها جوه الـ Pod

```yaml
volumeMounts:
- mountPath: /etc/kubernetes/auth
  name: auth
volumes:
- name: auth
  hostPath:
    path: /etc/kubernetes/auth
    type: DirectoryOrCreate
```

إعداد Basic Authentication على Kubernetes باستخدام kubeadm

⚠️ ملاحظة : ده غير آمن ومش بيتعمل في Production، ده بس للتعلم.

إنشاء ملف المستخدمين

أول حاجة، بتعمل ملف فيه بيانات المستخدمين:

المكان : /tmp/users/user-details.csv

المحتوى:

```
password123,user1,u0001
password123,user2,u0002
password123,user3,u0003
password123,user4,u0004
password123,user5,u0005
```

```
<password>,<username>,<user_id>
```

تعديل ملف kube-apiserver

❖ في كلاستر معمول بـ kubeadm ، الـ API Server متسطب كـ Static Pod

الملف موجود في: /etc/kubernetes/manifests/kube-apiserver.yaml

❖ إضافة Mount للمستخدمين

بتضيف فولدر المستخدمين جوه البود عشان الـ API Server يقدر يشوف الملف :

```yaml
volumeMounts:
- mountPath: /tmp/users
  name: usr-details
  readOnly: true
```

وتحت `volumes` :

Edit Copy yaml

```yaml
volumes:
- hostPath:
    path: /tmp/users
    type: DirectoryOrCreate
  name: usr-details
```

ده معناه: خد الفايل من السيرفر وحطه جوه البود.

تفعيل الـ Basic Auth فيAPI Server

في قسم command الخاص بالـ kube-apiserver ، تضيف:

- --basic-auth-file=/tmp/users/user-details.csv

يعني بقوله: استخدم ملف المستخدمين ده للمصادقة.

عمل Role و RoleBinding

بعد ما خلصنا المصادقة، لازم نحدد إيه الصلاحيات اللي اليوزرز يقدروا يعملوها.

ده بيتم عن طريق RBAC (Role-Based Access Control)

❖ Role : pod-reader

```yaml
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

معناه: فيه Role اسمه `pod-reader` في الـ `default` namespace

يسمح بقراءة الـ Pods (Get, Watch, List).

❖ RoleBinding: read-pods

```yaml
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: read-pods
  namespace: default
subjects:
- kind: User
  name: user1
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

ده بيربط اليوزر `user1` بالـ Role اللي اسمه `pod-reader` .

يعني user1 يقدر يقرأ البودات.

تجربة تسجيل الدخول

دلوقتي تقدر تعمل طلب للـ API Server باستخدام بيانات اليوزر:

```
curl -v -k https://localhost:6443/api/v1/pods -u "user1:password123"
```

- **-k عشان يتجاهل الـ TLS certificate error (لأنك في local)**
- **-u "user1:password123" (يوزر + باسورد من الملف).**

لو كله تمام فهاتقدر تشوف لستة البودات اللي في namespace default

---

🎯 **الخلاصة**

1. **عملنا ملف CSV فيه المستخدمين وكلمات السر.**
2. **عدلنا الـ API Server Static Pod نعمل له Mount للفايل ونستخدمه بـ --basic-auth-file**
3. **أنشأنا Role + RoleBinding عشان ندي صلاحيات لليوزر.**
4. **جربنا الدخول على الـ API Server باستخدام الـ curl**

TLS Introductions

مقدمة

- الموضوع عن تأمين Kubernetes Cluster باستخدام TLS Certificates
- الهدف هو حماية الاتصال بين مكونات الكلاستر.
- لكن أي مشكلة في TLS بتكون صعبة لو مش فاهم الأساسيات مثلاً (Expired Certificate أو CN غلط)

📌 مثال عملي:

لو شهادة apiserver.crt خلصت مدتها(Expired) ، الكلاستر هيوقف لأن الـ kube-apiserver مش هيقدر يشتغل.

- للتحقق من صلاحية الشهادة : openssl x509 -in /etc/kubernetes/pki/apiserver.crt -noout -enddate

لو ظهر notAfter=2024-12-31 يبقى دي نهاية صلاحيتها.

📌 TLS = Transport Layer Security

- بيأمن الاتصال (مثلاً بين kube-apiserver و etcd)
- بيعتمد على Certificates + Private/Public Keys

المحتوى والطريقة اللي هانشرح بيها الـ Topic ده علشان مهم جدا :

الـ Topic متقسم لجزئين:

1. الأساسيات (Basics) :
   - إيه هوCertificate ؟
   - إيه الفرق بين Private Key وPublic Key ؟
   - يعني إيه Certificate Authority (CA) ؟
   - يعني إيه Chain of Trust ؟
   - الفرق بين Self-Signed و CA-Signed Certificates
2. تطبيقها في Kubernetes :
   - أي مكون بيحتاج شهادة؟
   - إزاي تُنشئ شهادات لكل مكون؟
   - إزاي تتأكد من صحتها؟
   - إزاي تصلّح مشكلة TLS جوه الكلاستر؟

أوامر عملية أساسية مع TLS Certificates

TLS BASICS

ليه محتاجين Certificates ؟

الفكرة الأساسية إن أي اتصال بين جهازين على الشبكة (زي المستخدم والسيرفر) محتاج يكون:

1. مؤمّن (Encrypted) : البيانات اللي بتتبعت تكون مش مفهومة لأي هاكر بيتصنت.
2. موثوق (Trusted) : المستخدم يتأكد إن السيرفر اللي بيكلمه هو فعلاً السيرفر الصح مش حد مزيف.

مثال: لو دخلت على موقع البنك بتاعك من غيرSSL ، الباسورد هيتبعت نص صريح، والهاكر يقدر يقراه لكن مع TLS، البيانات بتتبعت مشفرة.

التشفير المتماثل (Symmetric Encryption)

- بيستخدم مفتاح واحد للتشفير وفك التشفير.
- المشكلة إن لازم تبعت نفس المفتاح للطرف التاني عشان يفك البيانات وبالتالي فالهاكر ممكن يعترض المفتاح أثناء ما أنت بتبعته

التشفير غير المتماثل (Asymmetric Encryption)

- هنا في مفتاحين:
  - Private Key (سري معاك بس)
  - Public Key (متاح لأي حد)
- أي رسالة تتقفل بـ Public Key، ما تتفتحش غير بـ Private Key

وده الأساس اللي اتبنى عليه SSH و HTTPS

استخدام التشفير في SSH

بدل الباسوردات، نستخدم Key Pairs :

- ❖ أوامر عملية: ssh-keygen
  - بينتج ملفين:
    - id_rsa → Private Key
    - id_rsa.pub → Public Key
- ❖ تحط الـ Public Key في السيرفر داخل: ~/ .ssh/authorized_keys
- ❖ وبعدها الدخول بيكون : ssh -i ~/.ssh/id_rsa user@server_ip

لو عندك سيرفرات كتير فاهاتكرر نسخ الـ Public Key لكل السيرفرات.

الرجوع لمشكلة الويب (HTTPS)

- التشفير المتماثل (Symmetric) أسرع، لكن فيه مشكلة تبادل المفتاح.
- الحل: نستخدم التشفير غير المتماثل (Asymmetric) في البداية عشان ننقل مفتاح متماثل بشكل آمن.

أوامر عملية لتوليد مفاتيح ويب:

```
openssl genrsa -out private.key 2048
openssl rsa -in private.key -pubout -out public.key
```

```
openssl genrsa -out my-bank.key 1024
my-bank.key
```

```
openssl rsa -in my-bank.key -pubout > mybank.pem
my-bank.key mybank.pem
```

المشكلة: هجمات انتحال الهوية (Phishing)

هاكر ممكن يقلد موقع البنك ويولّد شهادة لنفسه وطبعا المستخدم يكتب بياناته وهو فاكر إنه الموقع الصح.

الحل : الشهادات الموقعة من جهات موثوقة (Certificate Authority - CA)

Certificate Authorities (CA)

```
openssl req -new -key my-bank.key -out my-bank.csr
    -subj "/C=US/ST=CA/O=MyOrg, Inc./CN=my-bank.com"
my-bank.key my-bank.csr
```

- دي شركات مشهورة زي DigiCert, GlobalSign, Comodo :
- السيرفر بيولّد CSR (Certificate Signing Request) ويبعتها للـCA
- الـ CA تتحقق من الهوية، وتوقّع الشهادة.

أمر عملي لتوليد CSR : openssl req -new -key private.key -out my-bank.csr

- -new : بيعمل طلب شهادة جديد .(CSR)
- -key my-bank.key : بيستخدم المفتاح الخاص (Private Key) الموجود في الملف my-bank.key (متغير)
- -out my-bank.csr : بيكتب الـ CSR في الملف my-bank.csr
- ممكن تضيف Option الـ -subj وتحددله بيانات الـ Certificate (Country, State, Organization, Common Name) مباشرة في نفس الأمر، من غير ما يسألك. (ودا بيبقي مفيد في ال Automation)

آلية الثقة (Trust Mechanism)

- المتصفحات بيكون فيها مسبقًا Public Keys للـ CAs
- لما السيرفر يدي المتصفح شهادة، المتصفح يشيك: هل موقعة من CA معروف؟
- لو الشهادة Self-Signed (موقعة بنفسها) فالمتصفح يحذرك.

Private CAs (للاستخدام الداخلي)

في شركات كتير بتبني CA داخلي عشان الخدمات الداخلية (Payroll, Email)

- بيولّدوا CA داخلي.
- يوزعوا Public Key بتاعه على أجهزة الموظفين.
- كده أي Certificate داخلي هيكون موثوق.

ملخص عملية HTTPS

1. السيرفر يولّد مفتاحين (Private & Public)
2. يولّد CSR ويبعتها لـ CA
3. الـ CA توقّع وترجع Certificate
4. السيرفر يركّب الـ Certificate
5. المستخدم يزور الموقع وفي نفس الوقت المتصفح يتأكد من الشهادة.
6. المتصفح يولّد Symmetric Key ويشفره بـ Public Key بتاع السيرفر.
7. السيرفر يفكّه بـ Private Key
8. من اللحظة دي فكل الاتصالات بتكون Symmetric (سريعة وآمنة)

Client Certificates (أحياناً)

- مش بس السيرفر اللي يقدّم شهادة، ممكن كمان العميل يقدم شهادة عشان يثبت نفسه (مش مستخدم مزيف).
- لكن نادرًا ما بنشوف ده في الويب.

Public Key Infrastructure (PKI)

- المنظومة الكاملة:
  - Certificates + CAs + التوزيع والتحقق + Keys
- كل ده اسمه PKI

ملاحظات مهمة عن الملفات (ازاي تعرف وتفرق بين الملف ده هل هو public or private من خلال الاسم)

- الملفات اللي فيها Public Key غالبًا crt أو pem.
- الملفات اللي فيها Private Key غالبًا key أو key.pem-

مثال:

- server.crt → شهادة السيرفر (Public)
- server.key → المفتاح الخاص (Private)

TLS in Kubernetes

الأنواع الأساسية للشهادات

في أي Cluster Kubernetes فيه 3 أنواع من الشهادات:

1. Serving Certificates بتتثبت على الـ Servers زي (kube-apiserver, etcd, kubelet) عشان يقدّموا خدمات آمنة بالـ TLS
2. Root Certificates (CA) دي اللي بتصدرها Certificate Authority وبتستخدم لتوقيع باقي الشهادات عشان يثقوا فيها.
3. Client Certificates دي على الـ Clients (Admins, kubectl, scheduler, controller, kube-proxy) عشان يتأكدوا إنهم مستخدمين حقيقيين.

تمييز أسماء الملفات

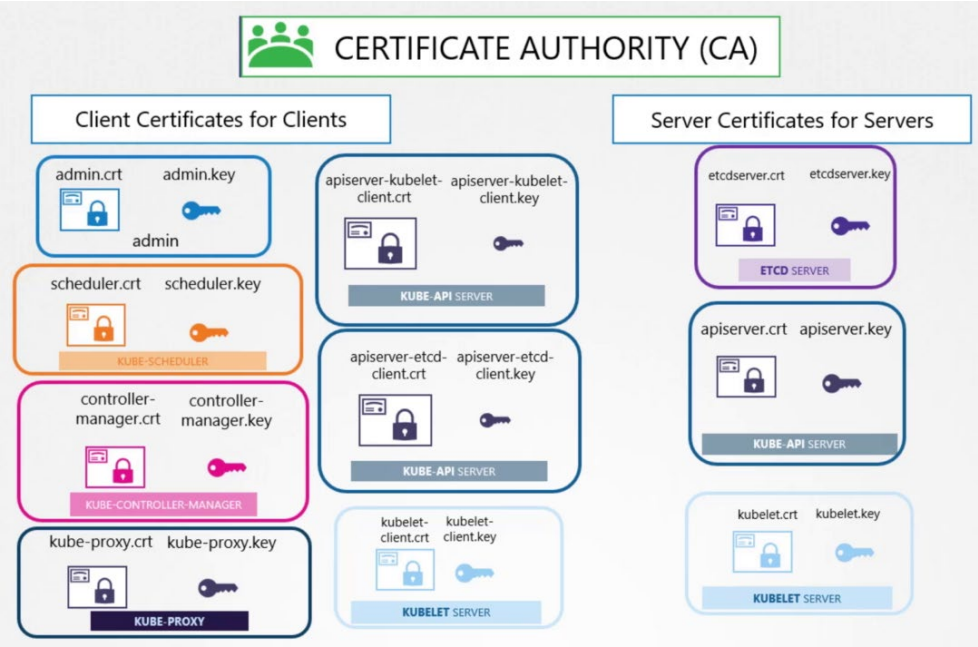
- الشهادات العامة (Public Certificates) : غالبًا تنتهي بـ crt أو pem
  - مثال server.crt , client.pem :
- المفاتيح الخاصة (Private Keys) : فيها كلمة key في الاسم أو الامتداد.
  - مثال server.key , server-key.pem :

دور الـCertificate Authority (CA)

- لازم يكون فيه CA واحد على الأقل للـ Cluster
- الـ CA نفسه عنده:
  - ca.crt (Public)
  - ca.key (Private)
- كل الشهادات الباقية بتتوقّع منه.

في بعض الحالات ممكن يكون فيه أكتر من CA :

- واحد عام للـ Cluster
- واحد خاص لـ etcd فقط.



خطوات عملية لتوليد الشهادات

❖ **توليد Root CA :**

```
openssl genrsa -out ca.key 2048
openssl req -x509 -new -nodes -key ca.key -subj "/CN=kubernetes-ca" -days 10000 -out ca.crt
```

❖ **توليد شهادة لـ apiserver :**

```
openssl genrsa -out apiserver.key 2048
openssl req -new -key apiserver.key -subj "/CN=kube-apiserver" -out apiserver.csr
openssl x509 -req -in apiserver.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out apiserver.crt -days 1000
```

❖ **مثال لشهادة Admin :**

```
openssl genrsa -out admin.key 2048
openssl req -new -key admin.key -subj "/CN=admin/O=system:masters" -out admin.csr
openssl x509 -req -in admin.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out admin.crt -days 1000
```

TLS In K8S – Certificate Creation

📌 الفكرة العامة

الدرس بيتكلم عن توليد الشهادات (TLS Certificates) في Kubernetes باستخدام أداة OpenSSL
هنمشي بالترتيب:

1. نبدأ بالـ Certificate Authority (CA)
2. نعمل شهادات Clients زي (admin, scheduler, controller, proxy)
3. نعمل شهادات Servers (ETCD, API server, Kubelet)
4. نشوف إزاي نربطهم داخل ملفات الإعدادات (config files)

🟢 إنشاء شهادة الـ CA (Root Certificate)

أول حاجة محتاجين CA يوقّع باقي الشهادات.

الخطوات العملية:

1. توليد الـ Private Key : `openssl genrsa -out ca.key 2048`
   - ده بينشئ مفتاح خاص (Private Key) للـ CA
2. إنشاء CSR (طلب توقيع شهادة) :

```
openssl req -new -key ca.key -subj "/CN=Kubernetes-CA" -out ca.csr
```

   - هنا CN = Common Name سميناه Kubernetes-CA
3. توقيع الشهادة بنفسها (Self-Signed) :

```
openssl x509 -req -in ca.csr -signkey ca.key -out ca.crt
```

- كده عندنا ca.key (المفتاح) و ca.crt (الشهادة)
- أي شهادة تانية هنعملها هتتوقّع بالـ CA ده.

🟢 إنشاء شهادة للـ Admin اللي بيستخدم kubectl

1. نعمل المفتاح الخاص: `openssl genrsa -out admin.key 2048`
2. نعمل CSR ونحدد الـ CN و الـ Group :

```
openssl req -new -key admin.key \
  -subj "/CN=kube-admin/OU=system:masters"  -out admin.csr
```

   - CN=kube-admin ده اسم المستخدم.
   - OU=system:mastersمعناها إن المستخدم ده في جروب الـ Admins اللي ليهم كل الصلاحيات.
3. نوقّع الشهادة بالـ CA :

```
openssl x509 -req -in admin.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out admin.crt
```

   - النتيجة : admin.key + admin.crt
   - زي كأننا عملنا User Account للـ admin لكن بطريقة Certificates

---

شهادات بقية الـ Clients

بنكرر نفس الخطوات (مفتاح ← CSR ← توقيع بالـ CA)
مع اختلاف الاسم:

❖ **kube-scheduler**

```
openssl genrsa -out scheduler.key 2048
openssl req -new -key scheduler.key -subj "/CN=system:kube-scheduler" -out scheduler.csr
openssl x509 -req -in scheduler.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out scheduler.crt
```

❖ **kube-controller-manager**

```
openssl genrsa -out controller-manager.key 2048
openssl req -new -key controller-manager.key -subj "/CN=system:kube-controller-manager" -out controller-manager.csr
openssl x509 -req -in controller-manager.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out controller-manager.crt
```

❖ **kube-proxy**

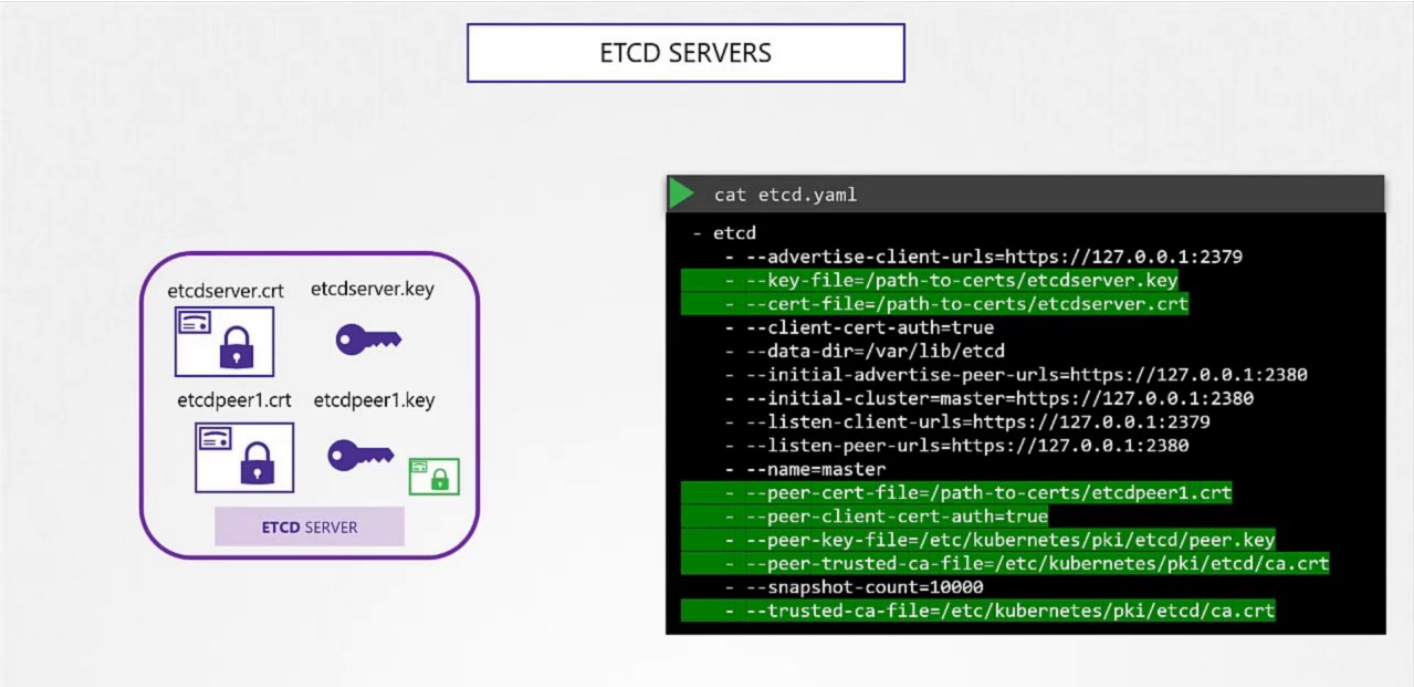
```
openssl genrsa -out kube-proxy.key 2048
openssl req -new -key kube-proxy.key -subj "/CN=system:kube-proxy" -out kube-proxy.csr
openssl x509 -req -in kube-proxy.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out kube-proxy.crt
```

شهادات الـ Servers

❖ **ETCD Server**

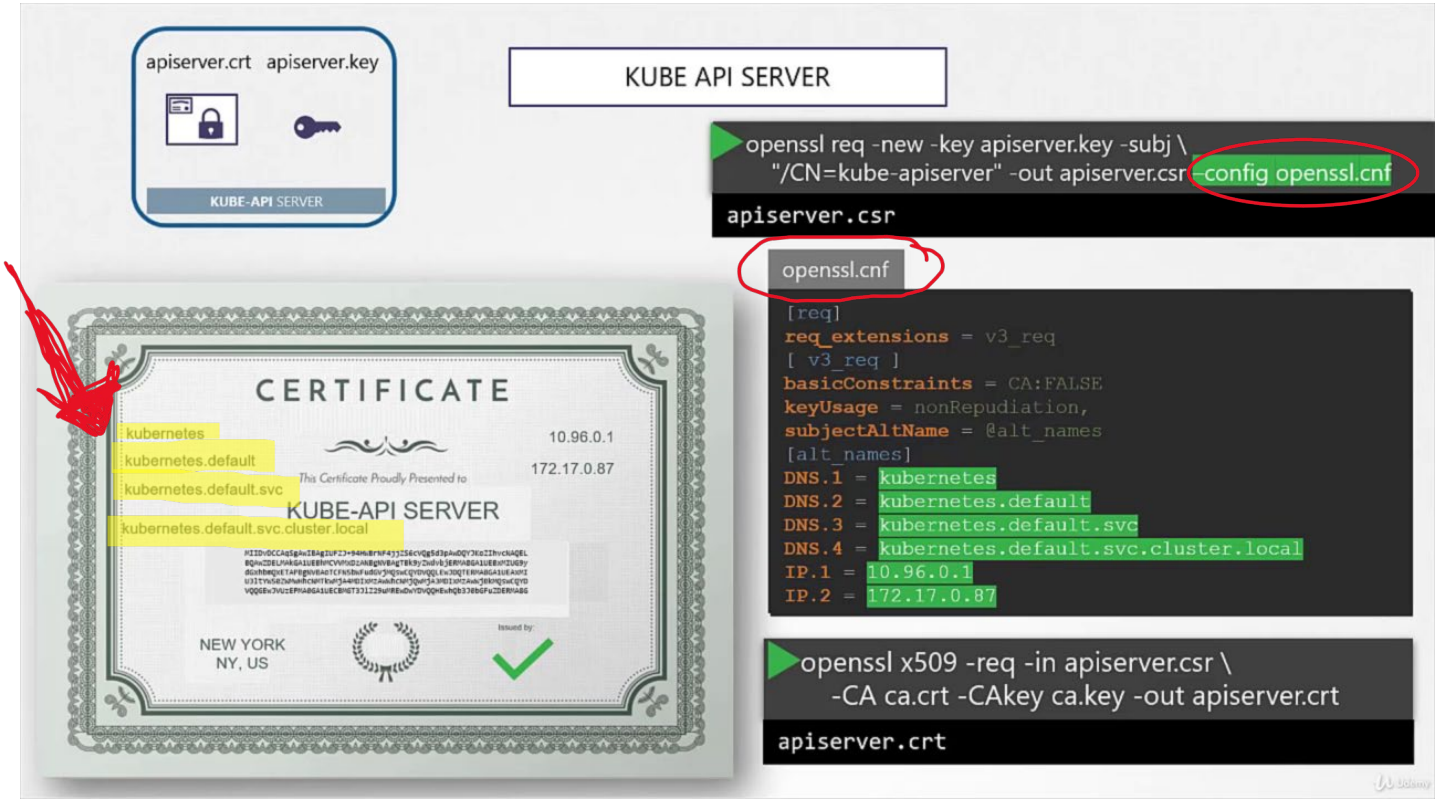
```
openssl genrsa -out etcd.key 2048
openssl req -new -key etcd.key -subj "/CN=etcd-server" -out etcd.csr
openssl x509 -req -in etcd.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out etcd.crt
```

في حالة Clustered ETCD محتاجين كمان Peer Certificates للتواصل بين النودز.



❖ API Server

ده أهم مكون ومحتاج أسماء متعددة (DNS + IP) فلازم نعمل ملف config :



الخطوات:

```
openssl genrsa -out apiserver.key 2048
openssl req -new -key apiserver.key -out apiserver.csr -config openssl.cnf
openssl x509 -req -in apiserver.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out apiserver.crt -extensions v3_req -extfile openssl.cnf
```

❖ Kubelet(لكل Node)

كل Node بياخد شهادة باسم مختلف:

مثال على Node01 :

```
openssl genrsa -out node01.key 2048
openssl req -new -key node01.key -subj "/CN=system:node:node01/OU=system:nodes" -out node01.csr
openssl x509 -req -in node01.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out node01.crt
```

- لاحظ CN = system:node:node01
- OU = system:nodes

نكررها لكل Node (node02, node03...).

استخدام الشهادات

- كل Client (admin, scheduler, controller, proxy, kubelet) يحط ملفاته (crt + key + ca.crt) في الـ kubeconfig الخاص بي. (هايتشرح قدام)
- كل Server (apiserver, etcd, kubelet) ياخد شهاداته ويتحطوا في إعداداته (--tls-cert-file, --tls-private-key-file, --client-ca-file)

🟡 **إضافات مفيدة:**

- **عشان تشوف محتوى أي شهادة:**

```
openssl x509 -in admin.crt -text -noout
```

- **لو الشهادات هتنتهي (expiry) لازم تعمل renew قبلها.**
- **ممكن تستخدم أدوات أسهل زي cfssl أو حتى kubeadm اللي بيولّد الشهادات تلقائي.** (هايشرح قدام)

View Certificate Details with Troubleshooting

📌 الفكرة العامة

الدرس بيتكلم عن إزاي تفحص وتراجع الشهادات (Certificates) في Cluster قائم.

السيناريو: انت انضمّيت لفريق جديد، فيه مشاكل متعلقة بالشهادات، ومطلوب منك تعمل Health Check لكل شهادات الكلاستر.

🟢 لازم تعرف الكلاستر اتعمل إزاي

- لو الكلاستر معمول Manually (The Hard Way) انت بنفسك بتكون مولّد كل الشهادات بالـ OpenSSL أو أدوات مشابهة.
- لو معمول باستخدام kubeadm فهو بيولّد ويدير الشهادات تلقائي.
  - في الـHard way : المكونات (API, etcd, scheduler...) بتشتغل كـ Services في الـ OS
  - في kubeadm : المكونات الأساسية بتشتغل كـ Pods جوه الـ Cluster

✅ أول خطوة مهمة: لازم تفهم الكلاستر اتبنى إزاي علشان تعرف فين تدور على الشهادات.

"The Hard Way"

```
cat /etc/systemd/system/kube-apiserver.service
[Service]
ExecStart=/usr/local/bin/kube-apiserver \\
  --advertise-address=172.17.0.32 \\
  --allow-privileged=true \\
  --apiserver-count=3 \\
  --authorization-mode=Node,RBAC \\
  --bind-address=0.0.0.0 \\
  --client-ca-file=/var/lib/kubernetes/ca.pem \\
  --enable-swagger-ui=true \\
  --etcd-cafile=/var/lib/kubernetes/ca.pem \\
  --etcd-certfile=/var/lib/kubernetes/kubernetes.pem \\
  --etcd-keyfile=/var/lib/kubernetes/kubernetes-key.pem \\
  --event-ttl=1h \\
  --kubelet-certificate-authority=/var/lib/kubernetes/ca.pem \\
  --kubelet-client-key=/var/lib/kubernetes/kubernetes-key.pem \\
  --kubelet-https=true \\
  --service-node-port-range=30000-32767 \\
  --tls-cert-file=/var/lib/kubernetes/kubernetes.pem \\
  --tls-private-key-file=/var/lib/kubernetes/kubernetes-key.pem
  --v=2
```

kubeadm

```
cat /etc/kubernetes/manifests/kube-apiserver.yaml
spec:
  containers:
  - command:
    - kube-apiserver
    - --authorization-mode=Node,RBAC
    - --advertise-address=172.17.0.32
    - --allow-privileged=true
    - --client-ca-file=/etc/kubernetes/pki/ca.crt
    - --disable-admission-plugins=PersistentVolumeLabel
    - --enable-admission-plugins=NodeRestriction
    - --enable-bootstrap-token-auth=true
    - --etcd-cafile=/etc/kubernetes/pki/etcd/ca.crt
    - --etcd-certfile=/etc/kubernetes/pki/apiserver-etcd-client.crt
    - --etcd-keyfile=/etc/kubernetes/pki/apiserver-etcd-client.key
    - --etcd-servers=https://127.0.0.1:2379
    - --insecure-port=0
    - --kubelet-client-certificate=/etc/kubernetes/pki/apiserver-kubelet-client.crt
    - --kubelet-client-key=/etc/kubernetes/pki/apiserver-kubelet-client.key
    - --kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname
    - --proxy-client-cert-file=/etc/kubernetes/pki/front-proxy-client.crt
    - --proxy-client-key-file=/etc/kubernetes/pki/front-proxy-client.key
    - --secure-port=6443
    - --service-account-key-file=/etc/kubernetes/pki/sa.pub
    - --service-cluster-ip-range=10.96.0.0/12
    - --tls-cert-file=/etc/kubernetes/pki/apiserver.crt
    - --tls-private-key-file=/etc/kubernetes/pki/apiserver.key
```

🟢 حصر كل الشهادات

- الهدف : تعمل قائمة (Inventory) تشمل:
  - اسم الملف (certificate file)
  - المكون اللي بيستخدمه (API server, etcd...)
  - CN (Common Name)
  - SANs (Subject Alternative Names) → DNS / IPs
  - الـ Organization
  - الـ Issuer (الشخص اللي وقّع الشهادة)
  - تاريخ الانتهاء.

📊 ممكن تعمل ده في Excel Sheet علشان تبقى الصورة أوضح.

مكان الشهادات في kubeadm

- في الكلاستر اللي معمول بـ kubeadm :
  تروح على : etc/kubernetes/manifests/kube-apiserver.yaml/
  - هتلاقي الـ Pod Definition للـ kube-apiserver
  - جواه : أوامر التشغيل (command args) بتوضح أماكن ملفات الشهادات زي:
    - --tls-cert-file
    - --tls-private-key-file
    - --client-ca-file

فحص الشهادة نفسها

لما تحدد ملف الشهادة (مثلاً شهادة الـ API Server) :

- أمر الفحص: openssl x509 -in /etc/kubernetes/pki/apiserver.crt -text -noout
- إيه اللي تبص عليه؟

1. Subject / CN : لازم يكون الاسم صحيح (مثلاً kube-apiserver)
2. SANs : لازم تحتوي كل الـ DNS والـ IP اللي الكلاستر بيستخدمه زي (kubernetes, kubernetes.default, 10.96.0.1 ... etc)
3. Validity : تاريخ البداية والانتهاء. لازم تتأكد إنها مش منتهية.
4. Issuer : لازم يكون الـ CA بتاع الكلاستر (عادة اسمه "kubernetes")

كرر الفحص لكل الشهادات

- apiserver.crt
- etcd.crt
- scheduler.crt
- controller-manager.crt
- kubelet (لكل نود شهادة)
- admin.crt
- proxy.crt

كل شهادة لازم تتحقق من:

- CN / Group (OU)
- Issuer
- Expiry
- SANs (لو مكون زي API Server)

أماكن الرجوع عند المشاكل

- لو الكلاستر معمول manually :
  - راجع الـ Service logs عن طريق الـ system logs (زي journalctl -u kube-apiserver)
- لو معمول بـ kubeadm يبقي المكونات عبارة عن Pods وبالتالي تقدر تستخدم :
  - kubectl logs -n kube-system <pod-name>
- لو المكون الأساسي (زي API Server أوETCD) واقع يبقي الـ kubectl مش هيشتغل.
  - في الحالة دي تنزل مستوى أقل (Docker أوContainer Runtime) عن طريق :
  - docker ps -a
  - docker logs <container-id>

إضافات مهمة

- الوثائق الرسمية Kubernetes Docs : فيها جدول يوضح كل شهادة، CN/SANs المطلوبة، والـ Expiry الافتراضي.
- kubeadm تجديد الشهادات:

```
kubeadm certs check-expiratio
kubeadm certs renew al
```

- تقدر تستخدم أمر سريع للـ:Expiry

```
openssl x509 -enddate -noout -in apiserver.crt
```

النتيجة بتكون مثلاً:

```
notAfter=Mar 15 12:00:00 2026 GMT
```

Certificates API

السيناريو قبل الـ Certificate API

- في الأول، الـ Admin (انت) عملت:
  - ولّدت CA (Certificate Authority) عبارة عن زوج ملفات ca.crt + ca.key :
  - عملت شهادات لباقي مكونات الكلاستر.(API Server, etcd, scheduler, controller, kubelet...)
- انت كـ Admin عندك شهادة خاصة بيك (admin.crt + admin.key)
- لما ييجي Admin جديد محتاج يدخل على الكلاستر:

1. هو بيعمل Private Key بنفسه : `openssl genrsa -out new-admin.key 2048`
2. يعمل CSR (Certificate Signing Request) :

```
openssl req -new -key new-admin.key -subj "/CN=new-admin/O=system:masters" -out new-admin.csr
```

3. يبعتهولك.
4. انت كـ Admin توقعه باستخدام الـ CA :

```
openssl x509 -req -in new-admin.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out new-admin.crt -days 365
```

5. ترجعله الشهادة الجديدة (new-admin.crt) ومعاها مفتاحه (new-admin.key)

- كده بقى عنده صلاحية يدخل على الكلاستر.
- لكن : كل شهادة ليها Expiry (مثلاً سنة واحدة).
- مع كل Expiry لازم نكرر نفس العملية (Sign ثم CSR ثم رجوع الشهادة)

يعني إيه CA Server ؟

- CA Server مش جهاز ضخم زي ما ممكن تتخيل لكنه مجرد ملفات ca.crt و ca.key
- أي حد معاه الملفات دي يقدر يوقّع شهادات ويدخل الكلاستر بصلاحيات Admin
- عشان كده:
  - لازم تتحفظ في مكان آمن جدًا.
  - في العادة بيتخزنوا على الـ Master Node ولو استخدمت kubeadm برضه بيحطهم هناك تحت /etc/kubernetes/pki وبالتالي ده معناه إن Master Node بيقوم بدور CA Server في الكلاستر.

المشكلة مع الـ Manual Signing

- مع زيادة عدد الـ Users والـ Admins
  - هتلاقي نفسك كل يوم بتوقّع CSR جديد.
  - وهتلاقي نفسك بتدوّر على Expired Certificates عشان تجددها.

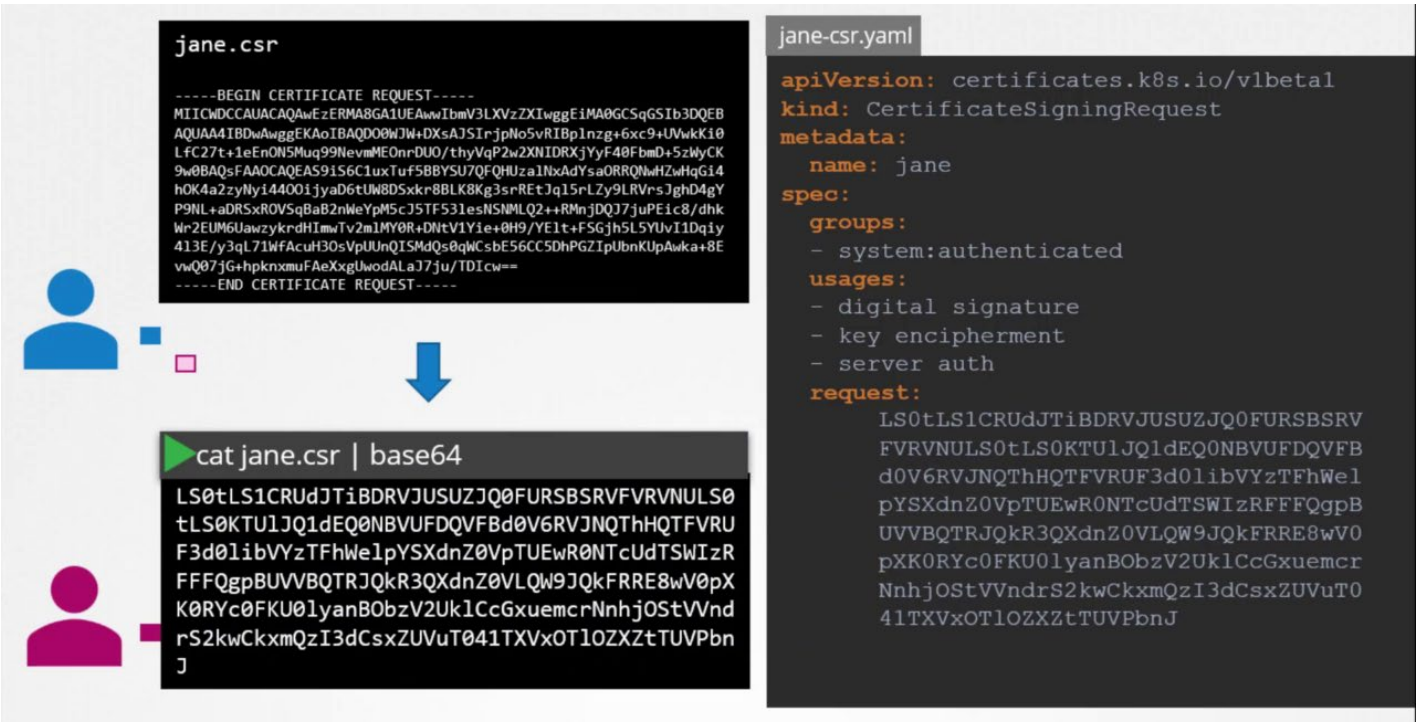
الحل Kubernetes عنده Certificates API

Certificate API

- بدل ما المستخدم يبعت CSR يدوي، يقدر يبعتها مباشرة للكلاستر كـ Kubernetes Object

الخطوات:

3. يحوّل CSR لقاعدة 64: cat dev-user.csr | base64



- الـ controller-manager فيه Controllers مخصوصة:
  - csr-approving مسؤولة عن عملية الـ Approval
  - csr-signing مسؤولة عن عملية التوقيع باستخدام CA
- وعلشان يقدر يوقع: لازم يكون عنده وصول لـ :
  - ca.crt
  - ca.key

وده بيتحدد في إعدادات تشغيل الـ controller-manager

Kube Config

مقدمة

- إحنا عرفنا قبل كده إزاي نولد شهادة (Certificate) لمستخدم، وإزاي يقدر يستعملها مع المفتاح (Key) عشان يتكلم مع Kubernetes API باستخدام أدوات زي curl
- مثلًا لو عندك كلاستر اسمه my-kube-playground، ممكن تبعت Request كده:

```
curl https://<API_SERVER>:6443/api/v1/pods --key admin.key --cert admin.crt  --cacert ca.crt
```

- الـ API Server هيشوف المفاتيح والشهادات ويتأكد من صحتها وي Authenticates المستخدم.

المشكلة مع الطريقة دي

- ينفع تعمل نفس الكلام مع kubectl باستخدام Options زي:
  - --server
  - --client-key
  - --client-certificate
  - --certificate-authority
- لكن: كل مرة هتكتبهم يدويًا وبالتالي الموضوع مرهق .

الحل KubeConfig

- عشان كده اتعمل ملف KubeConfig
- kubectl بشكل افتراضي بيدور على الملف:

  $HOME/.kube/config
- لو الملف موجود هناك، مش محتاج تضيف Options في كل أمر.

مكونات KubeConfig

الملف بيتكتب بصيغة YAML ، وبيكون فيه 3 أقسام رئيسية:

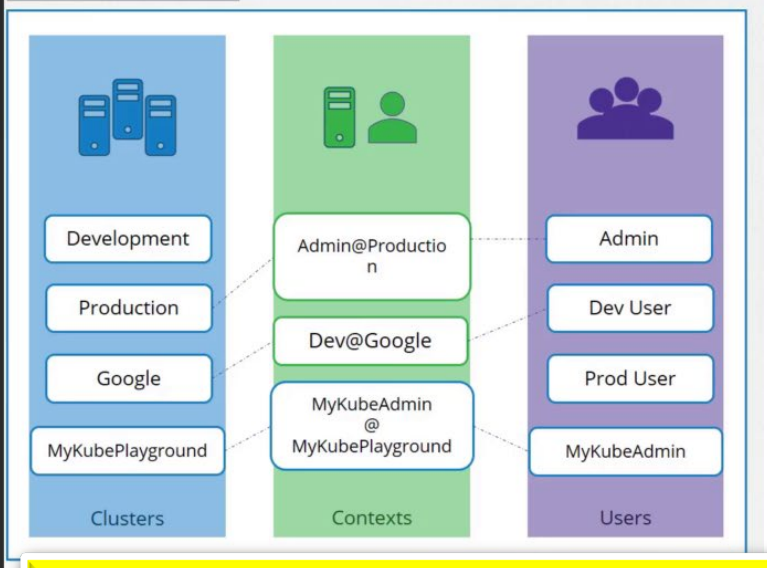
1. clusters : تعريف الكلاسترات اللي عندك.(Dev, Test, Prod, GKE, EKS...)
2. users : حسابات المستخدمين وصلاحياتهم.(admin, dev-user, prod-user...)
3. contexts : بيربط مستخدم بكلاستر محدد.

مثال عملي على KubeConfig

KubeConfig File

```yaml
apiVersion: v1
kind: Config

clusters:
- name: my-kube-playground
  cluster:
    certificate-authority: ca.crt
    server: https://my-kube-playground:6443

contexts:
- name: my-kube-admin@my-kube-playground
  context:
    cluster: my-kube-playground
    user: my-kube-admin

users:
- name: my-kube-admin
  user:
    client-certificate: admin.crt
    client-key: admin.key
```

$HOME/.kube/config

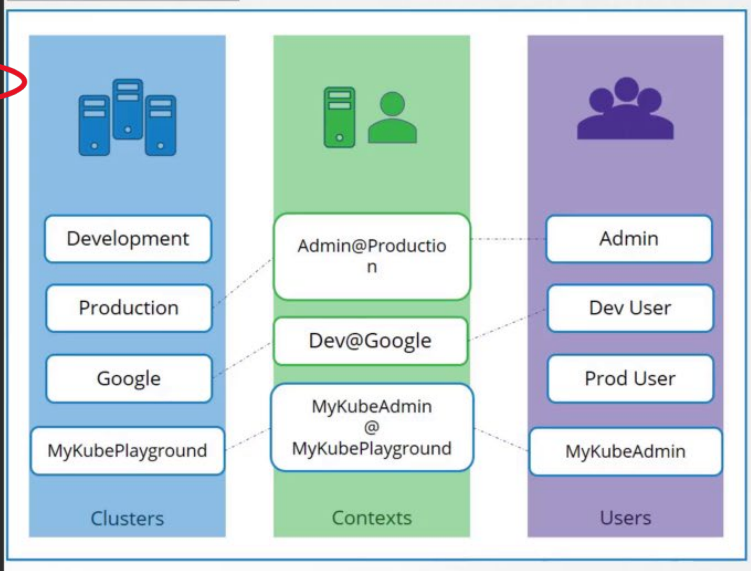


الفايل الاولاني هو هواه الفايل الثاني بس الفكرة اني في الفايل التاني خفيت الـ values بتاع كل الـ cluster or context or users علشان بس أوضحلك حتة اني ممكن اعمل كذا cluster or context or users في الفايل الواحد بس بشرط اني احددله الـ current context في الفايل

KubeConfig File

```yaml
apiVersion: v1
kind: Config
current-context: dev-user@google
clusters:
- name: my-kube-playground   (values hidden...)
- name: development
- name: production
- name: google

contexts:
- name: my-kube-admin@my-kube-playground
- name: dev-user@google
- name: prod-user@production

users:
- name: my-kube-admin
- name: admin
- name: dev-user
- name: prod-user
```

$HOME/.kube/config



أوامر عملية لإدارة KubeConfig

- عرض محتوى KubeConfig : kubectl config view
- تغيير الـ Context الحالي: kubectl config use-context prod-user@production للـ Prod
- عرض الـ Context الحالي: kubectl config current-context
- إضافة Cluster جديد:

```
kubectl config set-cluster dev-cluster \
 --server=https://192.168.1.101:6443 \
 --certificate-authority=ca.crt
```

- إضافة User جديد:

```
kubectl config set-credentials dev-user \
 --client-certificate=dev-user.crt \
 --client-key=dev-user.key
```

- إنشاء Context جديد:

```
kubectl config set-context dev@dev-cluster \
 --cluster=dev-cluster \
 --user=dev-user
```

التعامل مع الـ Namespaces في KubeConfig

- ممكن تحدد Namespace افتراضي جوه Context :

```yaml
contexts:
- name: dev@dev-cluster
  context:
    cluster: dev-cluster
    user: dev-user
    namespace: dev-namespace
```

- لما تستخدم الـ Context ده، أوامر kubectl هتشتغل تلقائيًا على الـ Namespace ده من غير ما تضيف -n

---

الشهادات في KubeConfig

- ممكن تحدد الملفات مباشرة : certificate-authority: /etc/kubernetes/pki/ca.crt
- أو تحط محتوى الشهادة Base64 Encoded :

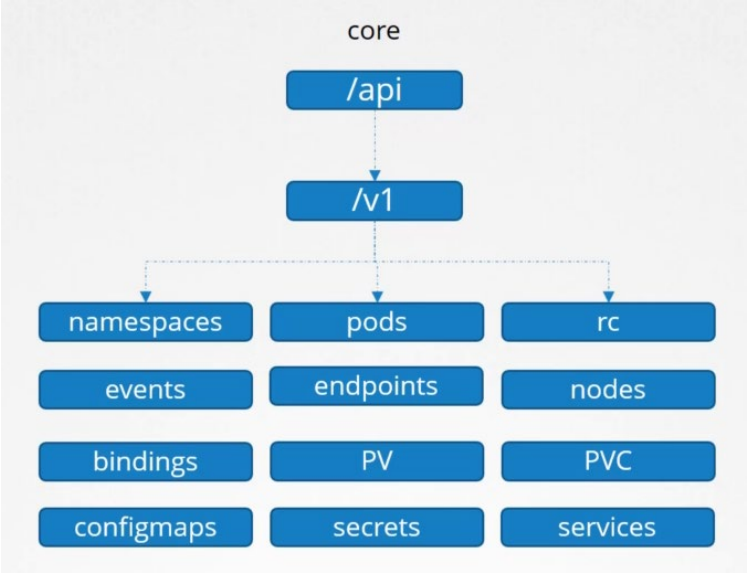
```
certificate-authority-data: <BASE64-ENCODED-CA>
```

- نفس الكلام ينطبق على مفاتيح المستخدمين (client-certificate-data, client-key-data)

API Groups

ايه هو Kubernetes API ؟

- أي عملية عملتها لحد دلوقتي مع الكلاستر (زي kubectl get pod) في الحقيقة هي Request للـ Kube-API Server.
- الـ API Server بيشتغل على Port 6443 (الافتراضي)
- مثال:
  - تجيب نسخة الكلاستر: curl -k https://<MASTER-IP>:6443/version
  - تجيب الـ Pods : curl -k https://<MASTER-IP>:6443/api/v1/pods



الـ Verbs

- لكل Resource في الـ API فيه Actions مسموح بيها.
- مثالDeployment :
  - get : تعرض Deployment
  - list : تعرض كل Deployments
  - create : تعمل Deployment جديد
  - update : تعدل في Deployment
  - delete : تمسح Deployment
  - watch : تتابع أي تغيير بيحصل

ملحوظة مهمة kube-proxy vs kubectl proxy : (سؤال انترفيو)

- **kube-proxy:**
  - **مكون أساسي في الكلاستر.**
  - **وظيفته إنه يدير Networking بين الـ Pods والـ Services على كل Node**
- **kubectl proxy:**
  - **Service بتشغله يدويًا عشان تتعامل مع الـ API**
  - **Proxy عادي HTTP بيحول Requests للـ API Server باستخدام الـ Credentials بتاعتك.**

Role Based Access Controls (RBAC)

يعني إيه RBAC ؟

- هو Role-Based Access Control يعني نظام لتحديد الصلاحيات (permissions) للمستخدمين والـ service accounts داخل الكلاستر.
- بيشتغل عن طريق:
  - Roles : بتحدد الصلاحيات.
  - RoleBindings : بتربط مستخدم أو مجموعة بصلاحيات معينة.

إنشاء Role

- الـ Role عبارة عن Object بيتكتب في YAML

developer-role.yaml

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: developer
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["list", "get", "create", "update", "delete"]
- apiGroups: [""]
  resources: ["ConfigMap"]
  verbs: ["create"]
```

📌 هنا:

- المستخدم يقدر يعمل Get/List/Create/Delete على الـ Pods
- ويقدر يعمل Create لـ ConfigMaps
- إنشاء الـ Role عمليًا: kubectl create -f developer-role.yaml

---

ربط User بالـ Role عن طريق (RoleBinding)

- عشان تدي الصلاحيات ليوزر، لازم تعمل RoleBinding :

devuser-developer-binding.yaml

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: devuser-developer-binding
subjects:
- kind: User
  name: dev-user
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: developer
  apiGroup: rbac.authorization.k8s.io
```

- إنشاء الـ RoleBinding :

kubectl create -f devuser-binding.yaml

📌 كده الـ dev-user هيبقى عنده صلاحيات الـ developer role

النطاق (Scope)

- Roles & RoleBindings دايمًا على مستوى Namespace
- يعني الـ dev-user هنا واخد الصلاحيات دي في default namespace بس.
- لو عاوز Namespace مختلف .. بتغير الـ metadata.namespace في الـ YAML

أوامر لفحص الـ Roles

- عرض كل الـ Roles : kubectl get roles -n default
- عرض كل الـ RoleBindings : kubectl get rolebindings -n default
- تفاصيل Role : kubectl describe role developer -n default
- تفاصيل RoleBinding : kubectl describe rolebinding devuser-developer-binding -n default

---

Check Access (اختبار الصلاحيات) (Auth Can-I)

- تشوف إذا كان يوزر يقدر يعمل حاجة:

```
kubectl auth can-i create pods --as dev-user -n default
kubectl auth can-i delete nodes --as dev-user
```

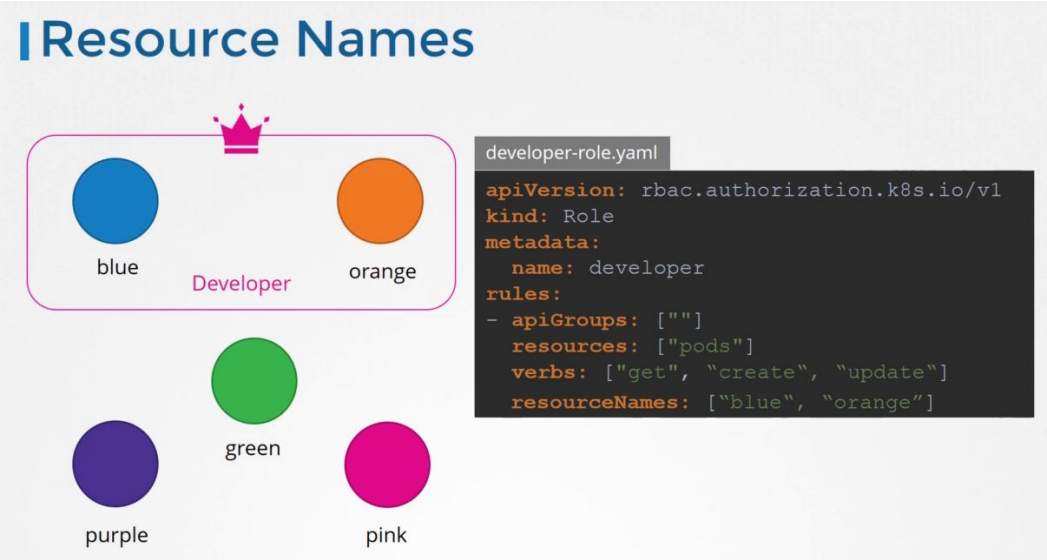
- لو معندوش الصلاحية .. هايطلعلك no
- لو عنده .. هايطلعلك Yes

📌 كده الـ Admin يقدر يختبر صلاحيات أي يوزر من غير ما يعمل Login بيه.

تخصيص صلاحيات لموارد معينة (ResourceNames)

- ممكن تديله صلاحيات على Pod معين مش على كل الـ Pods :

📌 هنا اليوزر يقدر يتحكم بس في blue-pod و orange-pod



الخلاصة

- الـ Role يحدد الصلاحيات.
- الـ RoleBinding يربط اليوزر بالـ Role
- النطاق (Scope) دايمًا Namespace
- ممكن تختبر الصلاحيات بـ kubectl auth can-I
- ممكن تحدد موارد معينة بالـ resourceNames

Image Security

مقدمة

- طول الكورس استخدمنا Images جاهزة.(nginx, redis, web apps, databases ...)
- الصور دي افتراضيًا بتتسحب من Docker Hub أو أي Registry عام
- لكن في الواقع:
  - أحيانًا بنحتاج نستخدم Images خاصة بالشركة (In-house apps)
  - الصور دي ماينفعش تكون Public
  - هنا بييجي دور Private Registries + التحكم في الـ Credentials

---

(Image Names)

nginx-pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
spec:
  containers:
  - name: nginx
    image: nginx
```

- كلمة nginx هنا مش مجرد اسم، دي بتمشي حسب Docker image naming convention
  - الصيغة الكاملة : [registry-url]/[username]/[repository]:[tag]
  - مثال كامل : docker.io/nginx/nginx:latest
- لو كتبت بس nginx بيتحول تلقائيًا إلى: docker.io/nginx/nginx:latest
- يعني:
  - docker.io هو الـ Registry (افتراضي DockerHub)
  - nginx/nginx أول جزء اسم الحساب (username) ، التاني اسم الـ repo
  - latest الـ Tag الافتراضي.

الـ Registries المختلفة

- Docker Hub → docker.io
- Google Container Registry (GCR) → gcr.io
- AWS Elastic Container Registry (ECR) : خاص بأكاونت AWS
- Azure Container Registry (ACR)
- GitHub Container Registry → ghcr.io
- ممكن كمان تبني Private Registry داخلي على سيرفر الشركة.

📌 الفكرة: أي صورة مش Public .. محتاجة Authentication قبل السحب.

استخدام صور من Private Registry

- في الـ Docker CLI :

```
docker login <registry-url>
docker run <registry-url>/<username>/<image>:<tag>
```

- في Kubernetes الموضوع بيختلف شوية، لإن الـ Pod نفسه اللي بيطلب الصورة، والكوبليت (kubelet) في الـ Worker Node هو اللي بيسحبها . يعني المشكلة دلوقت هاتبقي اني ازاي أعمل الـ Authentication بمعني اخر ازاي الـ K8S بتجيب الـ credentials علشان تـ Access الـ Private Registry ؟؟؟؟

الحل في الـ Kubernetes هو ImagePullSecrets

How do you pass the credentials to the docker runtime on the worker node ??

1. إنشاء Secret من نوع Docker Registry :

```
kubectl create secret docker-registry regcred  --docker-server=<registry-url> \
--docker-username=<username>  --docker-password=<password>  --docker-email=<email>
```

📌 هنا:

- regcred يعبر عن اسم الـ Secret
- بيتخزن فيه الـ credentials (username/password/email/registry)

2. استخدام الـ Secret في Pod

nginx-pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
spec:
  containers:
  - name: nginx
    image: private-registry.io/apps/internal-app
  imagePullSecrets:
  - name: regcred
```

📌 عند تشغيل الـ Pod

- الكوبليت (kubelet) على الـ Worker node يستخدم الـ Secret
- يعمل Login للـ Registry
- يسحب الصورة ويشغلها.

- أوامر عملية مهمة

- عرض الـ Secrets : kubectl get secrets
- وصف Secret : kubectl describe secret regcred
- تحديث Secret (في حالة لو غيرت الباسورد في الـ Registry)

kubectl delete secret regcred

kubectl create secret docker-registry regcred ...

- استخدام Secret مع ServiceAccount (مفيد للـ CI/CD)

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
name: deployer
imagePullSecrets:
- name: regcred
```

وبعدها أي Pod بيشتغل بالـ ServiceAccount ده هيستخدم الـ Secret تلقائيًا.

إضافات مهمة للأمان

- Scanning Images : استخدم أدوات زي trivy أو anchore للكشف عن ثغرات في الصور.
- استخدام Signed Images : عشان تتأكد إن الصورة مش متعدَّل عليها.
- Policy Enforcement :
  - أدوات زي OPA/Gatekeeper تمنع أي Pod يستخدم Image من Registry غير مصرح بيه.
    مثال : السماح فقط بالصور من myregistry.com

الخلاصة

- الاسم البسيط nginx وراه Registry افتراضي (docker.io)
- ممكن تستخدم أي Registry تاني (AWS, GCP, Azure, Private)
- لو الصورة Private يبقي لازم Secret من نوع Docker Registry
- الـ Pod بيستخدم Secret ده عشان يسحب الصورة.
- ممكن توصل لمرحلة أمان أعلى بإضافة Scanning + Signed Images + Policies

Security Context

المقدمة

- زي ما اتكلمنا في Docker: تقدر تتحكم في إعدادات الأمان (security standards) الخاصة بالـ Container
- أمثلة:
  - تحديد الـ User IDاللي الـ Container هيشتغل بيه) بدل ما يشتغل كـ.(Root
  - تحديد أو إزالة (Linux Capabilitiesقدرات خاصة زي NET_ADMINأو .SYS_TIME)
- في Kubernetes تقدر تعمل نفس الكلام ده.

المستويات (Pod vs Container)

- في Kubernetes : الـ Containers دايمًا بيشتغلوا جوة Pods
- تقدر تضبط الـ Security Context على :
  1. مستوى الـ Pod : كل الـ Containers جوة الـ Pod يرثوا الإعدادات.
  2. مستوى الـ Container : ينطبق بس على الـ Container ده، ولو فيه تضارب .. فإعدادات الـ Container هو اللي بيتطبق (بيـ Override)

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: ubuntu-pod
spec:
  securityContext:              # 👈 Security Context على مستوى الـ Pod
    runAsUser: 1000             # الـ Pod يشتغل كـ User ID = 1000
  containers:
  - name: ubuntu
    image: ubuntu
    command: ["sleep", "3600"]
    securityContext:            # 👈 Security Context على مستوى الـ Container
      runAsUser: 2000           # الـ Container ده يشتغل كـ User ID = 2000 (Override هنا بيعمل)
      capabilities:             # إضافة قدرات خاصة للـ Container
        add: ["NET_ADMIN", "SYS_TIME"]
```

📌 **ملاحظات:**

- runAsUser: 1000 :: كل الـ Containers هتشتغل بالـ User ID ده، إلا لو Container معين اتحدد له غير كده.
- في المثال ده:
  - الـ Pod اتضبط على 1000
  - الـ "Container "ubuntu اتضبط على .2000
  - النتيجة: الـ "Container "ubuntu هيشتغل بالـ User ID 2000
- capabilities : بيديك صلاحيات محددة من الـ Kernel للـ Container
  - مثال NET_ADMIN: عشان تقدر تعمل إعدادات Networking زي تغيير IP tables
  - مثال SYS_TIME: عشان تقدر تغير توقيت النظام.

الدخول جوة الـ Pod والتأكد من الـ User ID :: kubectl exec -it ubuntu-pod -- id

هاتشوف :: uid=2000 gid=2000 groups=2000

إضافات مهمة

1. **readOnlyRootFilesystem**
   - تخلي الـ Container Filesystem للقراءة فقط (مفيد للأمان).

```yaml
securityContext:
  readOnlyRootFilesystem: true
```

2. **runAsNonRoot**
   - تتأكد إن الـ Container مش بيشتغل كـ Root بأي شكل .

```yaml
securityContext:
  runAsNonRoot: true
```

3. **Drop Capabilities**
   بدلاً من إضافة قدرات، ممكن تشيل قدرات مش محتاجها (وده أكثر أمانًا).

```yaml
securityContext:
  capabilities:
    drop: ["ALL"]   # تشيل كل القدرات
```

4. **Privileged Mode (خطر جدًا)**
   يخلي الـ Container يشتغل زي Root كامل على الـ Node.
   ⚠️ ممنوع تستخدمه إلا لو في حالة خاصة جدًا.

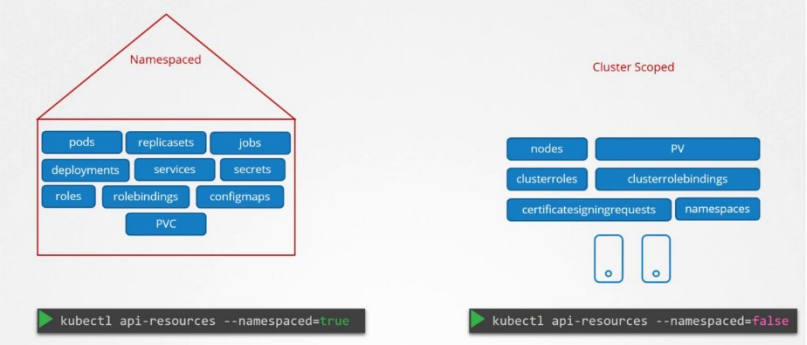
```yaml
securityContext:
  privileged: true
```

Cluster Role and Role Binding

مراجعة سريعة من المحاضرة السابقة

- اتكلمنا قبل كده عن Role و RoleBinding
  - بيكونوا Namespaced يعني شغالين داخل Namespace محدد.
  - لو معملتش Specify للـ Namespace محددة .. فبيتعملوا في default namespace
- بيستخدموا عشان تدير صلاحيات الموارد (Pods, Deployments, Services ...) داخل Namespace واحد.

المشكلة مع Resources مش مرتبطة بـ Namespace



- فيه موارد في الكلاستر مش بتتبع Namespace ، زي:
  - Nodes
  - Persistent Volumes (PV)
  - Namespaces نفسها
  - CertificateSigningRequests
- الموارد دي اسمها Cluster-scoped resources
- بالتالي الـ Roles العادية مش هتنفع معاها .

الحل ClusterRole

- ClusterRole زي الـ Role بالظبط لكن بيطبق على مستوى الكلاستر كله.
- تقدر تستخدمه عشان تدي صلاحيات:
  - على موارد Cluster-wide زي Nodes, PVs
  - أو حتى على موارد Namespaced لكن Across كل الـ Namespaces

مثال عملي ClusterRole :

- نفترض عايز تعمل ClusterRole لإدارة الـ Nodes :

📌 هنا:

- اليوزر اللي هيرتبط بالـ ClusterRole ده يقدر يعمل إدارة كاملة للـ Nodes
- إنشاء ClusterRole :

kubectl create -f cluster-admin-role.yaml

cluster-admin-role.yaml

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: cluster-administrator
rules:
- apiGroups: [""]
  resources: ["nodes"]
  verbs: ["list", "get", "create", "delete"]
```

ClusterRoleBinding

- زي الـ RoleBinding ، لكنه بيربط ClusterRole بمستخدم أو جروب أو ServiceAccount

cluster-admin-role-binding.yaml

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: cluster-admin-role-binding
subjects:
- kind: User
  name: cluster-admin
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: cluster-administrator
  apiGroup: rbac.authorization.k8s.io
```

- تطبيق الـ Binding ::

kubectl create -f cluster-admin-role-binding.yaml

📌 كده الـ user اللي اسمه cluster-admin-user بقى عنده صلاحيات ClusterRole كاملة على الـ Nodes

ملحوظة مهمة

- مش شرط ClusterRoles تبقى بس للـ Cluster resources
- ممكن تدي Access على Pods أو ConfigMaps باستخدام ClusterRole .. بس في الحالة دي الصلاحيات هتبقى على كل الـ Namespaces مش Namespace واحد.
- مثال:

```yaml
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list"]
```

📌 ده معناه إن اليوزر يقدر يـ List/Get كل الـ Pods في أي Namespace

---

أوامر عملية مفيدة

- عرض كل الـ ClusterRoles : kubectl get clusterroles
- عرض كل الـ ClusterRoleBindings : kubectl get clusterrolebindings
- تفاصيل ClusterRole : kubectl describe clusterrole cluster-admin-role
- تفاصيل ClusterRoleBinding : kubectl describe clusterrolebinding cluster-admin-role-binding

---

الصلاحيات الافتراضية

- Kubernetes بيعمل شوية ClusterRoles افتراضية مع أول Setup
- زي:
  - cluster-admin : أعلى صلاحيات.
  - view : صلاحيات قراءة بس.
  - edit : صلاحيات تعديل موارد.Namespaced
- تقدر تعمل Bind لأي يوزر على طول للـ ClusterRoles دي بدل ما تعرّف واحدة جديدة.

الخلاصة

- Role + RoleBinding = صلاحيات جوة Namespace
- ClusterRole + ClusterRoleBinding = صلاحيات على مستوى الكلاستر كله أو Resources مش تبع Namespace
- ClusterRole ممكن يدي Access للـ Namespaced resources لكن Across كل الـ Namespaces
- فيه ClusterRoles افتراضية ممكن تستخدمها بدل ما تعمل من الصفر.

Network Policy

الأساسيات (Networking & Security Basics)

قبل ما ندخل في Kubernetes ، لازم نفهم أساسيات حركة المرور (Traffic) :

- عندك نظام مكوّن من:
  - Web Server : يستقبل طلبات المستخدمين على Port 80
  - API Server : يتعامل مع الـ Backend على Port 5000
  - Database Server : يتعامل مع البيانات على Port 3306

مسار الطلب:

1. المستخدم Web Server (Port 80)
2. Web Server → API Server (Port 5000)
3. API Server → Database Server (Port 3306)
4. Database Server يرجّع البيانات للـ API وبعدين يروح للمستخدم.

📌 فيه نوعين من الترافيك:

- Ingress = الترافيك اللي داخل الـ Pod
- Egress = الترافيك اللي طالع من الـ Pod

⚠️ الرد (response) مش بيتحسب في Ingress/Egress rules لكن المهم هو اتجاه الترافيك الأساسي.

Kubernetes Networking by Default

- في Kubernetes ، كل Pod بياخد IP Address خاص.
- كل Pods يقدروا يتواصلوا مع بعض افتراضيًا بدون أي قيود.
- بمعنى : أي Pod يقدر يكلم أي Pod تاني (All Allow Rule)

🔎 المشكلة : ده خطر أمني، لأن ممكن الـ Web Pod يكلم الـ Database Pod مباشرةً من غير المرور بالـ API وده مخالف لقواعد الأمان.

الحل Network Policies

- NetworkPolicy هو Object في Kubernetes بيحدد مين يقدر يكلم مين وعلى أي Ports
- زي الـ Firewall rules لكن جوة Kubernetes

📌 الفكرة:

- بتربط الـ Policy بالـ Pods باستخدام Labels + Selectors
- بتحدد نوع الترافيك المسموح:
  - Ingress : يسمح بدخول الترافيك.
  - Egress : يسمح بخروج الترافيك.
  - ممكن تستخدم الاتنين.

مثال عملي (منع الاتصال المباشر للـ DB إلا من خلال API)

❖ الهدف:

- السماح للـ DB Pod يستقبل ترافيك فقط من الـ API Pod على Port 3306
- منع أي Pod تاني (زي Web Pod) من الوصول للـ DB

❖ الخطوات:

1. إضافة Label للـ DB Pod : kubectl label pod db-pod app=db
2. إضافة Label للـ API Pod : kubectl label pod api-pod app=api
3. إنشاء NetworkPolicy :

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: db-policy
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          name: api-pod
    ports:
    - protocol: TCP
      port: 3306
```

4. تطبيق الـ Policy : kubectl apply -f db-policy.yaml

📌 النتيجة:

- الـ DB Pod هيسمح بترافيك جاي من الـ API Pod بس على 3306.
- أي Pod تاني (زي Web Pod) مش هيقدر يوصل للـ DB

ملاحظات مهمة

1. الـ Default Behavior مع NetworkPolicies :
   - بمجرد تطبيق Policy على Pod .. كل الترافيك يتمنع إلا اللي اتسمح له صراحة.
   - يعني : الـ Policy بتشتغل كـ deny-all by default بعد إنشائها.
2. المكان اللي بيتطبق فيه
   - الـ Policy بتتطبق على الـ Pod اللي متحدد في podSelector فقط.
3. الـ Network Plugin
   - مش كل الـ CNI Plugins بيدعموا Network Policies
   - أمثلة بتدعم : Calico, WeaveNet, Kube-Router, Romana
   - مثال لا يدعم : Flannel (ممكن تعمل Policy بس مش هتتطبق)
4. Egress Policies
   - زي Ingress بالظبط بس بتتحكم في الترافيك اللي خارج من الـ Pod
   - مثال: السماح لـ API Pod يتصل بالـ DB Pod فقط.

أوامر عملية مفيدة

- عرض الـ Policies : kubectl get networkpolicy
- تفاصيل Policy معينة: kubectl describe networkpolicy db-policy
- حذف Policy : kubectl delete networkpolicy db-policy

الخلاصة

- Network Policies = Firewall جوة Kubernetes
- بتستخدمها للتحكم في:
  - مين يقدر يوصل للـ Pods
  - أي Ports مسموحة.
  - تحدد Ingress أو Egress أو الاتنين.
- بتعتمد على الـ Labels
- لازم تتأكد إن الـ CNI Plugin بيدعم Network Policies

Intro. To Docker Storage

أهمية فهم التخزين مع الـ Containers

- قبل ما تفهم التخزين في Kubernetes ، لازم تكون فاهم الأول التخزين مع الـ containers (زي Docker)
- ليه؟ لأن Kubernetes مبني فوق Docker أو أي Container Runtime تاني وبالتالي نفس المفاهيم الأساسية بتنتقل، لكن بشكل أوسع وأكتر مرونة.
- مثال:
  - في Docker ، أي ملف بتكتبه جوه الـ container بيتمسح بمجرد ما الـ container يتمسح.
  - عشان كده لازم يكون عندنا آلية نخزن البيانات بشكل دائم (Persistent Storage)

مقارنة بسيطة

- في Docker :
  - عندنا Volumes : وسيلة نخزن بيها البيانات خارج دورة حياة الـ container
  - عندنا Bind Mounts : ربط مباشر بين فولدر في جهازك وفولدر جوه الـ container
- في Kubernetes :
  - بيتم التوسع في المفاهيم دي عن طريق Persistent Volumes (PV) و Persistent Volume Claims (PVC)عشان تديك إدارة مركزية ومرنة للتخزين.

مفاهيم مهمة

✅ Storage Drivers (درايفرات التخزين)

- دي المسؤولة عن إدارة التخزين الأساسي للـ container
- أمثلة في Docker :
  - overlay2 : الأكثر استخدامًا.
  - aufs, btrfs, zfs.
- هي اللي بتتحكم إزاي الملفات بتتخزن وتتشارك بين الـ containers

⚙️ أمر عملي تشوف بيه الدرايفر المستخدم في Docker : docker info | grep Storage

---

✅ Volume Drivers / Plugins :

- دي بتسمحلك تستخدم حلول تخزين خارجية (زي NFS, Ceph, AWS EBS, GCP Persistent Disk)
- يعني بدل ما التخزين يكون بس محلي على الجهاز، تقدر تخزن بياناتك على سيرفرات أو أنظمة تخزين خارجية.

⚙️ مثال أمر عملي لإنشاء volume مخصص بدرايفر معين : docker volume create --driver local myvolume

أو باستخدام Plugin : docker volume create --driver rexray/ebs myebsvolume

---

ربط الكلام بكوبيرنتس (تحضير للمحاضرات الجاية)

- في Kubernetes هنشوف:
  - PersistentVolume (PV) : هو المورد اللي بيمثل قطعة التخزين نفسها.
  - PersistentVolumeClaim (PVC) : ده زي الطلب اللي بيعمله البود عشان ياخد مساحة تخزين.
  - StorageClass : لتحديد نوع التخزين وطريقة تخصيصه (مثلاً SSD أو HDD أو AWS EBS)

📌 الخلاصة

- لازم الأول نفهم أساسيات التخزين في Docker (Drivers & Plugins)
- ده هيسهل علينا نفهم إزاي Kubernetes بيبني layer أعلي عشان يدير التخزين بطريقة أوتوماتيكية ومرنة.

Storage In Docker

الـ Docker بيخزن البيانات فين ؟

- لما تثبت Docker على أي نظام تشغيل Linux مثلاً فهو بيعمل فولدر أساسي غالبًا في:
- **/var/lib/docker**
- جواه هتلاقي فولدرات رئيسية:
  - /containers : فيه كل الملفات الخاصة بالـ containers اللي شغالة (زي logs, configs)
  - /image : فيه بيانات الـ images اللي سحبتها أو بنيتها.
  - /volumes : فيه الـ volumes اللي أنشأتها للتخزين الدائم.
  - /overlay2 أو أي driver تاني : ده بيحتوي على الطبقات (layers) الخاصة بالـ images

---

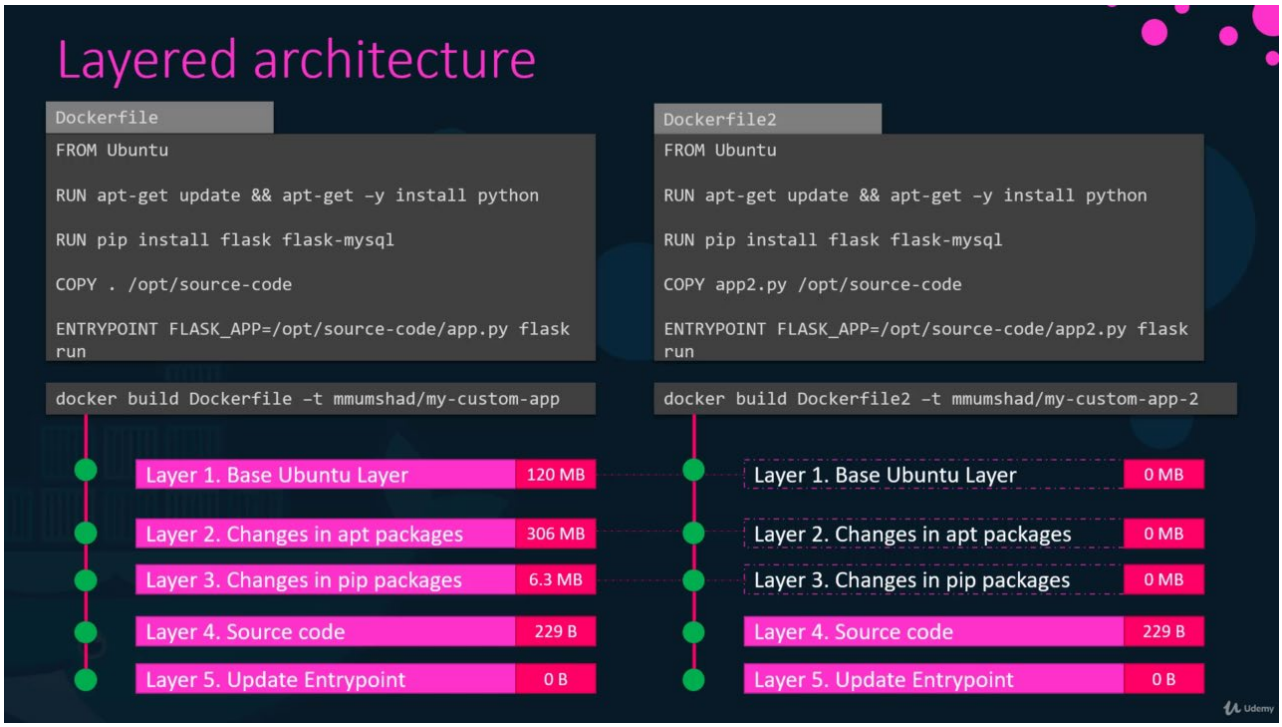
إزاي Docker بيخزن الـ Images ؟

- الـ images مبنية على Layered Architecture (معمارية الطبقات)
- يعني كل سطر في Dockerfile بيعمل طبقة جديدة فوق اللي قبلها:
  1. Base OS (مثلاً Ubuntu)
  2. Install packages (apt-get install)
  3. Install Python
  4. Copy source code.
  5. Define entrypoint
- كل طبقة Read-Only، والتغييرات بتتبني كطبقة جديدة فوقها.

⚙️ أمر تشوف بيه تفاصيل layers : docker history <image_name >

ميزة الطبقات (Layer Caching)

- لو عندك صورتين (images) ليهم نفس الـ base والـ dependencies ، Docker بيستخدم نفس الطبقات المشتركة من الكاش ومبيعملش إعادة بناء كامل.
- ده معناه:
  - سرعة أكبر في docker build
  - توفير مساحة تخزين.

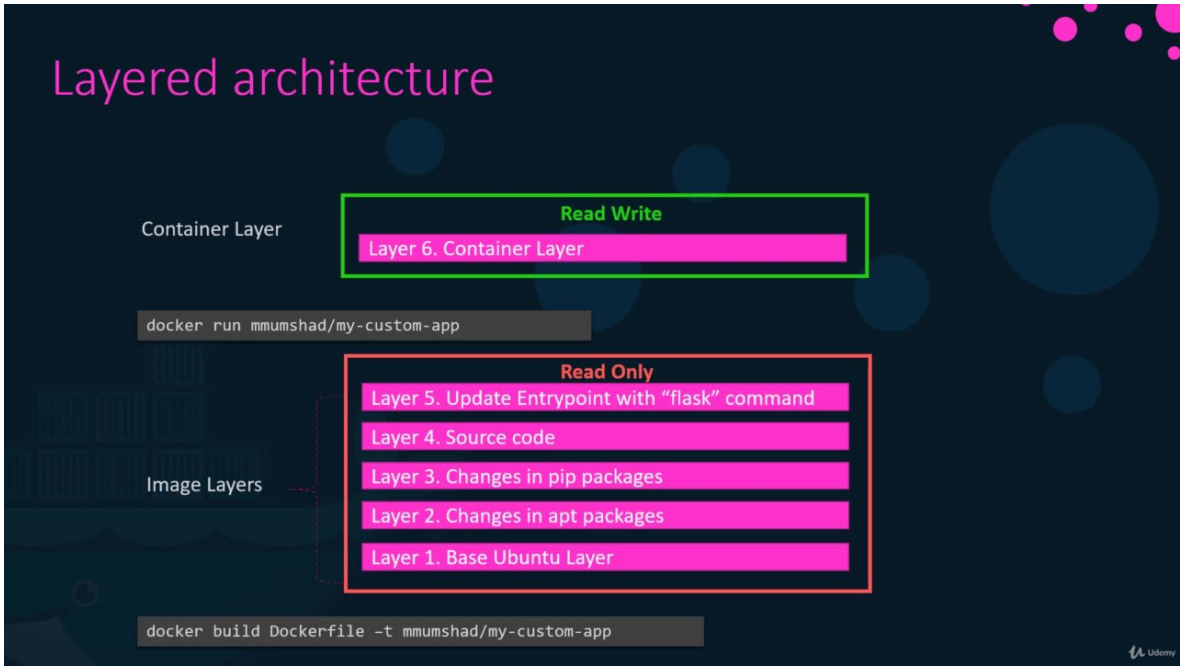


تشغيل Container من Image

- لما تعمل : docker run <image>
- فالـ Docker بيضيف طبقة جديدة قابلة للكتابة (Writable Layer) فوق الطبقات السابقة.
- دي اللي بتتخزن فيها:
  - Logs
  - Temporary files
  - أي تغييرات بتحصل داخل الـ container

⚠️ لكن:

- الطبقة دي بتتمسح أول ما الـ container يتمسح.
- عشان كده البيانات مش دائمة (Ephemeral)



Copy-On-Write Mechanism

- لو حاولت تعدل ملف من طبقة قديمة (Read-Only) :
  - Docker بيعمل نسخة منه في الـ Writable Layer
  - بعد كده التعديل بيحصل على النسخة الجديدة.
- ده بيضمن إن الطبقات القديمة مستقرة ومش بتتغير.

إزاي نخلي البيانات دايمة (Persistent Storage) ؟

عندنا ثلاث حلول :

✅ Volumes (الأكثر استخدامًا) :

1. إنشاء volume : docker volume create mydata

بيتخزن في: var/lib/docker/volumes/mydata/_data/

2. تشغيل container مع ربط الـ volume : docker run -d -v mydata:/var/lib/mysql mysql

✅ Bind Mounts (ربط فولدر موجود من الـ Host) (مش مستحبة)

- لو عندك بيانات في فولدر على الجهاز (data/ مثلاً) وعايز تستخدمه داخل الـ container :

```
docker run -d -v /data:/var/lib/mysql mysql
```

- بس خلي بالك ان في الحالة دي احنا هانوفر الـ Absolute Path بتاع الـ folder (في المثال دا data/) اللي عاوزين نعمله mount
- هنا بنسميها Bind Mount عكس الـ Volume اللي بيتخزن تحت Docker مباشرة

✅ طريقة جديدة (موصى بها):

- بدل v- القديم، نستخدم --mount لأنه أوضح .. ودا مثال بس بالـ --mount علي المثال السابق اللي بالـ Bind Mount :

```
docker run -d  --mount type=bind,source=/data,target=/var/lib/mysql  mysql
```

Storage Drivers

- هم المسؤولين عن:
  - إدارة الطبقات (Layers)
  - تنفيذ Copy-On-Write
  - تحكم في الأداء والثبات.

🛠️ أشهر الـ Storage Drivers :

- overlay2 : الافتراضي في Ubuntu وده الأكثر استقرارًا وأداء عالي.
- aufs : قديم جدًا ومش مدعوم في أنظمة حديثة.
- devicemapper : كان شائع في RHEL/CentOS
- btrfs و zfs : بتدي مميزات متقدمة لكن أقل شيوعًا.

⚙️ أمر تشوف الـ driver الحالي: `docker info | grep Storage`

الخلاصة

- Docker بيخزن بياناته في /var/lib/docker
- Images مبنية من طبقات Read-Only
- Containers بياخدوا طبقة Writable مؤقتة.
- البيانات بتضيع مع حذف الـ container إلا لو استخدمنا Volumes أو Bind Mounts
- Storage Drivers هم العقل اللي بينفذ كل ده والـ overlay2 هو الأشهر

Volume Driver Plugins in Docker

مراجعة سريعة

- قبل كده شرحنا إن Storage Drivers مسؤولين عن إدارة الـ Layers الخاصة بالـ Images والـ Containers (writable layer ، Copy-on-Write ... etc)
- لكن لما نيجي نتكلم عن Volumes(التخزين الدائم)، الوضع مختلف.

من المسؤول عن إدارة الـ Volumes ؟

- الـ Volumes مش بيتداروا عن طريق Storage Drivers
- اللي بيديرهم هو حاجة اسمها Volume Drivers أو Plugins
- فيه Plugin افتراضي اسمه local

◆ الـ local driver ببساطة بيخزن البيانات في: /var/lib/docker/volumes/<volume_name>/_data

---

أنواع Volume Drivers

- غير الـ local، فيه Plugins كتير من طرف ثالث (Third-Party) تسمحلك تخزن البيانات على حلول تخزين مختلفة زي :
  - Amazon EBS (Elastic Block Store)
  - Google Persistent Disk
  - Azure Disk/Files
  - DigitalOcean Block Storage
  - EMC, NetApp, Ceph ...
  - Plugin RexRay : مشهور بيدعم أكتر من backend زي AWS EBS, S3, , OpenStack Cinder.

إزاي أستخدم Plugin معين بدل الافتراضي؟

- ممكن تحدد الـ Driver وأنت بتنشئ Volume :

```
docker volume create  --driver rexray/ebs  --name myebsvolume
```

- كده بيتم عمل Volume مربوط مباشرة بخدمة AWS EBS

ربط Volume بالـ Container

- بعد إنشاء الـ volume ، تربطه عادي بالـ container

```
docker run -d  --mount source=myebsvolume,target=/var/lib/mysql   mysql
```

- النتيجة: البيانات بتاعتك محفوظة في الـ Cloud (EBS) حتى لو مسحت الـ container

مقارنة سريعة بين Local Volume & External Volume

- **Local Volume (Default):**
  - سهل وسريع.
  - بيتخزن محليًا على نفس الـ Host
  - البيانات بتضيع لو الـ Host ضاع.
- **External Volume (Third-Party Plugin) :**
  - بيديك تخزين دائم ومرن.
  - ممكن توصّل نفس الـ volume لأكتر من Host
  - أفضل للـ Production خصوصًا في بيئة Cluster زي Kubernetes أو Swarm

أوامر عملية مهمة

- إنشاء volume محلي: **docker volume create mylocaldata**
- عرض الـ volumes المتاحة: **docker volume ls**
- معرفة تفاصيل Volume معين: **docker volume inspect mylocaldata**
- حذف Volume : **docker volume rm mylocaldata**

الخلاصة

- Storage Drivers مسؤولين عن إدارة ملفات الـ Images والـ Containers
- Volume Drivers أو Plugins مسؤولين عن إدارة Volumes
- الافتراضي local: بيخزن البيانات على الـ Host نفسه
- ممكن تستخدم Plugins من طرف ثالث للتخزين السحابي أو الخارجي زي AWS EBS, Google PD, RexRay
- باستخدام الـ Plugins ، تقدر تضمن إن البيانات بتفضل موجودة حتى بعد ما الـ container يتمسح أو يتنقل.

Container Storage Interface (CSI)

مقدمة

- قبل كده اتكلمنا عن Storage Drivers و Volume Drivers داخل Docker
- لكن ده كله كان مرتبط بالـ Docker Runtime بس.
- مع الوقت ظهرت Runtimes تانية غير Docker زي:
  - rkt (Rocket)
  - CRI-O
- عشان كده كان لازم يكون فيه واجهة قياسية موحدة تسمح لأي Orchestrator زي Kubernetes إنه يتكلم مع أي Runtimeأو Storage Backend

---

Container Runtime Interface (CRI)

- زي ما فيه Container Runtime Interface (CRI) اللي بيحدد إزاي Kubernetes يتعامل مع أي Container Runtime
- بقى فيه برضه Container Network Interface (CNI) عشان الشبكات.
- نفس الفكرة اتطبقت على التخزين Container Storage Interface (CSI)

ما هوCSI ؟

- CSI = Container Storage Interface
- هو Standard/Specification (يعني مش منتج بحد ذاته)
- الهدف منه:
  - يخلي أي Orchestrator : (Kubernetes, Mesos, Cloud Foundry ... etc) يقدر يشتغل مع أي Storage Provider
  - من غير ما كل Orchestrator يكتب Integration خاص بيه لكل Vendor

---

أمثلة لمزودين عندهم CSI Drivers

- AWS EBS (Elastic Block Store)
- Google Persistent Disk
- Azure Disk/Files
- Dell EMC
- NetApp
- Ceph, OpenStack Cinder ...

بالتالي: أي Vendor بيقدر يكتب CSI Driver هايقدر يشتغل مباشرة مع Kubernetes أو أي Orchestrator بيدعم CSI

إزاي CSI بيشتغل؟

الـ CSI بيعتمد على gRPC Calls (Remote Procedure Calls) ما بين الـ Orchestrator زي Kubernetes والـ Storage Driver

أمثلة على الـ Calls

- CreateVolume : لإنشاء Volume جديد.
- DeleteVolume : لحذف Volume
- ControllerPublishVolume : لربط Volume بـ Node معينة.
- NodeStageVolume / NodePublishVolume : لعمل Mount للـ Volume جوه Pod

مثال عملي (Kubernetes + CSI)

لو عندي Kubernetes Cluster وعاوز أستخدم AWS EBS ، بخطوات بسيطة:

1. أعمل PersistentVolume (PV) :

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: my-ebs-pv
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  csi:
    driver: ebs.csi.aws.com
    volumeHandle: vol-0abcd1234efgh5678
```

2. أعمل PersistentVolumeClaim (PVC) عشان الـ Pod يستخدم الـ PV :

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-ebs-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
```

3. أربط الـ PVC بالـ Pod :

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: app-pod
spec:
  containers:
  - name: app
    image: nginx
    volumeMounts:
    - mountPath: "/usr/share/nginx/html"
      name: my-storage
  volumes:
  - name: my-storage
    persistentVolumeClaim:
      claimName: my-ebs-pvc
```

النتيجة: الـ Pod بيستخدم Volume من AWS EBS من خلال الـ CSI Driver

---

مميزات CSI

- Standard مفتوح : يعني نفس الـ Plugin يشتغل مع أكتر من Orchestrator
- مرونة : أي Vendor Storage يقدر يعمل Driver خاص بيه.
- قابلية التوسع : تدعم أي نوع Storage (Block, File, Object)
- فصل المسؤوليات Orchestrator : مش محتاج يعرف تفاصيل عن Storage Backend

---

الخلاصة

- Docker كان محدود بالتخزين المحلي والـ Volume Plugins
- مع تطور الـ Orchestration ظهر CSI كمعيار عالمي.
- CSI = بيسمح لأي Orchestrator إنه يتكامل مع أي Storage Vendor بسهولة.
- بيشتغل عن طريق gRPC API Calls زي Create/Delete/Attach/Mount
- النهاردة معظم الـ Vendors الكبار عندهم CSI Drivers رسميين.

Volumes In Kubernetes

فكرة عامة

- الـ Containers في Docker أو Kubernetes طبيعتهم مؤقتة (Transient)
  يعني بتتعمل وقت الحاجة ولما يخلص دورها بتتمسح.
- نفس الفكرة مع البيانات جوه الـ Container بتتمسح بمجرد مسح الـ Container

---

الحل Volumes :

- عشان نحتفظ بالبيانات حتى بعد مسح الـ Pod/Container بنستخدم Volumes
- الـ Volume بيبقى عبارة عن مساحة تخزين منفصلة متوصلة بالـ Pod
- البيانات اللي بيتولدها الـ Pod تتخزن جواه، وحتى لو الـ Pod اتمسح فالبيانات بتفضل موجودة.

مثال عملي

عندنا Pod بيولد رقم عشوائي ويخزنه في ملف /opt/number.out

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: random-number-generator
spec:
  containers:
  - image: alpine
    name: alpine
    command: ["/bin/sh","-c"]
    args: ["shuf -i 0-100 -n 1 >> /opt/number.out;"]
    volumeMounts:
    - mountPath: /opt
      name: data-volume

  volumes:
  - name: data-volume
    hostPath:
      path: /data
      type: Directory
```

الحل الأفضل External / Distributed Storage :

Kubernetes بيدعم أنواع تخزين كتيرة زي:

- NFS (شبكة ملفات)
- GlusterFS .. CephFS .. FibreChannel .. ScaleIO .. AWS EBS .. Azure Disk / Azure File.

مثال AWS EBS Volume

لو عايز أخزن البيانات على AWS بدل ما أخزن على الـ hostPath :

```
volumes:
- name: data-volume
  awsElasticBlockStore:
    volumeID: <volume-id>
    fsType: ext4
```

- VolumeID = ID بتاع الـ EBS Volume في AWS
- fsType = نوع نظام الملفات زي ext4

الخلاصة

- الـ Volumes = حل لتخزين البيانات بشكل دائم حتى بعد مسح الـ Pod
- hostPath ينفع في Node واحدة لكن مش مناسب لـ Production Multi-Node Cluster
- في الـ Production نستخدم حلول خارجية أو Cloud Storage
- ده كان المدخل لمفهوم أكبر وأهم في Kubernetes وهو Persistent Volumes (PV) و Persistent Volume Claims (PVC) اللي هنتكلم عنها بعد كده.

Persistent Volumes

مراجعة سريعة

- قبل كده اتكلمنا عن Volumes العادية.
- كنا بنكتب إعدادات الـ Volume جوه ملف تعريف الـ Pod نفسه (pod.yaml)
- ده معناه إن كل مرة أي مستخدم يعمل Pod جديد ← لازم يكتب إعدادات التخزين بنفسه في الملف.

مشكلة Volumes العادية

- تخيل عندك Cluster كبير فيه عشرات أو مئات المستخدمين، وكل مستخدم بيعمل Pods كتير.
- كل مستخدم هيضطر يكتب إعدادات التخزين في كل Pod !!!!
- ولو غيرت إعداد التخزين زي نوع الـ Storage أو مكانه فلازم تعدل كل ملفات الـ Pods يدويًا.
- ده غير عملي ومجهد جدًا.

الحل Persistent Volumes (PV) :

- هنا بييجي دور الـ Persistent Volumes (PV)
- الـ PV عبارة عن Pool مركزي للتخزين بيتم إنشاؤه من قبل الـ Administrator
- بعد كده أي مستخدم يقدر "يحجز" جزء من التخزين ده من خلال Persistent Volume Claims (PVC)
- كده يبقى عندنا فصل كامل بين إدارة التخزين (Admins) واستخدامه (Users)

تعريف Persistent Volume

- الـ PV هو Resource مستقل في الـ Cluster ، مش مربوط بـ Pod واحد.
- بيتم إنشاؤه مرّة واحدة، وبعدين أي Pod يقدر يستخدمه عن طريق الـ PVC
- الـ Admin ممكن يجهّز أكتر من PV بأحجام وأنواع مختلفة.

خطوات إنشاء Persistent Volume

مثال عملي:(YAML)

pv-definition.yaml

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-vol1
spec:
  accessModes:
      - ReadWriteOnce
  capacity:
      storage: 1Gi
  hostPath:
     path: /tmp/data
```

❖ شرح:

- API : apiVersion: v1 المستخدم.
- metadata.name اسم الـ PV
- capacity.storage : ده حجم التخزين (هنا 1 Gi)
- accessModes : (Define how a volume should be mounted on the host)
  - ReadOnlyMany : قراءة فقط من أكثر من Pod.
  - ReadWriteOnce : قراءة/كتابة لكن Pod واحد بس.
  - ReadWriteMany : قراءة/كتابة من أكتر من Pod
- hostPath : (Node Local dir)
  - المكان اللي هيستخدمه على الـ Node وده مجرد مثال للتجربة مش Production (زي ماشرحنا قبل كدا)

ملاحظات مهمة

- hostPath ينفع للتجارب أو بيئة صغيرة، لكن غير مناسب لـ Production
- في Production نستخدم حلول Storage معتمدة زي:
  - AWS EBS
  - Azure Disk / Azure File
  - Google Persistent Disk
  - NFS, CephFS, GlusterFS ...

مثال : AWS EBS PV

pv-definition.yaml

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
   name: pv-vol1
spec:
  accessModes:
      - ReadWriteOnce
  capacity:
      storage: 1Gi

  awsElasticBlockStore:
    volumeID: <volume-id>
    fsType: ext4
```

أوامر عملية

- إنشاء PV : kubectl create -f pv.yaml
- عرض كل الـ PVs : kubectl get pv
- تفاصيل PV معيّن: kubectl describe pv pv-vol1

الخلاصة

- Volumes العادية : اللي بتتكتب جوه كل Pod مرهقة للمستخدمين.
- Persistent Volumes (PV) : بتتجهز مرة واحدة بشكل مركزي من الـ Admin
- الـ Users بياخدوا منها اللي محتاجينه عن طريق Persistent Volume Claims (PVC)

Persistent Volume Claims (PVS)

مراجعة سريعة

- في المحاضرة اللي فاتت عملنا Persistent Volume (PV)
- الـ PV عبارة عن مخزن مركزي جهّزه الـ Admin
- دلوقتي عايزين نعرف إزاي الـ Users يستعملوا التخزين ده جوه Pods بتاعتهم ؟

➡️ الحل Persistent Volume Claim (PVC)

العلاقة بين PV و PVC

- PV و PVC هما Objects منفصلين في الـ Kubernetes
- الـ Admin بيجهّز مجموعة Persistent Volumes (PVs)
- المستخدم (User) بيعمل Persistent Volume Claim (PVC) علشان يطلب مساحة تخزين معينة.
- الـ Kubernetes بعد كده بيعمل Binding ما بين الـ Claim والـ Volume المناسب.

عملية الـ Binding

- لما يتعمل PVC :
  - الـ Kubernetes بيدوّر على PV متوافق معاه.
  - بيطابق السعة (capacity) المطلوبة.
  - بيطابق خصائص إضافية زي:
    - AccessModes (طريقة الوصول)
    - VolumeMode
    - StorageClass
- لو فيه أكتر من PV ينفع يتربط فممكن نستخدم Labels + Selectors علشان نوجّه الـ Claim لواحد محدد.

⚠️ ملاحظة:

- ممكن PVC صغير (500MB) يتربط مع PV أكبر(1GiB)
- لكن ماينفعش Claim تاني يستخدم المساحة الفاضية اللي باقية.
- يعني العلاقة One-to-One

حالة Pending

- لو مفيش PV متاح يناسب الـ PVC : الـ Claim يفضل Pending
- أول ما يضاف PV مناسب : يتعمل الـ Binding تلقائيًا.

إنشاء PVC

pvc-definition.yaml

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: myclaim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 500Mi
```

pv-definition.yaml

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-vol1
spec:
  accessModes:
    - ReadWriteOnce
  capacity:
    storage: 1Gi
  awsElasticBlockStore:
    volumeID: <volume-id>
    fsType: ext4
```

```
kubectl create -f pvc-definition.yaml
```

✅شرح:

- kind: PersistentVolumeClaim :: نوع الـ Resource
- metadata.name : اسم الـ PVC
- accessModes: ReadWriteOnce : Podواحد يقدر يقرأ ويكتب.
- resources.requests.storage : الحجم المطلوب (500Mi)

أوامر عملية

- إنشاء PVC : kubectl create -f pvc.yaml
- عرض كل الـ PVCs : kubectl get pvc
- متابعة حالة الـ Binding : kubectl describe pvc myclaim
- عرض الـ PVs المرتبطة: kubectl get pv

مصير الـ PV عند حذف الـ PVC

- لما تمسح PVC : في خيارات مختلفة للـ PV المرتبط:

1. Retain (By Default) :
   - الـ PV يفضل موجود.
   - مينفعش يتربط مع Claim جديد إلا لو الـ Admin عمله Delete يدوي.
2. Delete :
   - الـ PV بيتحذف تلقائيًا.
   - التخزين الأساسي بيتحرر زي AWS EBS أو Azure Disk
3. Recycle (قديم ومش مدعوم في الإصدارات الجديدة) :
   - البيانات القديمة بتتـ"مسح" وتتجهز للمستخدم الجديد.

الخلاصة

- PVC هو الطريقة اللي المستخدم بيطلب بيها Storage من الـ PV pool
- Kubernetes يربط الـ PVC مع PV واحد فقط.
- العلاقة واحد لواحد.
- لازم تكون إعدادات الـ PVC متوافقة مع الـ PV (Storage size + Access mode + Storage class)
- حذف الـ PVC بيأثر على الـ PV حسب سياسة الـ Reclaim

Networking – Intro

المتطلبات الأساسية (Prerequisites)

قبل ما نخش في تفاصيل Networking في Kubernetes ، لازم يكون عندك خلفية عن شوية أساسيات:

- إزاي تشوف وتعدل الـ IP Address على أي جهاز
  - مثال عملي على Linux :

```
ip addr show
ip addr add 192.168.1.100/24 dev eth0
```

- مفهوم الـ Gateway والـ Routes (إزاي البيانات بتخرج من جهازك وتروح لشبكات تانية)
  - مثال عملي:

```
ip route show
ip route add default via 192.168.1.1
```

- الـ DNS (Domain Name System) : إزاي أسماء الـ Domains زي (google.com تتحول لـ IP Address)
  - على Linux : إعداد DNS في /etc/resolv.conf
- CoreDNS : هو DNS Solution مدمج في Kubernetes علشان يحل أسماء الخدمات (Services) جوه الكلاستر.

Networking Namespaces

- Linux بيوفر ميزة اسمها Network Namespace : كل Namespace بيبقى عنده شبكة مستقلة (IP، Routing، Interfaces)
- Docker وكمان Kubernetes بيستغلوا ده علشان يعزلوا الشبكات بتاعة كل Container/POD
  - جرب تشوف الـ network namespaces : ip netns list

ترتيب الدروس في Networking

هنمشي بالترتيب التالي علشان يبقى الفهم أسهل:

1. Cluster Networking Needs : متطلبات الشبكة في الكلاستر كله.
2. Pod Networking Concepts : إزاي الـ PODs بتكلم بعض.
3. CNI (Container Network Interface) : الـ Plugins اللي بتحل مشكلة توصيل الـ PODs ببعض.
   - مثال Plugins: Flannel, Calico, Weave
4. Cluster DNS : إزاي الخدمات بتتسمى وبيحصلها resolve جوه Kubernetes
5. Ingress Networking : التحكم في دخول الترافيك من برة الكلاستر للـ Services جوة.

ملاحظات هامة

- الجزء دا (Networking in Kubernetes) يعتبر من أهم الأجزاء، ولازم تستوعبه كويس علشان الامتحان + علشان الشغل العملي.
- من أكتر الحاجات اللي تتسئل عمليًا في الـ CKA Exam :
  - التعامل مع الـ DNS.
  - فهم الـ Pod-to-Pod Networking
  - إزاي الـ Services تشتغل وتوجه الترافيك.

Prerequisite Switching Network

ما هو الـ Network ؟

- شبكة = مجموعة أجهزة (PCs, VMs, Servers) متصلة مع بعض.
- الاتصال بيتم عن طريق Interface كارت شبكة سواء حقيقي أو Virtual
- للتأكد من وجود الـ Interfaces : ip link
  - مثلًا eth0, wlan0

ربط الأجهزة عبر Switch

- لو عندك جهازين A و B وعايزهم يتواصلوا فبتوصلهم على Switch
- كل جهاز بياخد IP Address في نفس الشبكة:
  - الشبكة 192.168.1.0/24 :
  - A = 192.168.1.10
  - B = 192.168.1.11

📌 أمر تعيين IP : ip addr add 192.168.1.10/24 dev eth0

بعد كده يقدرو يعملوا ping لبعض : ping 192.168.1.11

مشكلة الاتصال بين شبكتين مختلفتين

- شبكة الأولي 192.168.1.0/24 ودي فيها جهازين (A & B)
- شبكة الثانية 192.168.2.0/24 : ودي فيها جهازين (C & D)
  - ❖ فلو (الجهاز A) عايز يكلم (الجهاز C) فمش هيعرف يوصل لأن الـ Switch بيشتغل جوا شبكة واحدة بس

الحل Router

- Router بيعمل ربط بين شبكتين مختلفتين.
- بياخد IP من كل شبكة:
  - الشبكة الأولي هياخد 192.168.1.1
  - الشبكة الثانية هيأخذ 192.168.2.1
- أمر إضافة Route : ip route add 192.168.2.0/24 via 192.168.1.1
  - معناها: أي ترافيك متوجه للشبكة 192.168.2.0/24يعدي عن طريق 192.168.1.1 (الراوتر)
- أمر عرض الـ Routing Table : ip route

لاحظ اني بعمل Routing + Default GW (مع بعض) للشبكة اللي عاوزها تتواصل مع شبكة اخري داخليا من غير مايطلعوا عالنت ودا الـ Routing وفي نفس الوقت شبكات تانية عالنت وطبعا مش هاتطلع الا بالـ Default GW زي ماوضحنا

Default Gateway

- علشان الأجهزة تطلع لأي شبكة خارجية (مثال الإنترنت) فبتضيف Default Gateway
- صيغة : ip route add default via 192.168.1.1 or ip route add 0.0.0.0/0 via 192.168.1.1
  - 📌 معنى = 0.0.0.0/0 أي عنوان IP مش معروف مساره.

Linux as a Router

- لو عندك جهاز B متوصل بشبكتين (يعني هواللي في النص) ..
  - 192.168.1.6على eth0
  - 192.168.2.6على eth1
  - وكل شبكة من الشبكتين متوصلة بجهاز A & C .. جهاز A 192.168.1.5 .. وكمان جهاز C 192.168.2.5

- فعلشان A يقدر يكلم C فهايبقي من خلال B ... فلازم عند الجهاز A اعمل :

ip route add 192.168.2.0/24 via 192.168.1.6

- ولو عوزت ان C يكلم A .. فلازم عند الجهاز C اعمل :

Ip route add 192.168.1.0/24 via 192.168.2.6

- B يقدر يشتغل كـ Router لكن لازم نسمح له يعمل Packet Forwarding يعني عند B هانعمل :
  - ❖ أمر التفعيل المؤقت : echo 1 > /proc/sys/net/ipv4/ip_forward
  - ❖ لجعله دائم : هاتفتح /etc/sysctl.conf ثم تفعل net.ipv4.ip_forward=1
    - ○ طبق التغيير: sysctl -p

أوامر أساسية تلخص المحاضرة

- عرض الـ Interfaces ip link
- عرض/إضافة IP : ip addr show or ip addr add 192.168.1.10/24 dev eth0
- عرض/إضافة Routes : ip route or ip route add 192.168.2.0/24 via 192.168.1.1
- تفعيل Forwarding : cat /proc/sys/net/ipv4/ip_forward or echo 1 > /proc/sys/net/ipv4/ip_forward

---

🔗 الربط مع Kubernetes

- كل Node في الكلاستر بيبقى ليه Interfaces و Routes
- الـ Pods بتحتاج + Network Isolation إمكانية تواصل بين بعضها.
- هنا بييجي دور (CNI Plugins) زي Calico, Flannel علشان يعملوا نفس اللي شرحناه لكن بشكل أوتوماتيكي.

Prerequisite DNS

السيناريو الأول : استخدام ملف /etc/hosts

- عندنا جهازين A و B على نفس الشبكة :
  - A → 192.168.1.10
  - B → 192.168.1.11
- تقدر تعمل ping من A على B باستخدام الـ IP
- لكن لو عاوز تفتكر اسم زي db بدلIP ، بتضيف سطر في /etc/hosts على جهاز A : db 192.168.1.11
- من دلوقتي أي أمر زي : ping db هيوصل لـ B

❖ المهم : اللي في /etc/hosts هو الحقيقة المطلقة بالنسبة للجهاز حتى لو الاسم الحقيقي بتاع B مختلف

- ممكن حتى تخلي A يفتكر إن B = google.com لو كتبت كده في الملف. 😅

📝 عيب الطريقة دي: مع زيادة عدد الأجهزة، الملف بيكبر وصعب تديره، ولو IP جهاز اتغير لازم تعدل في كل الأجهزة.

السيناريو التاني : استخدام DNS Server

- بدل ما كل جهاز يحتفظ بملف كبير، بنخلي فيه سيرفر DNS مركزي يحتفظ بالـ records
- كل جهاز بيتوجه للـ DNS server من خلال إعداد في /etc/resolv.conf : nameserver 192.168.1.100
- كده أي جهاز يحتاج يحل اسم هايروح يسأل السيرفر المركزي.
- لو IP اتغير فهاتعدل في DNS server مرة واحدة فقط.
- ممكن كمان تضيف record محلي في /etc/hosts لو حاجة خاصة بيك (زي سيرفر تجارب)، بس مش هيبان لباقي الناس.

أولوية البحث

- الجهاز بيشوف الأول /etc/hosts بعدين DNS server
- ترتيب الأولوية بيتحدد في /etc/nsswitch.conf : hosts: files dns

  يعني يدور في الملفات (hosts) الأول، بعدين يسأل DNS

ربط بالـ Domains

- أسماء زي www.google.com بتتكون من:
  - com → Top Level Domain (TLD)
  - google → Domain Name
  - www → Subdomain
- نفس الفكرة تقدر تعملها في شركتك:
  - web.mycompany.com
  - db.mycompany.com
- ولتسهيل الاستخدام داخليًا،

  ممكن تضيف في /resolv.conf : search mycompany.com

فلو كتبت بس ping web هيترجم تلقائيًا لـ web.mycompany.com

```
cat >> /etc/resolv.conf
nameserver      192.168.1.100
search          mycompany.com  prod.mycompany.com
```

```
ping web
PING web.mycompany.com (192.168.1.10) 56(84) bytes of d
64 bytes from web.mycompany.com (192.168.1.10): … time=
64 bytes from web.mycompany.com (192.168.1.10): … time=
```

```
ping web.mycompany.com
PING web.mycompany.com (192.168.1.10) 56(84) bytes of d
64 bytes from web.mycompany.com (192.168.1.10): ttl=64
```

📌 أنواع الـ DNS Records وأشهرها

- **A Record** : يربط اسم مع IPv4
- **AAAA Record** : يربط اسم مع IPv6
- **CNAME Record** : يعمل Alias مثلاً food.com = hungry.com

🛠️ أوامر اختبار DNS

- **ping** ينفع، لكن مش دايمًا الأفضل.
- أدوات أهم:
  - **nslookup** : يسأل DNS server
  - **dig** : زي nslookup لكن يدي تفاصيل أكتر.

⚠️ ملحوظة nslookup و dig مش هيبصوا على **/etc/hosts**، بيكلموا الـ DNS server فقط.

✅ الخلاصة

- /etc/hosts : حل يدوي بسيط مناسب للشبكات الصغيرة.
- DNS Server : حل مركزي ديناميكي ← للشبكات الكبيرة والمؤسسات.
- الجهاز دايمًا يبدأ بالـ **/etc/hosts** و لو مش لاقي يروح للـ DNS
- نقدر نزود search domains عشان نكتب أسماء قصيرة.
- فيه أوامر مخصصة لاختبار DNS زي nslookup و dig

Prerequisite Network NameSpaces

🎯 الفكرة العامة

الفيديو بيتكلم عن Network Namespaces في لينكس.
الـ Namespace يعني بيئة معزولة جوه نفس الـ Host ، بحيث كل Container أو Service يبان إنه عنده شبكته الخاصة، Routing خاص، و ARP Table خاصة، من غير ما يشوف اللي على الـ Host أو باقي الـ Containers

❖ طيب هانمشي ازاي علشان متتوهش مني ؟!

1. إنشاء Network Namespace
2. إضافة Interfaces جواه.
3. توصيل Namespaces ببعض بكابل افتراضي (veth pair)
4. توصيل أكتر من Namespace بشبكة داخلية باستخدام Bridge
5. عمل Routing بين Namespaces والـ Host
6. عمل NAT عشان يخرجوا للإنترنت.
7. عمل Port Forwarding عشان الإنترنت يوصل للـ Namespaces

🏠 التشبيه الأول

- الـ Host زي بيت كبير.
- كل Namespace زي أوضة مخصوصة لواحد من الأولاد.
- كل واحد يشوف بس الحاجة اللي جوه أوضته وبالتالي بيوفر Privacy كاملة.
- الأب (هانعتبره الـ Host في الحالة دي) يقدر يشوف الكل، ويربط بين أوضتين لو حب.
- نفس الكلام للـ Containers : بيتحطوا في Network Namespace عشان ما يشوفوش بعض.

🛠️ أوامر البداية

1. عرض الشبكات والـ Namespaces : ip netns list ولو لسه ماعملتش أي Namespace هتلاقي القائمة فاضية
2. إنشاء Namespace : ip netns add red or ip netns add blue
   - دلوقتي عندنا اتنين red و blue
3. استعراض Interfaces في الـ Host : ip link
   - هتلاقي موجود :
     - lo (loopback)
     - eth0 أو أي Interface فعلي.
4. استعراض Interfaces داخل Namespace : ip netns exec red ip link
   - هتلاقي بس الـ lo، مفيش أي حاجة من اللي في الـ Host

🌐 العزل (Isolation)

- الـ Host عنده Routing Table و ARP Table
- Namespace جديد (يعني معزول) : يعني مفيهوش غير loopback ، ومفيش Routing ولا ARP
- كده عملنا Isolation كامل.

🔌 التوصيل بين Namespaces (veth pair)

- محتاجين نعمل زي كابل افتراضي.
- الكابل ده اسمه veth pair : طرفين متوصلين ببعض.

1. إنشاء الكابل: ip link add veth-red type veth peer name veth-blue
2. ربط كل طرف بـ Namespace :

- ip link set veth-red netns red
- ip link set veth-blue netns blue

3. تفعيل الطرفين + إعطاء IP :

- ip netns exec red ip addr add 192.168.15.1/24 dev veth-red
- ip netns exec blue ip addr add 192.168.15.2/24 dev veth-blue
- ip netns exec red ip link set veth-red up
- ip netns exec blue ip link set veth-blue up
- ip netns exec red ip link set lo up
- ip netns exec blue ip link set lo up

4. اختبار الاتصال: ip netns exec red ping 192.168.15.2

✅ هيشتغل → لأن فيه كابل افتراضي.

🔀 التوصيل المتعدد (Bridge)

لو عندك أكتر من Namespace مش بس (red وblue) ، مش منطقي تعمل كابلات كتير بينهم .. طاب ايه الحل ؟!؟!

الحل : Bridge = Switch افتراضي.

1. إنشاء Bridge :

- ip link add v-net-0 type bridge
- ip link set v-net-0 up

2. عمل veth pair لكل Namespace :

- ip link add veth-red type veth peer name veth-red-br
- ip link set veth-red netns red
- ip link set veth-red-br master v-net-0
- ip link set veth-red-br up
- ip link add veth-blue type veth peer name veth-blue-br
- ip link set veth-blue netns blue
- ip link set veth-blue-br master v-net-0
- ip link set veth-blue-br up

3. إعطاء IP لكل Namespace :

- ip netns exec red ip addr add 192.168.15.1/24 dev veth-red
- ip netns exec blue ip addr add 192.168.15.2/24 dev veth-blue
- ip netns exec red ip link set veth-red up
- ip netns exec blue ip link set veth-blue up
- ip netns exec red ip link set lo up
- ip netns exec blue ip link set lo up

4. كده كل Namespaces متوصلين مع بعض عبر الـ Bridge

🌍 ربط الـ Host مع الـ Namespaces

- ندي الـ Host IP على الـ Bridge : ip addr add 192.168.15.5/24 dev v-net-0
- دلوقتي الـ Host يقدر يعمل Ping على أي Namespace

🚪 التوصيل بالإنترنت

الـ Namespaces مش شايفين الإنترنت .. وعشان يطلعوا:

1. لازم نعمل Route + Default Gateway عندهم:

```
ip netns exec blue ip route add default via 192.168.15.5
```

   - لاااحظ ان الـ Host هنا هو الـ Gateway
2. لازم الـ Host يعمل NAT عشان يترجم IPs الداخلية لـ IP الخارجي بتاعه:

```
iptables -t nat -A POSTROUTING -s 192.168.15.0/24 -o eth0 -j MASQUERADE
```

✅ كده أي Namespace هيقدر يخرج للإنترنت.

🔁 التوصيل من الإنترنت للـ Namespace (Port Forwarding)

لو في Service جوا Namespace مثلاً Web Server على Port 80 وعاوز توصله من بره:

- نستخدم Port Forwarding :

```
iptables -t nat -A PREROUTING -p tcp --dport 8080 -j DNAT --to-destination 192.168.15.2:80
```

- دلوقتي أي حد يفتح http://<host-ip>:8080 : هيوصل للـ Web Server في الـ Namespace

---

✅ الخلاصة

- Network Namespace : عزل شبكي كامل.
- veth pair : كابل افتراضي يوصل 2 Namespaces
- Bridge = Switch افتراضي يوصل عدد لا نهائي من Namespaces + Host
- Routing + NAT : يخلوا الـ Namespaces يطلعوا بره.
- Port Forwarding : يخلي الإنترنت يدخل عليهم.

Prerequisite Docker Networking

Networking in Docker

لما تثبت Docker على أي سيرفر، بيكون عندك كذا Networking Option عشان تحدد إزاي الكونتينر يتواصل مع باقي العالم.

---

1 None Network

- الكونتينر مش بيتوصل بأي شبكة.
- مفيش إنترنت، مفيش تواصل مع الكونتينرات التانية.
- بتستخدمها لو عاوز تعمل عزل كامل.

◇ مثال : docker run --network none -it alpine sh

- ولو جربت جوا الكونتينر: ping 8.8.8.8
  هيفشل لأن مفيش شبكة

---

2 Host Network

- الكونتينر بيستخدم نفس شبكة الـ Host
- يعني لو الـ Host عنده eth0 بـ IP 192.168.1.10 يبقي الكونتينر كمان هيبان بنفس الشبكة.
- أي خدمة في الكونتينر على بورت 80، هتكون مباشرة على بورت 80 بتاع الـ Host

◇ مثال : docker run --network host -p 80:80 nginx

  - مش محتاج تعمل Port Mapping

- العيب: مش هتقدر تشغل 2 كونتينر على نفس البورت.

---

3 Bridge Network (الافتراضية)

دي اللي تهمنا أكتر:

- أول ما تثبت Docker ف بيتعمل Network افتراضية اسمها bridge
- على مستوى Linux هتلاقيها اسمها docker0 (عالـ Host يعني)
- الـ Subnet الافتراضي بتاعها 172.17.0.0/16

◇ تحقق:

- docker network ls
- ip link show docker0
- ip addr show docker0

- هتلاقيها واخدة IP زي 172.17.0.1 :
- أي كونتينر جديد بيتخلق → بياخد IP من نفس الشبكة زي DHCP صغير

◇ مثال تشغيل Nginx :

- docker run -d --name web nginx
- docker exec -it web ip addr show eth0

هتلاقي IP زي 172.17.0.2

⚡ إزاي Docker بيوصل الكونتينرات بالـ Bridge ؟

- بيعمل لكل كونتينر Network Namespace
- وبيربطها بالـ Host عن طريق veth pair (كابل افتراضي طرفين) (لو مش فاهم .. راجع المحاضرة السابقة)
- طرف بيركب في الكونتينر، والتاني في الـ Host اللي هو docker0
- زي ما شرحنا قبل كده مع Linux namespaces

◇ تحقق : ip link show or ip -n "namespaces(u can know it from ip netns)" link

هتلاقي pairs زي: veth1234@if5 or veth5678@if6

- جوة الكونتينر: ip addr

  هتلاقي IP مخصص له (مثلا 172.17.0.3)

⚡ Port Mapping

- الكونتينر جوة شبكة خاصة (172.17.x.x) ، يعني مش ظاهر للعالم الخارجي.
- عشان توصله من برة، محتاج Port Mapping

◇ مثال : docker run -d -p 8080:80 nginx

- أي حد يدخل على http://HOST_IP:8080 هايوصل لبورت 80 جوة الكونتينر.

⚡ إزاي Docker بيعمل Port Mapping ؟

- بيستخدم iptables NAT rules
- أي Traffic جاي للـ Host على بورت 8080 .. بيتحول لبورت 80 على IP الكونتينر.

◇ شوف القاعدة: sudo iptables -t nat -L -n

هتلاقي Rule بيحول 8080 → لـ IP الكونتينر.

☑ الخلاصة:

- None : مفيش نت.
- Host : مشاركة شبكة الـ Host
- Bridge (افتراضي) = شبكة خاصة 172.17.x.x + veth + docker0
- Port Mapping = NAT rules بالـ iptables

Prerequisite CNI

1 الفكرة الأساسية

- قبل كده شفنا إزاي نعمل network namespace و نوصلهم بكوبري veth pairs + NAT + (bridge) علشان يخرجوا للإنترنت.
- Docker كان بيعمل نفس الكلام بشكل أوتوماتيك.
- لكن في الواقع كل Container Runtime زي (Kubernetes – Mesos – rkt) محتاج يعمل نفس الشغل.
- فبدل ما كل واحد يكتب الكود بنفسه ← اتعمل معيار موحد اسمه CNI

---

2 ليه محتاجينCNI ؟

- لو كل Runtime بيكتب طريقته .. هنلاقي chaos
- علشان كدا CNI وظيفته انه يعرف :
  - إيه المطلوب من الـ Runtime زي Kubernetes (how containers runtime should invoke plugins)
  - وإيه المطلوب من الـ Plugin زي Flannel, Calico (how plugins should be developed)

وبعدين يعمل CNI Standards that define how prog should be developed to solve network challenges

---

3 دور Container Runtime (زي Kubernetes) (اللي المفروض يعمله حسب المسؤوليات اللي CNI Standards محددهاله)

- يعمل network namespace لكل Pod
- يحدد أي شبكة هيتوصل عليها.
- ينادي على Plugin باستخدام أوامر قياسية زي:
  - ADD : أول ما Pod يتخلق.
  - DEL : لما Pod يتمسح .

◆ مثال تخيلي: bridge add <container-id> --netns=/var/run/netns/<container-id>

هنا Kubernetes بينادي على CNI plugin

```
bridge add 2e34dcf34 /var/run/netns/2e34dcf34
```

هنا:

- `2e34dcf34` = الـ Container ID (CID).
- `var/run/netns/2e34dcf34/` = الـ path اللي فيه الـ network namespace الخاص بالكونتينر ده.

بالتالي الأمر ده بيقول للـ bridge plugin:

"ضيف الكونتينر اللي ID بتاعه `2e34dcf34` ، والـ network namespace بتاعه موجود في المسار `var/run/netns/2e34dcf34/`."

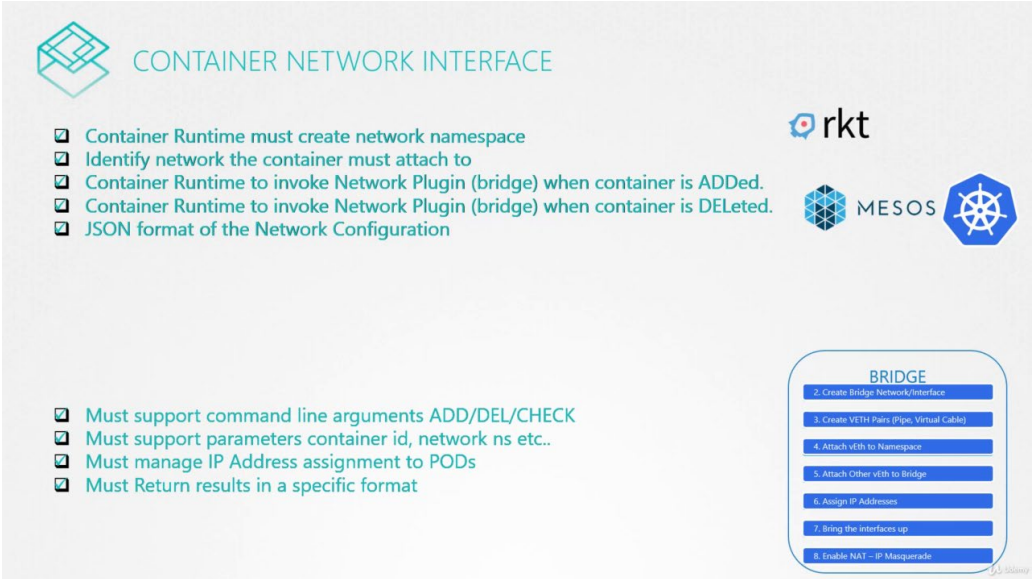
---

4 دور الـ Plugin (زي Bridge, Calico, Flannel) (اللي المفروض يعمله حسب المسؤوليات اللي CNI Standards محددهاله)

- لازم يدعم أوامر قياسية ADD, DEL, CHECK :
- ياخد Parameters زي:
  - Container ID
  - Network namespace
- يعمل:
  - إنشاء veth pair
  - توصيل طرف بالـ container namespace
  - توصيل الطرف التاني بالـ bridge أو بالشبكة المطلوبة
  - تعيين IP عن طريق IPAM
  - إضافة Routes عشان الكونتينرات يتكلموا مع بعض

◆ مثال Config للـ Plugin بصيغة JSON :

```
{
  "cniVersion": "0.3.1",
  "name": "mybridge",
  "type": "bridge",
  "bridge": "cni0",
  "isGateway": true,
  "ipam": {
    "type": "host-local",
    "subnet": "10.22.0.0/16"
  }
}
```



---

5 IPAM (IP Address Management)

- CNI بيعتمد على Plugins لإدارة IPs
- موجود افتراضي:
  - host-local : يوزع IPs من subnet محلي.
  - DHCP : ياخد IPs من DHCP Server
- مثال تشغيل Plugin DHCP :

```
{
  "cniVersion": "0.3.1",
  "name": "dhcp-test",
  "type": "bridge",
  "bridge": "cni0",
  "ipam": {
    "type": "dhcp"
  }
}
```

---

6 Plugins (المشهورة) :

- Bridge, MACVLAN, IPVLAN, VLAN (أساسية)
- Flannel, Calico, Weave, Cilium, VMware NSX, Infoblox (طرف ثالث)

❖ أي Runtime يقدر يشتغل مع أي Plugin طالما بيطبق CNI Standards

---

7 النقطة الخاصة بـ Docker

- Docker مش بيستخدم CNI لأن عنده Standard تاني اسمه CNM (Container Network Model
- علشان كده مش هتقدر تقول docker run –network=cni-bridge nginx
- لكن Kubernetes بيحلها كده:
  - يشغل Docker container بـ docker run –network=none nginx
  - وبعدين manually invoke the CNI Plugins علشان يكملوا الشغل :

bridge add <container-id> --netns=/var/run/netns/<container-id>

✅ الخلاصة

- CNI = Standard بيحدد إزاي Runtimes زي Kubernetes يتكلموا مع Plugins زي Flannel, Calico
- Runtime مسؤول عن + namespace اختيار الشبكة + استدعاء Plugin
- Plugin مسؤول عن veth + IP + Routes
- فيه IPAM Plugins لإدارة الـ Ips
- Docker ما بيستخدمش CNI ولكنه بيستخدم CNM والـ Kubernetes بيتعامل مع الموضوع ده عن طريق تشغيل الكونتينر بدون شبكة وبعدين يوصل CNI

Cluster Networking

1 مكونات الشبكة في الـ Master وWorker Nodes

- كل Node سواء Master أو Worker لازم يبقى متوصل على الأقل بواجهة شبكة (Interface) زي eth0
- كل Interface لازم يتخصص له IP Address
- كل Node لازم يبقى له hostname مميز مش متكرر
- كمان لازم يبقى له MAC Address مختلف.

👉 النقطة دي مهمة جدًا لو انت عملت Cloning لـ Virtual Machines ساعات بيتنسخ hostname & MAC addr، وده يعمل مشاكل في الـ Cluster

◆ أوامر عملية:

❖ عرض كل الـ interfaces : ip addr

❖ عرض الـ hostname : hostnamectl

❖ عرض الماك أدرس : ip link show

---

2 البورتات (Ports) المهمة في Kubernetes Cluster

فيه Ports لازم تتفتح سواء على Master أو Worker عشان المكونات تتواصل مع بعض:

❖ على الـ Master :

- 6443 : الـ API Server (بيخدم كل حاجة زي : kubectl, worker nodes, external users)
- 10251 : الـ kube-scheduler
- 10252 : الـ kube-controller-manager
- 2379 : الـ etcd server (لو عندك single master)
- 2380 : مستخدم في حالة وجود أكثر من Master للتواصل بينهم (ETCD clustering)

❖ على الـ Worker :

- 10250 : kubelet (بيشتغل على Master كمان)
- 30000 – 32767 : الـ NodePort Services (لما تفتح Service من نوع NodePort)

---

3 ملاحظات على الـ Firewall / Security Groups

- سواء كنت على On-premises (سيرفرات عندك) أو على Cloud (AWS, GCP, Azure) لازم تفتح البورتات دي في الـ Firewall أو الـ Security Groups
- أي مشكلة في تواصل الـ Cluster زي إن Pod مش قادر يكلم (API Server) .. فأول حاجة تبص عليها هي الـ Ports

◆ أوامر عملية:

❖ لو عايز تشوف البورتات المفتوحة محليًا : ss -tulnp

❖ لو بتستخدم iptables وعايز تشوف القواعد : iptables -L -n -v

❖ في Cloud زي AWS: تأكد من تعديل الـ Security Group الخاص بالـ Nodes عشان يفتح البورتات المطلوبة

---

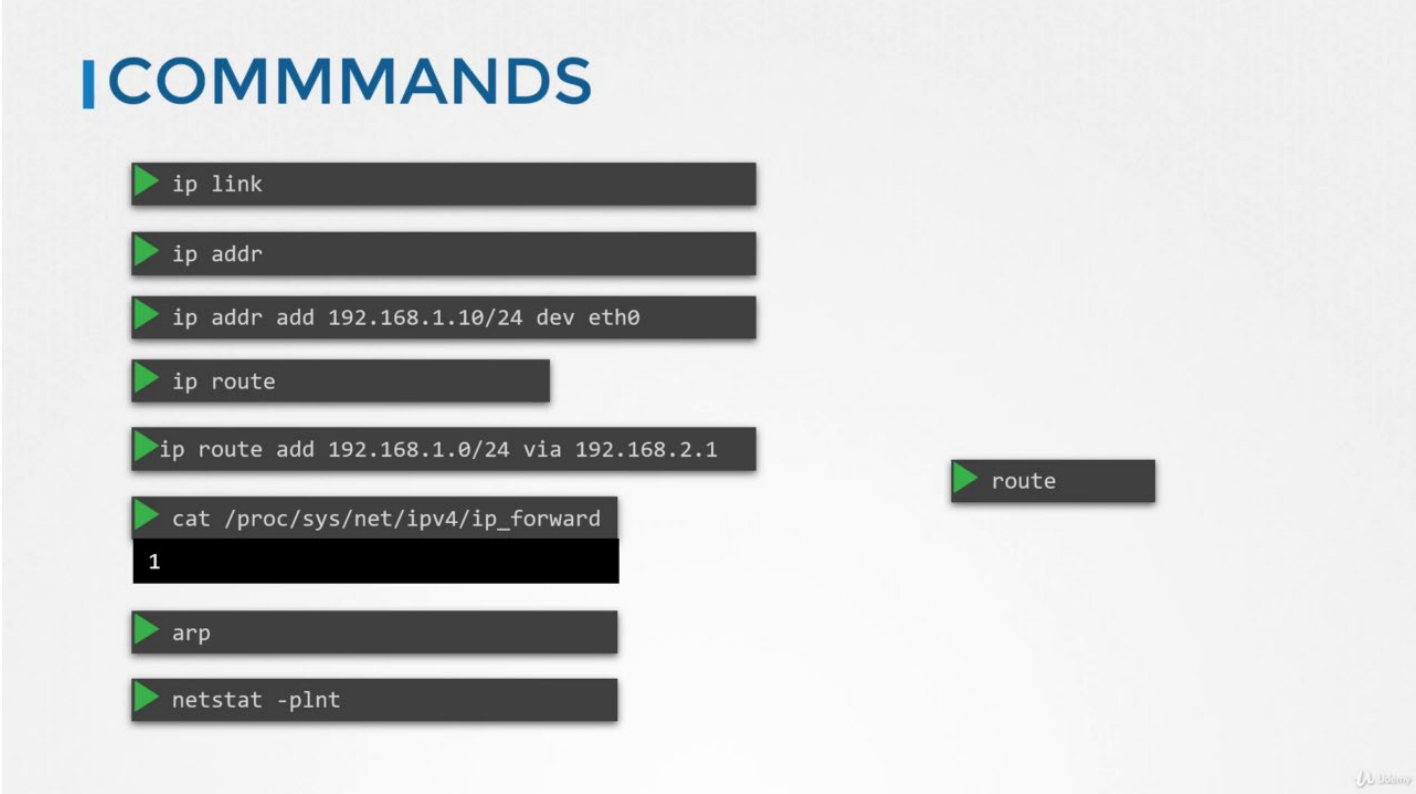
4 أوامر استكشافية (للتدريب العملي)

قبل ما ندخل في تمارين متقدمة، محتاج نتأكد من بيئة الـ Cluster :

❖ اعرض كل الـ Nodes بالـ IPs : kubectl get nodes -o wide

❖ اعرف الـ hostname والـ IP على كل Node : hostname -I

❖ اعرف إيه البورتات اللي الـ kubelet و الـ API server فاتحينها : netstat -tulnp | grep kube



---

5 الخلاصة

- كل Node لازم يبقى له IP – Hostname – MAC address مختلف.
- البورتات الأساسية للـ Control Plane هي (6443, 10251, 10252, 2379, 2380)
- البورتات الأساسية للـ Worker هي (10250, 30000–32767)
- افتح البورتات دي في الـ Firewall أو Security Groups
- أي مشكلة في التواصل ← راجع الـ Ports والـ Network setup

Pod Networking & CNI In K8S

◆ مقدمة

- بعد ما جهزنا Master و Worker Nodes وشغلنا المكونات(API server, etcd, kubelet ...) ، وفتحنا الـ Ports المطلوبة ... بقينا جاهزين ننشر التطبيقات.
- لكن في مشكلة لازم نحلها : إزاي الـ Pods هتتواصل مع بعض؟

◆ طبقة الشبكة الخاصة بالـ Pods

- عندنا طبقتين من الشبكات:
  1. شبكة الـ Nodes (اللي بيخلي كل السيرفرات تشوف بعض)
  2. شبكة الـ Pods (لازم كل Pod ياخد IP ويتكلم مع Pods تانية)

💡 الفكرة بقا ان Kubernetes مش بيقدم حل شبكات جاهز ..

يعني هو بيديني المتطلبات اللي المفروض تتعمل وانا دوري اني اعمل Implement a model يحقق المتطلبات دي (زي CNI Plugin) مثلاً

◆ متطلبات Kubernetes لشبكة الـ Pods

1. كل Pod لازم ياخد IP Address فريد.
2. كل Pod يقدر يكلم أي Pod تاني في نفس الـ Node على طول بالـ IP
3. كل Pod يقدر يكلم أي Pod على Node تانية باستخدام نفس الـ IP من غير NAT
4. طريقة توزيع الـ IPs و الـ Subnet بتاعتك، مش فارق لـKubernetes ، المهم تبقى موحدة وتشتغل.

◆ مثال عملي علي Cluster مكون من 3 Nodes

❖ Nodes network :
- Node1 → 192.168.1.11
- Node2 → 192.168.1.12
- Node3 → 192.168.1.13

❖ لكل Node بنعمل bridge network داخلي (زي ما عملنا في Linux Network Namespaces قبل كده)

❖ بنديله Subnet خاص بيه:
- Node1 → 10.244.1.0/24
- Node2 → 10.244.2.0/24
- Node3 → 10.244.3.0/24

◆ خطوات إعداد Pod Networking (Manual)

1. إنشاء Bridge على كل Node :

- ip link add name cni0 type bridge
- ip addr add 10.244.1.1/24 dev cni0 (مثال على Node1)
- ip link set cni0 up

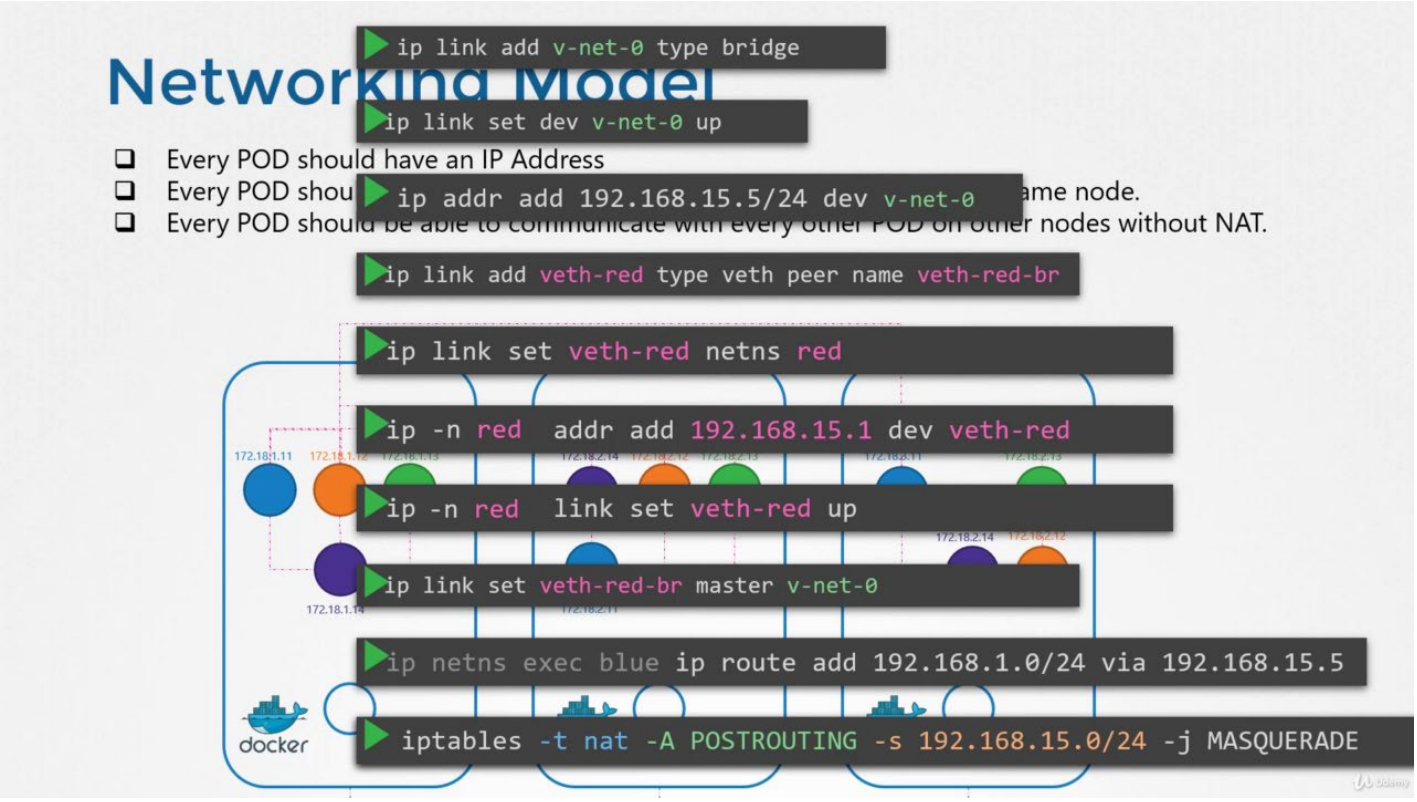
2. كل ما Container (Pod) يتولد:
   - نعمل veth pair (وصلة افتراضية) :
     - ip link add veth1 type veth peer name veth1-br

- نوصل طرف بالـ Container Namespace ، والطرف التاني بالـ Bridge
  - ip link set veth1 netns <container-ns> and ip link set veth1-br master cni0
- ندي للـ Pod IP من subnet الـ Node (مثلاً 10.244.1.2)
  - ip netns exec <container-ns> ip addr add 10.244.1.2/24 dev veth1
  - ip netns exec <container-ns> ip link set veth1 up

3. إضافة Default Route جوه الـ Pod :

- ip netns exec <container-ns> ip route add default via 10.244.1.1

➡️ كده الـ Pods على نفس الـ Node شايفين بعض.



◆ ربط الـ Pods بين الـ Nodes

- Pod في Node1 (10.244.1.2) مش هيعرف يوصل Pod في Node2 (10.244.2.2) .. لأنه subnet مختلف.

🔧 الحل : لازم نضيف Route في Routing Table بتاع Node1 :

- ip route add 10.244.2.0/24 via 192.168.1.12 # Route traffic for Node2 via Node1
- نفس الفكرة في Node2 و Node3

🔧 النتيجة: كل الـ Subnets تبقى جزء من شبكة كبيرة (10.244.0.0/16)

💡 في البيئات الكبيرة .. بدل ما تضيف Routes يدويًا على كل Node ، بتحطها على Router مركزي أو Firewalls/SDN

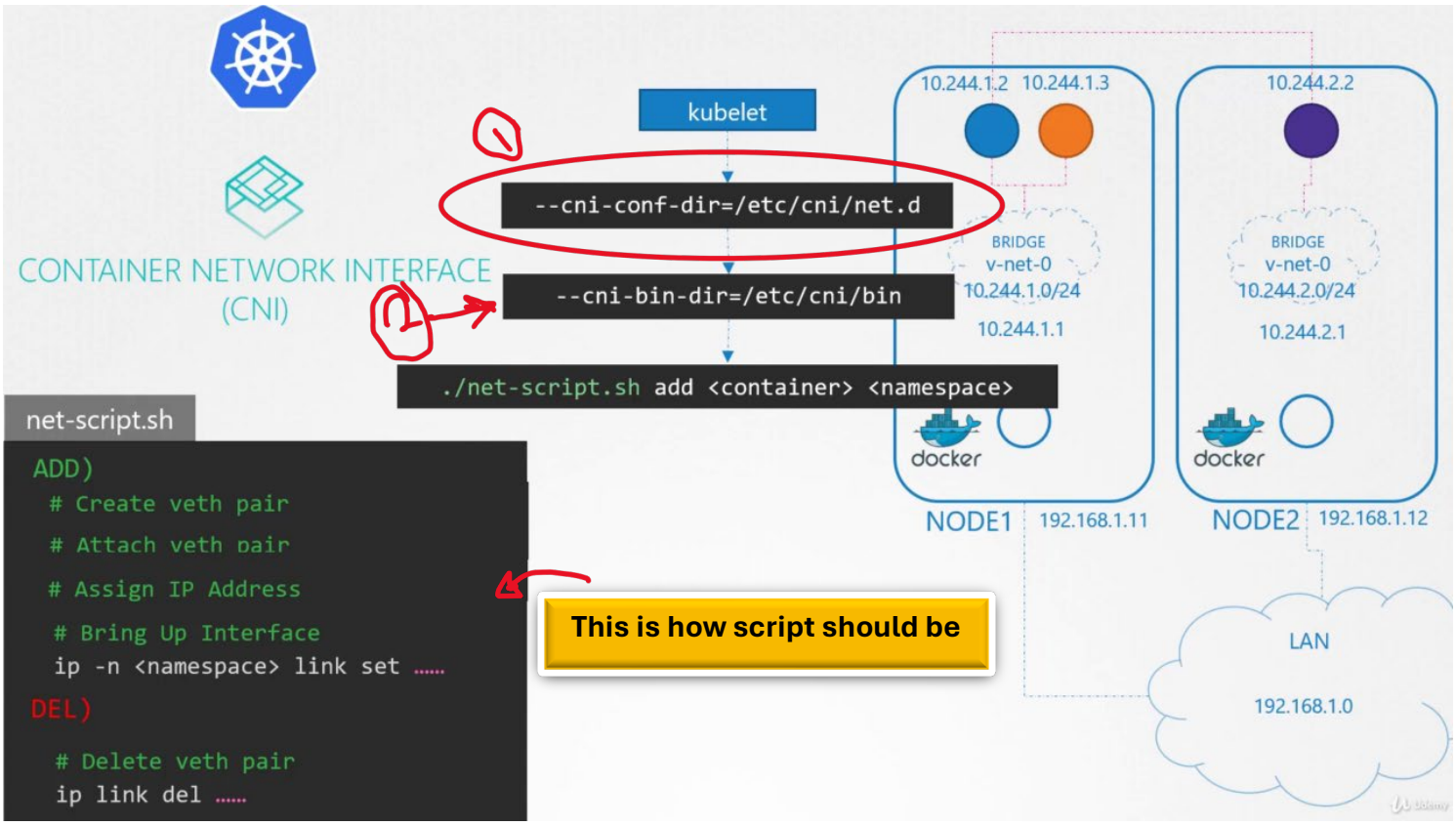
◆ مشكلة السكريبتات اليدوية

- إحنا عملنا Script بيربط Pod بالـ Bridge ويديله IP
- لكن Kubernetes بيشغل آلاف الـ Pods في الدقيقة وبالتالي مين هيفضل يشغل سكريبت يدوي؟

◆ هنا ييجي دور CNI (Container Network Interface)

- CNI = المعيار اللي بيربط Kubernetes بالـ Networking Plugins
- تكمن أهميته في Kubernetes في أنه يوفر طريقة لتزويد كل Pod بـ عنوان IP فريد، مما يسمح للـ Pods بالتواصل فيما بينها.
- بشكل عام، عندما تقوم بإنشاء Pod، فإن أول مهمة لـ CNI هي إعطاء الـ Pod عنوان IP.
- بعد ذلك، يقوم الـ CNI بتهيئة الشبكة لتمكين الـ Pod من التواصل مع الـ Pods الأخرى.
- بيقول:
  - أي سكريبت أو Plugin لازم يبقى فيه Add لما Pod يتولد و Del لما Pod يتشال
- **الـ kubelet هو المسؤول:**
  - **كل ما Pod يتولد، kubelet يبص في Config CNI و يشوف Plugin أو Script**

وبعدين يروح يشغله بالـ parameters المناسبة (Container ID + Namespace ID)



View kubelet options

```
ps -aux | grep kubelet
```

```
root      2095  1.8  2.4 960676 98788 ?        Ssl  02:32   0:36 /usr/bin/kubelet --bootstrap-kubeconfig=/etc/kubernetes/bootstrap-kubelet.conf --kubeconfig=/etc/kubernetes/kubelet.conf --config=/var/lib/kubelet/config.yaml --cgroup-driver=cgroupfs --cni-bin-dir=/opt/cni/bin --cni-conf-dir=/etc/cni/net.d --network-plugin=cni
```

```
ls /opt/cni/bin
```

```
bridge  dhcp  flannel  host-local  ipvlan  loopback  macvlan  portmap  ptp  sample  tuning
vlan  weave-ipam  weave-net  weave-plugin-2.2.1
```

```
ls /etc/cni/net.d
```

```
10-bridge.conf
```

◆ كيف تعمل CNI Plugins؟

- هناك العديد من CNI Plugins المتاحة، مثل Flannel وCalico وWeave Net وغيرها، وكل منها لديه طريقة خاصة لتطبيق معيار CNI.
- **بغض النظر عن الـ Plugin المستخدم، الفكرة الأساسية هي نفسها :**

1. يتم إنشاء (virtual network interface) داخل الـ Pod.
2. ثم يتم ربطها بـ (virtual bridge) موجود على (Node).
3. يتم تعيين عنوان IP فريد للـ Pod من خلال آلية تسمى IPAM (IP Address Management).

علشان الفكرة توضح اكتر فإزاي الـ Pods بتتواصل مع بعضها في وجود الـ CNI Pugins ؟! (سؤال انترفيو)

- **التواصل داخل نفس (Node):**
  - عندما يتحدث Pod مع Pod آخر على نفس الـ Node، فإن كليهما متصلان بنفس الـ Bridge
  - الـ Bridge يعمل مثل (network switch)، مما يسمح للـ Pods بالتواصل مباشرةً مع بعضها البعض.
- **التواصل بين (Nodes) المختلفة:**
  - عندما يحتاج Pod على " Node1" للتحدث مع Pod على " Node2"، يتم استخدام تقنيات التوجيه (routing) والتنكر (masquerading).
  - يتم تغليف حركة البيانات (traffic) وإرسالها عبر (tunnel) بين 2 Nodes، ثم يتم فك تغليفها وتوجيهها إلى الـ Pod المستهدف.
  - هذا يضمن أن الـ Pods لا تحتاج إلى معرفة بنية الشبكة الأساسية، وأنها يمكن أن تتواصل ببساطة باستخدام عناوين IP الخاصة بها.

◆ تحليل ملف CNI Config

- مثال لملف CNI Config بسيط باستخدام الـ plugin من نوع bridge.
- أهم الحقول التي تم ذكرها:
  - Name: اسم الشبكة (مثل mynet).
  - type : نوع الـ CNI plugin المستخدم (في هذه الحالة، bridge).
  - isGateway : يحدد ما إذا كانت واجهة Bridge الافتراضية ستحصل على عنوان IP لتعمل كـ (gateway).
  - ipMasq: يحدد ما إذا كان يجب إضافة قاعدة NAT (Network Address Translation) لتمكين التنكر (IP masquerading).

❖ ipam: قسم خاص بإدارة عناوين IP، يحدد نوعها (host-local) ونطاق العناوين (subnet) الذي سيتم استخدامه

```
ls /etc/cni/net.d
10-bridge.conf
```

```
cat /etc/cni/net.d/10-bridge.conf
{
    "cniVersion": "0.2.0",
    "name": "mynet",
    "type": "bridge",
    "bridge": "cni0",
    "isGateway": true,
    "ipMasq": true,
    "ipam": {
        "type": "host-local",
        "subnet": "10.22.0.0/16",
        "routes": [
            { "dst": "0.0.0.0/0" }
        ]
    }
}
```

◆ الخلاصة

- كل Pod لازم يبقى له IP فريد، ويتكلم مع Pods في نفس Node ومن Nodes تانية بدون NAT
- الحل اليدوي Bridges + veth + routes
- الحل الواقعي CNI Plugins (Flannel, Calico, WeaveNet ...)
- CNI بيسمح للـ kubelet يشغل Plugin أو سكريبت بشكل أوتوماتيك كل ما Pod يتولد أو يتمسح.

CNI Weave

مقدمة CNI Weave :

- الحل مبني على CNI (Container Network Interface)
- بدل ما نكتب سكريبتات يدوية زي ما عملنا قبل كده، نستخدم Weave CNI Plugin

---

📌 فكرة الشبكات اليدوية .Weave VS :

- السكربت اليدوي اللي عملناه قبل كده:
  - كان بيبني Routing Tables عشان أي Pod على Node1 يعرف يوصل لـ Pod على Node2
  - ده ينفع لو عندك 2 أو 3 نود بس.
- المشكلة : مع مئات النودز وآلاف البودز Routing Tables .. مش هتستحمل.
- هنا ييجي دور :Weave بحيث يعمل شبكة Overlay ويشيل منك الهم ده.

📌 مثال علشان تفهم الدنيا أكتر .. تخيل إنك عندك الاتي :

- الكلستر = الشركة.
- النودز = فروع الشركة.
- البودز = المكاتب جوه الفروع.
- فالـ Weave Agents = هي شركة الشحن اللي عندها وكلاء في كل فرع.

🔹 وبالتالي أي بود عاوز يبعت Packet لبود تاني:

- الـ Agent اللي هو (وكيل Weave) هايستلم الباكت ثم يغلفها ← (Encapsulation) ثم يبعتها للوكيل التاني .. والوكيل التاني ده يفكها ← (Decapsulation) ثم يوصلها صح.

📌 طريقة عمل Weave :

- **لما تعمل Deploy للـ Weave :**

1. بيعمل Bridge اسمه weave على كل Node : ip link show weave
1. يدي Range IPs للبودز (مثال: 12/10.32.0.0).

   ممكن تشوف الرينج الفعلي بـ : kubectl describe pod -n kube-system -l name=weave-net
2. بينزل (Peers) Agents على كل Node
3. كل Agent بيحتفظ بـ Topology كاملة (عارف كل بود وكل IP في الكلستر)

📌 بالنسبة للـ Routing & Paths وطريقة عملهم مع الـ Weave :

- أي Pod ممكن يتوصل بأكتر من Bridge :
  - مثال weave + docker0
- الـ Route Table جوه الـ Container هو اللي بيحدد أي Path يمشي فيه.
- Weave بيظبط الـ Routing بحيث البود يعرف يوصل للـ Agent ، والـ Agent يتولى الباقي.

📌 تشوف الـ Routes جوه بود : kubectl exec -it <pod-name> -- ip route

---

📌 Encapsulation & Decapsulation

- لو pod على Node1 بيكلم pod على Node2 :

1. Weave Agent يمسك الباكت.
2. يغلفها (Encapsulation) ويحط عنوان Node2
3. يبعتها
4. Weave Agent في Node2 يفك الغلاف (Decapsulation)
5. يوصل الباكت للبود الهدف.

📌 نشر Weave في الكلستر

"We can deploy weave and weave peers as services or daemons on each node in cluster manually. But if k8s is already setup then the easiest way is to be deployed as DaemonSet pods in the cluster"

✅ بعد ما الكلاستر يجهز .. هانعملهم Deploy as Pods :

```
kubectl apply -f https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version | base64 | tr -d '\n')
```

شرح الامر :

- k8s-version : الـ Weave YAML محتاج يعرف إصدار الكلاستر Kubernetes عشان يديك Config متوافق معاه فعشان كده إحنا بنبعت له الإصدار كـ Query Parameter في الرابط.
- $ ('kubectl version | base64 | tr -d '\n) : kubectl version بيطبع نسخة الـ Client (الـ kubectl اللي عندك) + نسخة الـ Server (API Server بتاع الكلاستر). والإخراج بيكون JSON أو Text.
- base64 : بياخد الإخراج بتاع kubectl version ويشفره Base64.
  الهدف: يبقى نص مضغوط ومفهوم للسيرفر بدل ما يكون JSON معقد.
- 'tr -d '\n : أداة tr بتستخدم لتعديل النصوص.
  -d '\n' = يشيل أي New Line من النص (عشان Base64 ساعات بيطلع النص على كذا سطر).
  النتيجة = string في سطر واحد.

✅ متنساش تعمل الـ pod من نوع DaemonSet ؟

- عشان يضمن وجود Pod من Weave على كل Node
- تقدر تشوفهم : `kubectl get pods -n kube-system -l name=weave-net -o wide`

📌 Troubleshooting

- تشوف اللوجز لأي Pod من Weave : `kubectl logs -n kube-system -l name=weave-net`
- تشوف إذا الـ Bridge موجود فعلاً على النود : `ip addr | grep weave`
- تتأكد إن البود واخد IP من رينجWeave : `kubectl get pods -o wide`

📌 إضافات مهمة:

- Weaveمش بس بيعمل Networking .. دا بيوفر كمان Encryption للـ Traffic بين النودز (لو فعلت الميزة).
- Weave بيدعم Network Policiesعشان تحدد مين يقدر يكلم مين.
- Weave من أسهل الـ CNIs في التشغيل (مقارنة بـ Calico أو Flannel)

✅ الخلاصة:

Weave = شبكة Overlay بتعمل Automation لكل Routing & Encapsulation بين البودز على مختلف النودز.

- سهلة التنزيل (kubectl apply)
- بتشتغل بـ DaemonSet Pod على كل Node
- Agents بيتبادلوا المعلومات وبيحتفظوا ب Topology كاملة.
- بتديك IP Range موحد لكل الكلاستر.

IP Address Management (IPAM)

مقدمة عن IP Address Management (IPAM)

الموضوع هنا مش بيتكلم عن الـ IP بتاع الـ Nodes نفسها (دي مسئوليتك تديرها يدويًا أو بـ DHCP خارجي).
لكن بيتكلم عن:

- إزاي الـ Pods بياخدوا IP
- مين اللي بيضمن إن مفيش IP مكرر؟
- فين بتتخزن المعلومات الخاصة بالـ IPs ؟

مين المسئول عن إدارة الـ IP للـ Pods ؟

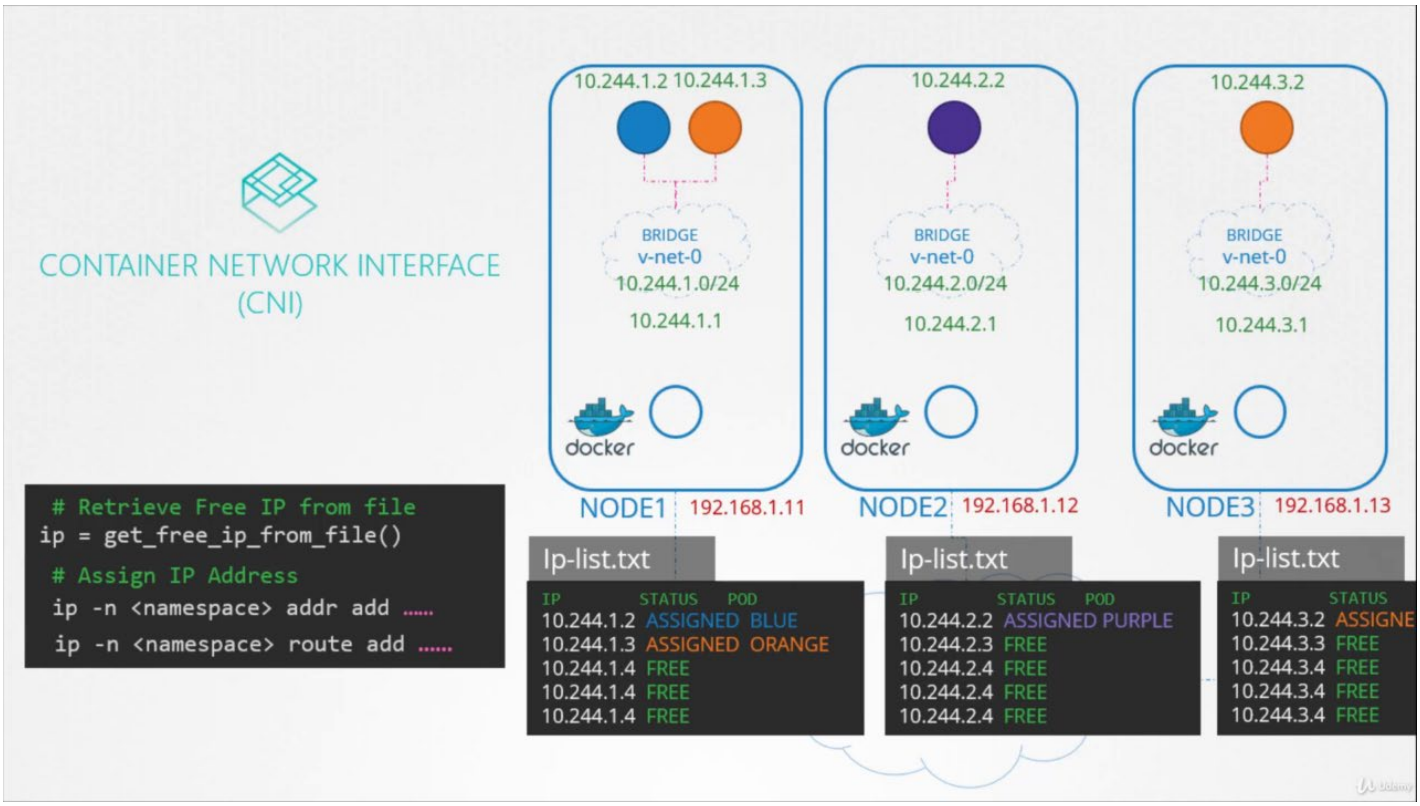
- الـ CNI Plugin هو اللي مسئول يوزّع IPs
- الـ Kubernetes نفسه مش بيدير IP للـ Pods ، هو بيعتمد على الـ CNI
- أي Network Plugin زي (Weave, Flannel, Calico) هو اللي بيعمل IPAM

مثال من الـ CNI البسيط (اللي كنا عاملينه قبل كده)

في الـ CNI Script البسيط اللي كتبناه:

- كان فيه كود بيعمل Assign لـ IP للـ Network Namespace الخاص بالـ Container
- بس انا ضحكت عليك علشان كنا ممكن نخزن الـ IPs المستهلكة في ملف على كل Node علشان نتجنب التكرار.

📌 مثال : ملف /var/lib/cni/networks/<mynetwork>/last_reserved_ip
ده بيتخزن فيه آخر IP تم توزيعه.



_

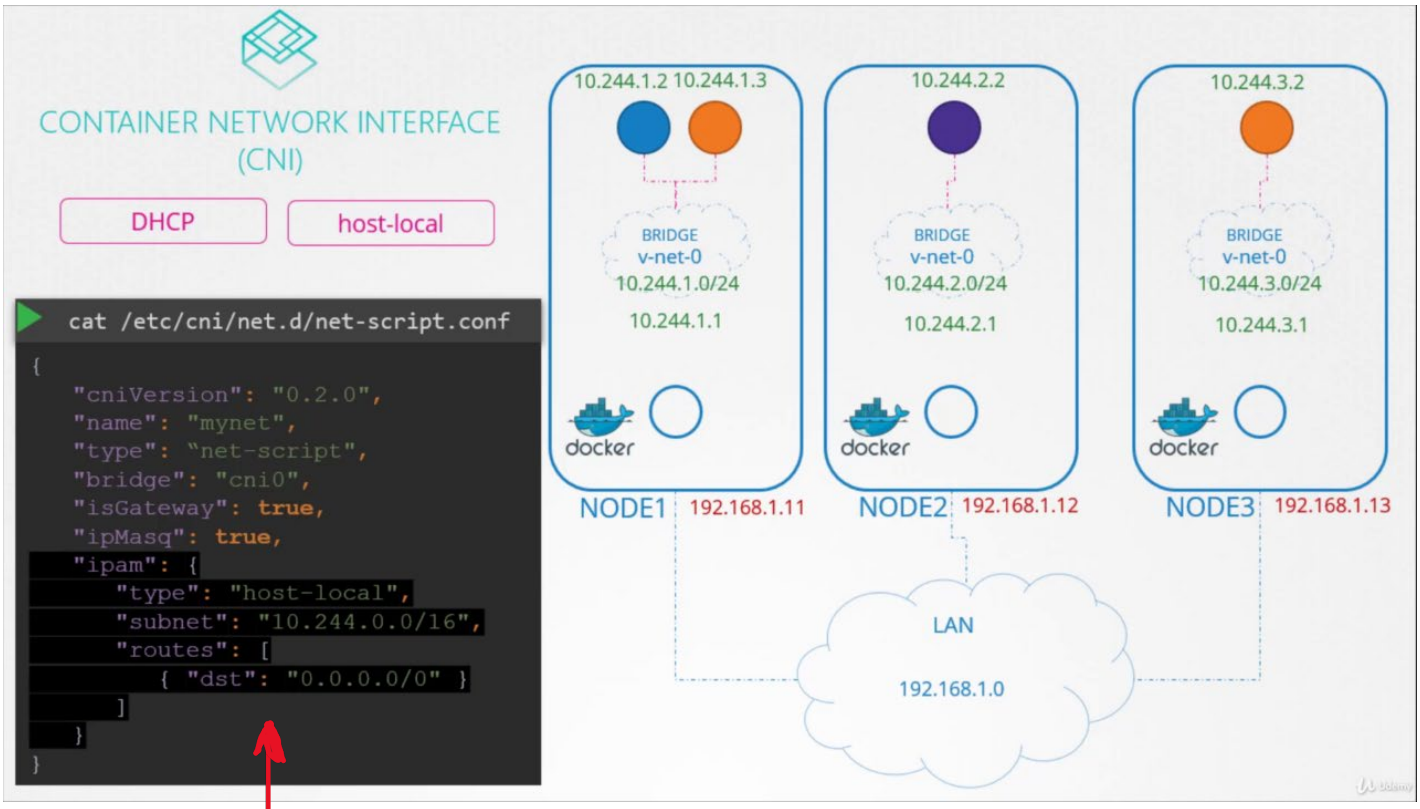
حلول جاهزة من CNI Plugins

الـ CNI نفسه جاي مع Plugins جاهزة لإدارة IPs بدل ما تكتبها بنفسك :

- host-local plugin : بيخزن IPs محليًا في كل Node

- فيه Plugins تانية بتسمح بـ Centralized IPAM لو عندك Cluster كبير .. زي Calico IPAM أو Whereabouts بتعمل توزيع IP بشكل مركزي (يعني كل الـ Nodes تسحب من Pool واحد بدل ما كل Node تخزن محلي).

📌 أمر عملي لتشوف الـ Config : cat /etc/cni/net.d/10-weave.conflist



إعدادات IPAM في CNI

في الـ CNI Config File فيه Section اسمه ipam بيتحدد فيه:

- نوع الـ Plugin (host-local أو غيره)
- الـ Subnet اللي هيتوزع منه.
- الـ Routes اللي البود يستخدمها.

Weave CNI Plugin و إدارة الـIP

- الـ Weave بيوزع الـ IPs بشكل افتراضي من 10.32.0.0/12
- ده معناه إن عندك حوالي مليون عنوان IP متاح للـ Pods
- Weave بيقسّم الرينج ده أوتوماتيك بين الـ Nodes
  - كل Node بياخد Range صغير خاص بيه.
  - ولما Pod يتنشأ على Node فبياخد IP من Range بتاع الـ Node

📌 أمر عملي تشوف الـ Range اللي Weave وزعه:

kubectl exec -it <weave-pod-name> -n kube-system -- /home/weave/weave --local status ipam

---

تعديل الـ Range في Weave

لو مش عاجبك الـ Range الافتراضي : (10.32.0.0/12)

- ممكن تخصص Subnet تاني أثناء Deploy Weave
- مثلًا:

```
kubectl apply -f "https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version | base64 |
tr -d '\n')&env.IPALLOC_RANGE=192.168.0.0/16"
```

كده بدل ما يدي IPs من 10.32.0.0/12 .. هيدّي من 192.168.0.0/16

ملاحظات مهمة

- لازم كل الـ Pods يبقوا في Subnet واحدة أو Subnets مترابطة علشان يتواصلوا.
- لو عندك أكتر منCNI Plugin ، كل واحد ممكن يدير IP Range خاص بيه.
- الـ IPAM مش بس بيحدد الـIP ، ده كمان بيحدد الـ Routes اللي البود هيمشي بيها.

✅ **الخلاصة:**

- الـ IPAM مسئولية الـ CNI Plugin
- Weave بيديرها أوتوماتيك ويقسم الـ IPs بين الـ Nodes
- ممكن تتحكم في الـ Subnet بنفسك.
- كمسؤول نظام لازم تفهم مكان تخزين الـ IPs (في ملفات محلية أو Centralized) عشان تحل مشاكل التكرار أو التعارض.

Services Networking

Recap عن Pod Networking

- الـ Pod لما بيتعمل له Deploy بياخد:
  - Namespace خاص + Network Interface + IP Address من الـ Subnet بتاع الـ Node
- الشبكة بتاعة الـ Pods في Cluster كبير بتتوصّل ببعض يا إما عن طريق Routing Tables أو Overlay Networks زي Weave, Flannel

📌 أمر عملي تشوف الـ Pods والـ IPs : kubectl get pods -o wide

ليه بنحتاج Services ؟

- صعب تعتمد إن الـ Pods يتكلموا مع بعض مباشرة (لأن الـ IP بتاع Pod ممكن يتغير)
- علشان كده بنستخدم Service
- الـ Service بياخد اسم + IP ثابت (Virtual IP)

📌 مثال: kubectl expose pod myapp --name=my-service --port=80 --target-port=8080

أنواع الـ Services

1. **ClusterIP (Default) :**
   - بيدي IP ثابت جوه الـ Cluster
   - Pods تانية جوه Cluster هي بس اللي تقدر توصله.
   - مثال Database service :

❖ kubectl expose deployment db --port=3306 --target-port=3306 --type=ClusterIP

2. **NodePort :**
   - البورتس اللي بيحجزها بتكون في الـ Range: 30000 - 32767 .. لو اخترت Port برّا الرينج هيرفض.
   - الـ Static Port بيتفتح على كل Nodes، مش بس على النود اللي فيها الـ Pod .. يعني تقدر تدخل على أي NodeIP:NodePort وتوصل للبود.
   - يخلّي الخدمة متاحة من برا Cluster
   - مثال Web App :

❖ kubectl expose deployment web --port=80 --target-port=8080 --type=NodePort

3. **LoadBalancer :**
   - زي NodePort بس معاها Load Balancer خارجي (بتاع مزوّد السحابة : AWS/GCP/Azure...).
   - الـ Cloud Controller Manager بيطلب من الكلاود يعمل LB ويخصص External IP (عام).
   - تحت الغطا .. Kubernetes بيفتح NodePort برضه، والـ LB بيحوّل عليه، ثم kube-proxy يوزّع على الـ Pods.
   - مثال سريع (nginx) :

❖ kubectl expose deploy web --port=80 --target-port=80 --type=LoadBalancer

4. **ExternalName :**
   - مش بيعمل ClusterIP ولا iptables ولا Load Balancer.
   - هو Alias DNS داخل الكلستر (CNAME) لاسم خارجي.
   - أي Pod ينادي http://mydb.default.svc.cluster.local (مثلًا الـ DNS يردّ بـ CNAME إلى db.example.com فالكلاينت يحلّها عادي ويكلم الوجهة الخارجية.

Service IP مش حقيقي :

- الـ ClusterIP مجرد Virtual IP (VIP)
- مفيش process قاعد عليه.
- كله بيتم بـ iptables أو ipvs Rules او الـ userspace اللي kube-proxy بيضيفها.

```
kube-proxy --proxy-mode [userspace | iptablkes | ipvs ] …
```

مكون جديد kube-proxy :

- موجود على كل Node
- بيراقب الـ API Server (أي Service جديدة تتعمل)
- يضيف Rules في النظام (iptables أو ipvs) علشان يربط الـ Service IP بالـ Pod IP

📌 أوامر عملية :

- تشوف الـ kube-proxy كـ Pod : kubectl get pods -n kube-system -o wide | grep kube-proxy
- تشوف Rules اللي عملها: sudo iptables -t nat -L -n -v | grep <service-name>

الـ Service مش Object حقيقي!

- مفيش Container أو Process اسمه "Service"
- هو Virtual Object
- اللي بيخلّيه يشتغل هو الـ Rules اللي kube-proxy بيضيفها.

📌 تشبيه :

Service = عنوان بريد ثابت.

Pod = المكتب الحقيقي اللي بيستقبل الرسائل.

kube-proxy = مكتب البريد اللي بيحوّل الرسائل للعنوان الصح.

Service IP Range

```
kube-api-server --service-cluster-ip-range ipNet (Default: 10.0.0.0/24)
```

- الـ API Server بيحدد Range للـ Services عن طريق : --service-cluster-ip-range
- مثال : 10.96.0.0/12 يعتبر هو (Default في K8s)
- لازم مايتعارضش مع (--pod-network-cidr) Pod Network Range

📌 أوامر عملية:

- تشوف Service IP : kubectl get svc
- تشوف Pod CIDR : kubectl cluster-info dump | grep -m 1 cluster-cidr

---

بالنسبة للـ ClusterIP Service ففيه iptables Rules (اللي بيضيفها kube-proxy)

- بيضيف DNAT Rules :

📌 السيناريو:

- عندك Service اسمه db-service واخد IP داخلي (ClusterIP) مثلًا: 10.103.132.104:3306
- البود الحقيقي اللي وراها هو 10.244.1.2:3306.
- المشكلة : الـ Service نفسها مش كيان حقيقي (مفيش process أو namespace عليها).
- الحل : kube-proxy بيضيف Rules في iptables (في جدول NAT) تقول:
- أي حد يحاول يكلّم السيرفس 10.103.132.104:3306 .. اعملَه (DNAT) وحوّله على البود 10.244.1.2:3306

📌 أمر عملي : sudo iptables -t nat -L KUBE-SERVICES -n -v

بالنسبة للـ NodePort Services ففيه iptables Rules (اللي بيضيفها kube-proxy)

📌 السيناريو:

- عملت Service نوعها NodePort
- Kubernetes خصصلها مثلًا 30080 Static Port على كل الـ Nodes
- بس في نفس الوقت .. جوّا الكلستر الـ Service لسه واخدة IP داخلي (ClusterIP) زي 10.103.200.5:80

- البود الحقيقي اللي وراها هو 10.244.2.7:80
- المطلوب: أي حد من برا الكلستر يضرب IP أي Node على البورت 30080 فيقدر يوصل للـ Pod في الآخر.
- إزاي؟
- kube-proxy بيضيف Rules إضافية في iptables تقول:
- أي ترافيك جاي للـ NodeIP:30080 .. اعملَه (DNAT) وحوّله على ClusterIP:80 وبعدين اعملَه (DNAT) تاني وحوّله تاني علي PodIP:80

📌 أمر عملي : kubectl describe svc my-service

🔍 Troubleshooting

- Logs بتاعة kube-proxy :

kubectl logs -n kube-system -l k8s-app=kube-proxy or cat/var/log/kube-proxy.log

- لو مش شايف Rules ، زوّد Verbosity :: --v=4

✅ الخلاصة :

- الـ Pods بتاخد IP من الـ Pod CIDR (عن طريق CNI)
- الـ Services بتاخد IP من Service CIDR (ولاحظ ان الـ API Server هو اللي بيحدد)
- مفيش Service Process حقيقي، الـ kube-proxy بيعمل Rules (iptables/ipvs) علشان يربط الـ Service IPs بالـ Pod IPs
- NodePort = ClusterIP + Port على كل Node

بعض الملاحظات المهمة علي LoadBalancer Service

- externalTrafficPolicy:
  - Cluster (الافتراضي): يقدر يعيد توجيه الطلب لأي Node جوّه الكلستر (أقل فقدان حِمل).
  - Local : يوجّه بس للـ Pods الموجودة على نفس الـ Node اللي استقبل الترافيك وبيحافظ على Client IP الحقيقي، لكن لازم يكون عندك بودز موزّعة كويس.
- loadBalancerSourceRanges : قوائم CIDR مسموح لها تدخل (Firewallعلى مستوى الـ Service)
- Annotations : حسب الـ Cloud Provider (أمثلة مختصرة) :
  - AWS NLB داخلي:
    - service.beta.kubernetes.io/aws-load-balancer-type: "nlb"
    - service.beta.kubernetes.io/aws-load-balancer-internal: "true"
  - Azure داخلي :
    - service.beta.kubernetes.io/azure-load-balancer-internal: "true"
  - GKE داخلي :
    - cloud.google.com/load-balancer-type: "Internal"
- allocateLoadBalancerNodePorts (Boolean) : تقدر تمنع فتح NodePorts لو الـ LB مش محتاجها (بس دي سيناريوهات متقدمة).
- على بيئات Bare-metal : مفيش Cloud LB افتراضي وبالتالي هااتستخدم MetalLBعلشان يديك External IPs :
  - تثبيت MetalLB → ثم تعريف AddressPool ثم تعمل Service نوع LoadBalancer فتلاقي IP خارجي من الـ Pool

```yaml
apiVersion: v1
kind: Service
metadata:
  name: web
spec:
  type: LoadBalancer
  selector:
    app: web
  ports:
    - port: 80
      targetPort: 80
      protocol: TCP
  externalTrafficPolicy: Cluster   # أو Local (هنتكلم عنها تحت)
  loadBalancerSourceRanges:        # تقييد الوصول (اختياري)
    - 203.0.113.0/24
  # loadBalancerIP: 203.0.113.10   # ثابت محجوز في مزوّد السحابة IP (اختياري)
```

Troubleshooting سريع

- **EXTERNAL-IP واقف على pending :**
  1. **تأكد إن الكلستر مربوط بمزوّد سحابة صح (Cloud Controller)**
  2. **على Bare-metal : فعّل MetalLB**
  3. **بص على Events : kubectl describe svc web**
  4. **راجع Security Groups / NSGs / Firewalls على البورت المطلوب.**
- **Latency أو فقدان IP الحقيقي : جرّب externalTrafficPolicy: Local**
- **مش واصل على الـ Pods اتأكد إن الـ Pods Ready ، والـ selector بتاع الـ Service مطابق للـ labels**

بعض الملاحظات المهمة على ExternalName Service

- **لا يوجد:**
  - **spec.selector ، ولا ports، ولا Endpoints لهذا النوع.**
  - **عنوان ClusterIP هيبقى <none>**
- **لازم تستخدم اسم الخدمة في الـ DNS لأن مفيش IP تقدر تضربه مباشرة.**
- **حط اسم DNS حقيقي في externalName (يعني ما تحطّش IP نصيًا لأن ده مش اسم DNS)**
- **Use cases :**
  - **تربط ما بين تطبيق جوّه الكلستر وخدمة خارجية باسم ثابت.**
  - **تعمل alias قصير لخدمة SaaS طويلة الاسم.**
- **قيود:**
  - **مفيش Load Balancing من K8s نفسه (الـ LB لو موجود بيكون عند الجهة الخارجية)**
  - **بيتوقف على DNS caching/TTL لو العنوان الخارجي اتغيّر.**
  - **ما ينفعش تستخدمه عشان "تفتح بورت" لبرا لأن النوع ده Service Discovery فقط.**

Troubleshooting سريع

- **الاسم مش بيتحل:**
  - **راجع CoreDNS و /etc/resolv.conf جوّه البود.**
  - **جرّب : kubectl -n kube-system logs deploy/coredns**
  - **اتأكد إن externalName اسم DNS صالح وبيرجع A/AAAA في النهاية.**
- **TLS/SNI :**
  - **لو بتستخدم HTTPS على اسم خارجي، لازم تستخدم الاسم نفسه (مشIP) عشان الشهادة تبقى صالحة.**

ملخص الفرق السريع

- **LoadBalancer : عنوان خارجي حقيقي + فتح ترافك من الإنترنت/شبكة خارجية .. مناسب للويب/واجهات عامة.**
- **ExternalName : Alias DNS داخلي بس .. مناسب للـ Service Discovery لوجهات خارج الكلستر من غير أي نتوركينج / Proxies إضافية من K8s**

DNS In Kubernetes

مقدمة

- DNS جزء أساسي في أي Cluster Kubernetes
- DNS هو اللي بيربط الأسماء بالـ IPs → وده مهم جدًا عشان البودز والـ services يتواصلوا.
- لو إنت مش فاهم DNS أساسًا خدلك بصة علي سكشن Prerequisite DNS
- في Kubernetes ، السيرفر اللي بيعمل دور الـ DNS اسمه (CoreDNS) وده بيتسطب افتراضيًا مع الكلاستر

📌 أمر عملي: تشوف إن CoreDNS شغال : kubectl get pods -n kube-system -l k8s-app=kube-dns

---

ليه DNS مهم في Kubernetes ؟

- Kubernetes بيهتم بالـ Pod-to-Pod و Pod-to-Service communication.
- كل Pod بياخد IP خاص بيه، لكن الـ IP ده ممكن يتغير بعد restart → عشان كده لازم اسم ثابت.
- الحل: DNS records بتتخزن جوه CoreDNS.
- يعني Services بتاخد اسم ثابت + IP افتراضي، و CoreDNS بيربط الاسم ده بالـ IP

📌 مثال:

تعمل expose ل Pod عشان يبقى Service :

❖ kubectl expose pod web --name=web-service --port=80

تشوف الـ Service اتعمل إزاي : kubectl get svc web-service

مثال عملي (Pod + Service)

- Pod اسمه test بـ IP = 10.244.1.5
- Pod تاني اسمه web بـ IP = 10.244.2.5
- عملنا Service اسمه web-service وخد ClusterIP ثابت (مثال : 10.107.37.188).
- دلوقت CoreDNS هأينشئ Record للـ Service اسمه web-service ويربطه بالـ IP

📌 ومن جوه test pod تقدر توصل للـ web-service كده : kubectl exec -it test -- curl http://web-service

---

الـ Namespaces والـ DNS

❖ لو كل الـ Pods والـ Service في نفس الـ namespace
  - تقدر توصله بالاسم مباشرة : curl http://web-service

❖ لو Service موجود في Namespace مختلف وليكن اسمه (مثلاً apps) :
  - لازم تحدد الـ Namespace : curl http://web-service.apps

🔎 لاحظ ان CoreDNS بيبني الـ DNS Hierarchy كده: <service-name>.<namespace>.svc.cluster.local

📌 مثال : web-service.apps.svc.cluster.local

📌 تشوف الـ Namespaces : kubectl get pods --all-namespaces

Fully Qualified Domain Name (FQDN)

- أي Service ليها اسم كامل (FQDN) : <service-name>.<namespace>.svc.cluster.local
- ده بيديك اسم ثابت guaranteed حتى لو الـ IP اتغير.

Hostname	Namespace	Type	Root	IP Address
web-service	apps	svc	cluster.local	10.107.37.188
10-244-2-5	apps	pod	cluster.local	10.244.2.5

DNS Records للـ Pods

- بشكل افتراضي : Kubernetes مابيعملش Records للـ Pods
- لكن ممكن تفعّلها في CoreDNS
- الاسم بيتكون من :
  - الـ IP بيتحوّل لاسم (النقط بتتحول الي داش).
  - الـ Namespace
  - كلمة pod
  - root domain = cluster.local.

📌 مثال:

اسم Pod بـ IP = 10.244.1.5 في الـ namespace اللي اسمها بيقى default: 10-244-1-5.default.pod.cluster.local

---

✅ الخلاصة

- CoreDNS بيعمل Records للـ Services تلقائيًا.
- Pods تقدر توصّل لبعض بالاسم بدل الـ IP
- الـ DNS Structure : <service>.<namespace>.svc.cluster.local
- ممكن تضيف DNS للـ Pods (لكن مش افتراضي)
- DNS بيسهّل Service Discovery جوه الكلاستر.

📌 أوامر عملية للتست:

- kubectl exec -it test-pod -- nslookup web-service
- kubectl exec -it test-pod -- curl http://web-service

CoreDNS In Kubernetes

مقدمة

- الهدف: نفهم إزاي الـ Pods والـ Services يقدروا يتواصلوا جوه الكلاستر عن طريق DNS
- DNS هنا بيشتغل كـ مترجم أسماء IPs
- Kubernetes بيوفر CoreDNS (خلي بالك انه قبل v1.12 كان اسمه kube-dns)
- CoreDNS بيتنصب جوه الـ namespace kube-system كـ Pods (غالبًا 2 علشان الـ redundancy)

📌 أمر عملي تشوف الـ CoreDNS : kubectl get pods -n kube-system -l k8s-app=kube-dns

---

المشكلة الأساسية ليه محتاجين DNS ؟؟!

- كل Pod بياخدIP ، بس الـ IP بيتغير لما الـ Pod يتعمله restart
- ممكن نظريًا نحلها بـ /etc/hosts بحيث (نكتب الـ PodName والـ IP لكل بود عالبود الاخر) ، بس ده كارثة مع آلاف البودز اللي بتتغير طول الوقت.
- الحل الواقعي : نستخدم DNS Server مركزي.
  - كل Pod يتوجه للـ DNS Server ده (بيكون واخد IP افتراضي 10.96.0.10)
  - أي Pod جديد .. بيتسجل له Record في الـ DNS
  - وبكده أي Pod يقدر يعمل resolve للـ Pod أو الـ Service التاني بالاسم بدل الـ IP

إزاي Kubernetes بيعمل ده

- Kubernetes بينشر CoreDNS كـ Deployment جوه kube-system
- CoreDNS بياخد Config من ملف اسمه Corefile (موجود في /etc/coredns/corefile)
- الملف ده بيتعمله Mount عن طريق ConfigMap : يعني تقدر تعدل في الإعدادات من غير ما تغير الـ Pod نفسه.
- جوه الـ Corefile فيه Plugins :
  - Errors / Health / Metrics / Cache ... إلخ.
  - أهم Plugin بالنسبة لنا هو : kubernetes اللي بيحدد:
    - الـ Domain الافتراضي (cluster.local)
    - طريقة عمل Records للـ Pods والـ Services

📌 أمر عملي تشوف الـ ConfigMap بتاع CoreDNS : kubectl -n kube-system get configmap coredns -o yaml

```
cat /etc/coredns/Corefile
.:53 {
    errors
    health
    kubernetes cluster.local in-addr.arpa ip6.arpa {
       pods insecure
       upstream
       fallthrough in-addr.arpa ip6.arpa
    }
    prometheus :9153
    proxy . /etc/resolv.conf
    cache 30
    reload
}
```

```
kubectl get configmap -n kube-system
NAME                                   DATA      AGE
coredns                                1         168d
```

DNS Records للـ Pods والـ Services

- **Services :** بيتعمل لها Record تلقائيًا باسم ثابت.
  - مثال: web-service.default.svc.cluster.local
- **Pods :** مش بيتعمل لهم Record بالاسم (بس خلي بالك ان الاوبشن ده Disabled by default)
  - لكن ممكن تفعّل خيار pods في الـ CoreDNS config
  - وقتها الـ Pod بياخد اسم بالشكل:

<pod-ip-replaced-with-dashes>.<namespace>.pod.cluster.local

  - مثال للتوضيح اكتر : IP = 10.244.1.5 هايبقي 10-244-1-5.default.pod.cluster.local

---

إزاي الـ Pods بتعرف توصل لـ CoreDNS

- لما CoreDNS يتعمله Deploy ، بيتعمل له Service اسمه kube-dns
- الـ Pods كلها بتتظبط أوتوماتيك بالـ DNS server ده (عن طريق kubelet)
- جوه كل Pod في /etc/resolv.conf هتلاقي:
  - nameserver = <kube-dns-clusterIP>
  - وكمان search entries :
    - default.svc.cluster.local
    - svc.cluster.local
    - cluster.local
  - وده السبب إنك تقدر تكتب:
    - web-service بس.
    - أو web-service.default
    - أو web-service.default.svc
    - أو الاسم كامل web-service.default.svc.cluster.local

📌 أمر عملي تشوف الـ resolv.conf جوه Pod : kubectl exec -it test -- cat /etc/resolv.conf

---

Service Discovery في Kubernetes

- لو عملت nslookup web-service جوه الـ Pod .. هيرجعلك الاسم الكامل (FQDN)
- لو حاولت تعمل lookup لـ Pod .. يبقي لازم تكتب الاسم بالكامل (يعني مش هيتجاب بالـ Short Name)
- أي أسماء خارج الكلاستر (مثال : www.google.com) فالـ CoreDNS بيـ Forward للـ DNS بتاع الـ Node نفسه.

(Forward to the nameserver specified in the coredns pods which located in /etc/resolv.conf file. Then the /etc/resolv.conf file is set to use the name server from the K8S Node)

📌 أمر عملي:

❖ kubectl exec -it test -- nslookup web-service **or** kubectl exec -it test -- nslookup www.google.com

---

✅ الخلاصة

- Kubernetes بيستخدم CoreDNS كـ DNS Server افتراضي.
- CoreDNS بيتنصب كـ Pods في kube-system وبيتظبط بملف ConfigMap (Corefile)
- Services بياخدوا Records أوتوماتيك (بـ FQDN ثابت)
- Pods مش بياخدوا Records إلا لو فعّلتها.
- /etc/resolv.conf جوه كل Pod بيتظبط أوتوماتيك عن طريق kubelet
- DNS structure : <service-name>.<namespace>.svc.cluster.local
- أي استعلام خارجي بيتبعت للـ DNS Server بتاع الـ Node

Ingress

البداية Services vs Ingress :

❖ الناس ساعات بتتلخبط : "إمتى أستخدم Service وإمتى أستخدم Ingress ؟"

- Service وظيفته الأساسية: يخلي البودز يبانوا جوه الكلاستر أو برّه (ClusterIP, NodePort, LoadBalancer)
- المشكلة : مع نمو التطبيقات والخدمات، Services لوحدها مش كفاية .. طاب ليه ؟!؟!؟!؟!
  ودا لأننا بتحتاج Layer أعلى يعمل URL routing + SSL + إدارة أسهل.
- هنا بييجي دور Ingress

📌 أمر تشوف الـ Services : kubectl get svc

سيناريو عملي : Online Store

- عندك تطبيق my-online-store.com
- التطبيق معمول كـ Pod → Deployment
- معاه MySQL كـ Pod + Service (ClusterIP) علشان يتوصل من الـ app
- التطبيق محتاج يتفتح برّه:
  - فعملت Service نوعه NodePort بياخد port عالي زي 38080
  - يبقى الدخول هايبقي من خلال : http://<NodeIP>:38080
- بعدين تضبط DNS علشان الدومين يروح للـ Node IP وبالتالي تقدر تخش بالصيغة دي :
  - my-online-store.com:38080

📌 لاالحظ ان أمر إنشاء NodePort سريع : kubectl expose deployment my-app --type=NodePort --port=80

⚠️ العيب: المستخدم هايضطر يكتب Port مش عملي في Production

حلول أبسط مع الـ Cloud

- على Google Cloud أو AWS/Azure :
  - بدل NodePort ممكن تعمل Service نوعه LoadBalancer
  - Kubernetes هيطلب من الكلاود Load Balancer حقيقي ويوصله بالـ Service
  - تاخد External IP وتوجه عليه الـ DNS

⚠️ العيب: كل Service = LoadBalancer مستقل = تكلفة $$$ + إدارة معقدة.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-app
spec:
  type: LoadBalancer
  selector:
    app: my-app
  ports:
    - port: 80
      targetPort: 8080
```

المشكلة عند التوسع

- عندك دلوقتي تطبيق تاني (video streaming)
- عايز:
  - my-online-store.com/wear يفتح التطبيق القديم.
  - my-online-store.com/watch يفتح الفيديو.
- لو مشيت بنفس الأسلوب القديم:
  - لازم LoadBalancer لكل Service
  - تظبط Proxy تاني يعمل URL-based routing وبالتالي تدفع أكثر
  - وكمان SSL Certificates لازم تتوزع على كل Service/LoadBalancer

🎯 الحل هو : Ingress

- Ingress = Layer 7 LoadBalancer جوه Kubernetes
- **بيعمل حاجتين رئيسيين:**
  1. URL/Host-based routing : يوجه الطلب حسب الـ Domain أو الـ Path
  2. SSL Termination : شهادة واحدة تتظبط في مكان واحد.
- **بيتكون من:**
  - Ingress Controller (NGINX, Traefik, HAProxy, Istio ...)
  - Ingress Resources (ودي عبارة عن YAML Rules بتحدد مين يستقبل إيه)

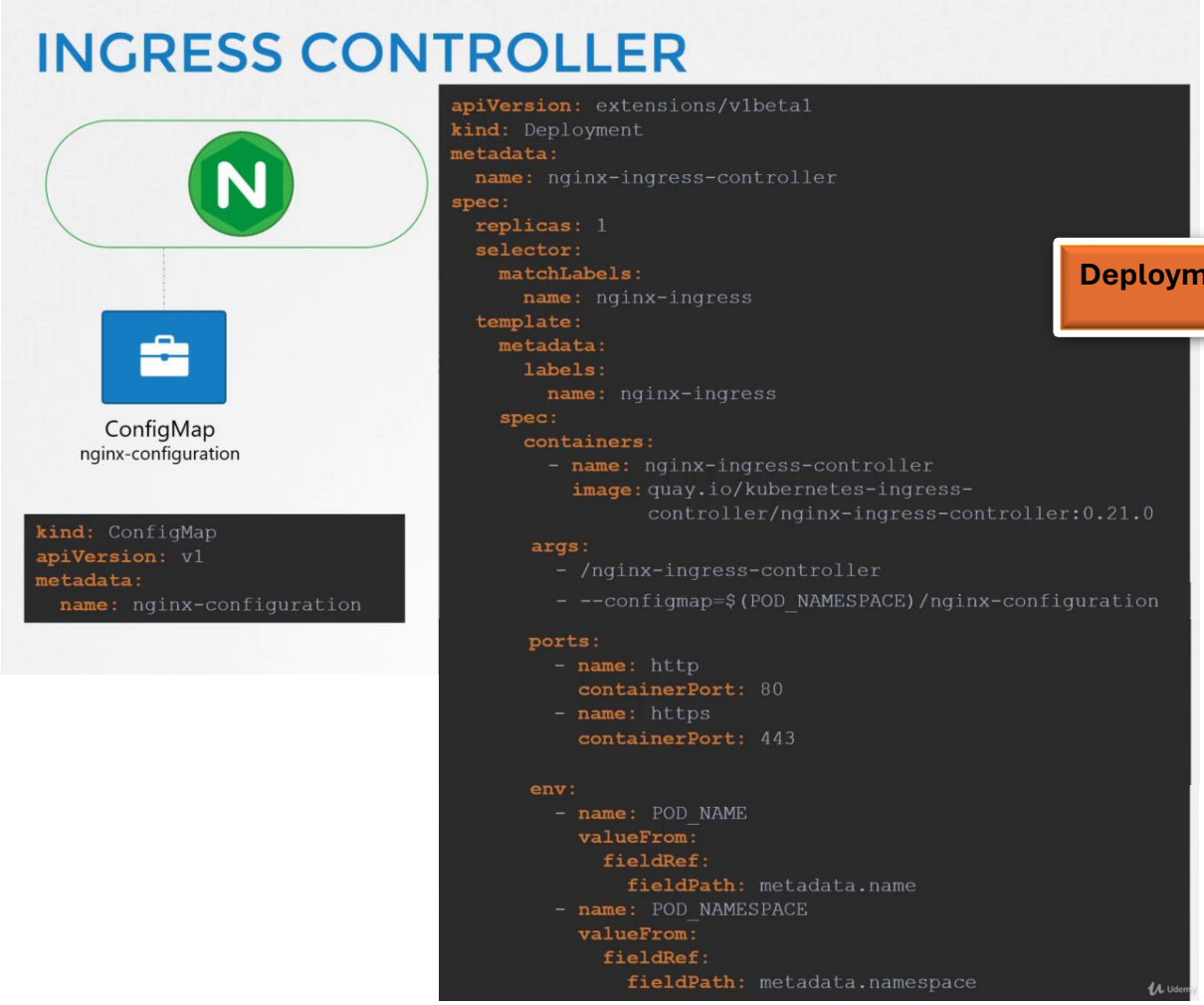
📌 ملاحظة : الكلاستر مش بييجي مع Ingress Controller جاهز ولازم تنزّله. لان هو المسئول عن تنفيذ الـ Rules دي

إزاي بننشر Ingress Controller (مثال NGINX)

- Deployment لـ NGINX Ingress Controller
- ConfigMap علشان إعدادات NGINX (حتى لو فاضية) .. (ولاحظ ان المقصود بإعدادات NGINX هي .. (Path to store Logs, keep-alive threshold, SSL Setting, Session Timeout ....etc)
- Service من نوع NodePort/LoadBalancer علشان تفتح الـ Controller برّه.
- ServiceAccount + RBAC Permissions علشان يقدر يقرأ الـ Ingress Resources ويظبط نفسه.

📌 مثال سريع Deployment :

❖ kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/main/deploy/static/provider/cloud/deploy.yaml



```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-ingress
spec:
  type: NodePort
  ports:
  - port: 80
    targetPort: 80
    protocol: TCP
    name: http
  - port: 443
    targetPort: 443
    protocol: TCP
    name: https
  selector:
    name: nginx-ingress
```

Then the service we need to expose the Ingress controller to the external world

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: nginx-ingress-serviceaccount
```

Roles | ClusterRoles | RoleBindings

To monitor the k8s cluster for ingress resources and configure the underlying nginx server when something is changed .. The Ingress Controller needs a Service Account with the correct roles and Role Binding

---

إزاي بننشئ Ingress Resource

- تعريف بالـ YAML
- تحدد الدومين والـ paths
- كل Path يروح لـ Service مختلف.

📌 مثال:

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-store-ingress
spec:
  rules:
  - host: my-online-store.com
    http:
      paths:
      - path: /wear
        pathType: Prefix
        backend:
          service:
            name: wear-service
            port:
              number: 80
      - path: /watch
        pathType: Prefix
        backend:
          service:
            name: video-service
            port:
              number: 80
```

To Split Traffic by Domain Name

📌 أوامر عملية:

- kubectl apply -f ingress.yaml
- kubectl get ingress
- kubectl describe ingress my-store-ingress

أي Traffic على /wear : يروح لـ wear-service

أي Traffic على /watch : يروح لـ video-service

أي حاجة مش متغطية : تروح للـ default backend (زي صفحة 404)

SSL مع Ingress

- تعمل TLS Secret بالشهادة:
  - ❖ kubectl create secret tls my-tls-secret --key my-tls.key --cert my-tls.crt
- تضيفه للـ Ingress :

```
spec:
  tls:
  - hosts:
    - my-online-store.com
    secretName: my-tls-secret
```

✅ الخلاصة

- Service بيعرض التطبيق (ClusterIP / NodePort / LoadBalancer)
- Ingress بيزود:
  - URL Routing (Paths, Hosts)
  - SSL في مكان واحد.
  - Layer 7 Load Balancing
- محتاج :
  - Ingress Controller (مثلًا NGINX)
  - Ingress Resources (YAML rules).
- ده بيخلي إدارة التطبيقات الكبيرة (بـ Subdomains و Paths مختلفة) أسهل وأرخص.

🔑 نقاط مهمة إضافية عن Ingress

Ingress vs Ingress Controller

- Ingress Resource : مجرد rules مكتوبة بالـ YAML ومش بتعمل أي حاجة لوحدها .(مجرد بتحدد مين يستقبل إيه)
- Ingress Controller : التطبيق اللي بينفذ القواعد دي.(NGINX, Traefik, HAProxy ...)
- يعني لو معندكش Controller ، الـ Ingress مش هيشتغل أصلًا.

Path Types

من اول Kubernetes v1.18 وانت طالع :

- Prefix : أي request يبدأ بالـ path ده.
  - مثال : /app يطابق /app, /app1, /app/test
- Exact : لازم يطابق الـ path بالظبط.
- ImplementationSpecific : حسب الـ Controller (وممكن يبقى زي Prefix)

Default Backend

أي Request مش مطابق لأي Rule في الـ Ingress فبيتبعت للـ Default Backend
يعني في Kubernetes ، الـ Ingress بيشتغل كـ "Gateway" يدخل عليه المستخدمين (users) من الخارج ويوصلهم للخدمات (Services)الداخلية.
لكن المشكلة بتظهر لو الـ path اللي المستخدم بيدخله مش متطابق مع اللي التطبيق الداخلي (backend app) متوقعه.

- يعني مثلا المستخدم هيزور: http://<ingress-service>:<ingress-port>/watch
- وفي نفس الوقت فالتطبيق الداخلي (watch app) متوقع يشتغل على: http://<watch-service>:<port>/
  - ❖ يعني مفيش /watch في URL اللي التطبيق متوقعه
  - ❖ وبالتالي في حالة ان مفيش أي إعدادات إضافية .. فالـ NGINX Ingress هيبعت request زي ما هو،
    وبالتالي يوصل لـ: http://<watch-service>:<port>/watch
    وبالتالي يديك 404 Not Found لإن التطبيق مش متعرف على /watch

---

◆ الحل rewrite-target :

الـ rewrite-target annotation بتقول للـ NGINX :

- "قبل ما تبعت الـ request للتطبيق الداخلي، عدّل الـ path (يعني إعمل search & replace)
- بكده نقدر نشيل أو نبدّل جزء من الـ URL اللي المستخدم دخله، ونخلي التطبيق يستقبل request بالطريقة اللي هو متوقعها.

🛠️ مثال بسيط (pay-service)

❖ إيه اللي بيحصل هنا؟

1. المستخدم يدخل:
   http://<ingress-service>:<ingress-port>/pay
2. الـ Ingress هيمسك الـ path اللي هو /pay
3. بسبب nginx.ingress.kubernetes.io/rewrite-target: /
   ، الـ /pay بيتعمله replace ويتحول لـ /
   replace("/pay", "/")
4. النتيجة النهائية اللي توصل للتطبيق الداخلي:
   http://pay-service:8282/

✨ وده بالظبط اللي التطبيق عايزه.

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: test-ingress
  namespace: critical-space
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - http:
      paths:
      - path: /pay
        backend:
          serviceName: pay-service
          servicePort: 8282
```

◆ مثال متقدم (Regex + Capture groups)

❖ شرح الـ path: /something(/|$)(.*) :

- /something : الجزء الأساسي اللي المستخدم بيكتبه.
- (/|$) : معناها: يا إما فيه / بعد الكلمة، يا إما نهاية الـ string
- (.*) : معناها: أي حاجة تيجي بعد كده (بأي طول).

يعني لو المستخدم كتب:

- /something
- /something/
- /something/abc/xyz

كل ده هيتم مطابقته.

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: rewrite
  namespace: default
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /$2
spec:
  rules:
  - host: rewrite.bar.com
    http:
      paths:
      - path: /something(/|$)(.*)
        backend:
          serviceName: http-svc
          servicePort: 80
```

❖ شرح rewrite: /\$2

- $2 معناها : "المجموعة التانية اللي اتعمل لها match في regex "
- المجموعة التانية هنا هي (.*) .. و أي حاجة بعد /something/....

مثال عملي

- Request : http://rewrite.bar.com/something/abc
- Regex match :
  - /something : الجزء اللي هيتشال.
  - /abc : ده هو (.*) = group 2
- Rewrite :

  /$2 → /abc
- النتيجة اللي توصل للتطبيق: http://http-svc:80/abc

🔑 الخلاصة

- من غير rewrite-target : المستخدم يزور /watch فيوصل للتطبيق /watch فيحصل 404.
- بـ rewrite-target : المستخدم يزور /watch فالـ ingress يعدّل الـ path لـ / وبالتالي يوصل للتطبيق بالطريقة الصح.

❖ استخدامات rewrite-target :

1. إخفاء أو تعديل paths عشان التطبيقات الداخلية تفضل بسيطة.
2. Routing ذكي باستخدام regex (زي /something(/|$)(.*))
3. فصل build-time path عن user-facing path

Annotations (التعليقات التوضيحية) (مهمة جدًا)

- Controllers (خصوصًا NGINX) بتدعم Annotations تضيف Features
- أمثلة :
  - Rewrite URL : nginx.ingress.kubernetes.io/rewrite-target: /
  - Force HTTPS Redirect : nginx.ingress.kubernetes.io/ssl-redirect: "true"
  - Rate Limiting, Load Balancing Algorithm, Whitelisting ....

Multiple Ingress Objects

- ممكن تعمل أكتر من Ingress Resource لنفس الـ Host
- الـ Controller بيجمع القواعد دي مع بعض.

```yaml
rules:
- host: shop.my-store.com
  http:
    paths:
    - path: /
      pathType: Prefix
      backend:
        service:
          name: shop-service
          port:
            number: 80
- host: video.my-store.com
  http:
    paths:
    - path: /
      pathType: Prefix
      backend:
        service:
          name: video-service
          port:
            number: 80
```

Subdomains

- تقدر تعمل Rules مختلفة حسب الـ Host
  - shop.my-store.com يروح لـ Service مختلف عن video.my-store.com.

SSL/TLS Advanced

- ممكن تعمل TLS لمجموعة Hosts
- ممكن تستخدم cert-manager علشان تجيب شهادات SSL أوتوماتيك من Let's Encrypt

Limitations

- Ingress بيدعم HTTP/HTTPS mainly (Layer 7)
- مش مناسب للبروتوكولات التانية زي TCP, UDP
- لو محتاج TCP/UDP : تستخدم Service type=LoadBalancer أو Ingress TCP/UDP ConfigMap (في NGINX)

Monitoring & Debugging

- أوامر عملية:
  - kubectl get ingress
  - kubectl describe ingress my-ingress
  - kubectl logs -n ingress-nginx <nginx-pod-name>
- ساعات المشكلة بتكون في Annotations أو Certificates

🔎 الخلاصة الإضافية:

- Ingress Resource = Rules
- Ingress Controller : اللي بينفذ الـ Rules دي
- تقدر تتحكم في الـ Routing بالـ path والـ host
- تقدر تدير SSL بشكل مركزي.
- تقدر تزود Features قوية بالـ Annotations
- مش مناسب لـ non-HTTP traffic (TCP/UDP)

Install K8S – MindSet

مقدمة

- السكشن دا عبارة عن Introduction للجزء الخاص بعملية تنصيب – إعداد – اختبار الكلاستر بتاع Kubernetes
- الهدف: نطبق كل المفاهيم اللي اتعلمناها قبل كده ونربطها ببعض في مشروع عملي.
- ده جزء مهم جدًا لأنك في الامتحان (CKA) أو في الشغل هتحتاج تعمل Install من الصفر وتجهز Environment كاملة.

تصميم الـCluster

- أول خطوة: نفكر في Design الكلاستر.
- محتاجين نحدد:
  - عدد الـ Master Nodes عشان الـ Control Plane
  - عدد الـ Worker Nodes
  - هل هنحتاج High Availability (HA) ولا لأ؟
- في الشرح هنا، هنبدأ بحاجة صغيرة تشتغل على Laptop أو Local Machine

📌 Tools مستخدمة:

- VirtualBox : عشان نعمل Virtual Machines
- Vagrant : يسهل إدارة الـ VMs بدل ما تعمل كل VM بإيدك

High Availability & ETCD

- لازم نفهم إزاي الـ HA Cluster بيشتغل.
- في Kubernetes :
  - أكتر من Master Node : يمنع الـ Single Point of Failure
  - الـ ETCD Cluster (Database) بيبقى موزع عشان يحفظ الـ State
- كمان هنتعلم Best Practices لإدارة ونشر الـ HA Cluster بشكل صحيح.

خطوات الـ Installation

- هنتبع عملية End-to-End :
  - نعمل Setup للـ Virtual Machines
  - نجهز Network و Configurations
  - نسطب Kubernetes يدويًا.

📌 هنا هنستخدم نسخة معدّلة من مشروع:

- Kubernetes The Hard Way – by Kelsey Hightower : ده مشهور جدًا لتعليم خطوات الـ Installation من الصفر.
- الميزة: بيديك فهم عميق بدل ما تعتمد على Automation Tools بس.

Bootstrapping Worker Nodes

- بعد ما نجهز الـ Master Nodes ، لازم نضم الـ Worker Nodes للكلاستر.
- عندنا طريقتين:
  1. الطريقة التقليدية : تعمل Configuration يدوي.
  2. TLS Bootstrap : طريقة أوتوماتيك شوية، بتستخدم Certificates عشان تأمن عملية Join للكلاستر.

Validation (اختبار الكلاستر)

- بعد الـ Installation ، لازم نعمل Test عشان نتأكد كل حاجة شغالة.
- بنستخدم Kubernetes Test Infrastructure : أدوات رسمية بتجرب Deployments و Services وتتأكد إن الكلاستر بيشتغل بشكل سليم.

📌 مثال عملي بعد الـ Install :

- ❖ kubectl get nodes
- ❖ kubectl run test-pod --image=nginx
- ❖ kubectl get pods -o wide

- كده تتأكد إن الـ Worker Nodes Join مظبوط.

✅ الخلاصة

- في الجزء ده :
  - هنتعلم إزاي نصمّم ونبني Cluster من الصفر.
  - نفهم High Availability & ETCD كويس.
  - نطبّق Install يدوي باستخدام VirtualBox + Vagrant
  - نجرّب طريقتين لضم Worker Nodes
  - نعمل Validation على الكلاستر باستخدام أدوات رسمية.

mmumshad/kubernetes-the-hard-way: Bootstrap Kubernetes the hard way on Vagrant on Local Machine. No scripts.

Design a Kubernetes Cluster

أسئلة قبل التصميم

قبل ما تبدأ تبني أي Cluster لازم تجاوب على شوية أسئلة :

- الغرض من الكلاستر ؟ (Testing – Production)
- إنتو كشركة تبع Adopt إزاي للـ Cloud ؟ Managed ولا Self-Hosted ؟
- نوع الـ Workloads ؟ Web Apps ؟ Big Data ؟ Analytics ؟
- عدد التطبيقات ؟ قليل ولا كتير؟
- نوع الترافيك ؟ مستمر وثقيل ولا Burst (بيزيد فجأة وبعدين يقل) ؟

👉 الفكرة: إجابتك على الأسئلة دي هي اللي بتحدد حجم الكلاستر، عدد النودز، وأدوات الـ Deployment

---

خيارات حسب الغرض

🔹 للتعلم (Learning) :

- كفاية Minikube أو Single Node Cluster باستخدام kubeadm
- ينفع تشتغل Local أو Cloud (AWS, GCP)

📌 أمر سريع لإنشاء Cluster بـ Minikube : minikube start --driver=docker

---

🔹 للتطوير و الـ Testing

- Multi-node Cluster (Master + Workers)
- مناسب نستخدم kubeadm
- أو Managed Services (GKE, EKS, AKS) عشان توفر وقت.

📌 أوامر باستخدام kubeadm :

❖ kubeadm init

❖ kubeadm join <master-ip>:6443 --token <token> --discovery-token-ca-cert-hash sha256:<hash>

🔹 للـ Production

- محتاج HA (High Availability)
- Multiple Masters + Multiple Workers
- Tools :
  - kubeadm (Self-hosted)
  - GCP GKE (Managed).
  - AWS EKS (Self-hosted) على AWS)
  - Azure AKS (Managed)

📌 في الـ Production : نركز على Fault Tolerance (يتحمل الأعطال) + Scalability

---

أرقام مهمة عن Kubernetes Limits

- لحد 5000 Nodes في الكلاستر.
- 150 ألف Pod
- 300ألف Container
- 100 Pod لكل Node

⚠️ مش محتاج تحفظ الأرقام دي في الامتحان، هتلاقيها في Documentation

اختيار مكان التشغيل (On-Prem vs Cloud)

- On-Prem:
  - kubeadm ممتاز وسهل.
  - تختار بنفسك حجم السيرفرات.
- Cloud:
  - GCP : GKE ، One-click upgrade
  - AWS: EKS
  - Azure: AKS

📌 CSPs (Cloud Providers) بيختاروا لك حجم الـ Nodes أوتوماتيك حسب احتياجاتك.

التخزين (Storage Considerations)

- SSD : High Performance
- Multiple Concurrent Access : Network-based storage
- Shared Access بين (Pods → Persistent Volumes) + Storage Classes

📌 مثال على StorageClass :

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast-ssd
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-ssd
```

اختيار نوع الـ Nodes

- ممكن تبني الكلاستر على:
  - Physical Servers
  - Virtual Machines (مثال VirtualBox)
  - Cloud VMs

📌 لاحظ ان هنا احنا : هنشتغل على 3 Nodes (1 Master + 2 Workers) باستخدام VirtualBox

---

دور الماستر و النودز

- Master Node → Control Plane (API Server, etcd, Scheduler, Controller Manager)
- Worker Nodes (Kubelet, Kupe-Proxy) : يشغلوا الـ Workloads (Pods)

📌 ملاحظة:

- الماستر ممكن يشغل Pods كمان، لكن = Best Practice نخليه Control Plane فقط.
- وافتكرررررر ان kubeadm بيمنع تشغيل Pods على Master باستخدام Taint

أمر لعرض الـ Taints : kubectl describe node <master-node> | grep Taints

---

اعتبارات إضافية

- لازم كل الـ Nodes يشتغلوا على 64-bit Linux
- في الكلاسترات الكبيرة : ممكن تعزل ETCD Cluster في نودز خاصة بيه.
  وده بيدي Reliability أكتر هنتكلم عنه في جزء HA

✅ الخلاصة

- قبل ما تبني Cluster لازم تحدد الغرض (Learning, Dev, Prod)
- الأدوات : Minikube ، kubeadm، GKE، EKS، AKS، kOps.
- في الـ Production : ركّز على HA + Storage + Scalability
- Master مسؤول عن Control Plane ، Workers يشغلوا Workloads
- Best Practice : فصل الماستر عن تشغيل الـ Pods

Choosig Kubernetes Infrastructure

فين تقدر تشغل Kubernetes ؟

- Laptop / Local Machine
- Physical Servers (On-Premises).
- Virtual Machines داخل المؤسسة.
- Public Cloud (AWS, GCP, Azure)

👉 الاختيار بيعتمد على:

- هل الغرض تعلم / تطوير / إنتاج؟
- إيه التطبيقات اللي هتشغلها؟
- إيه الـ Cloud Ecosystem المتاح عندك؟

التشغيل محليًا (Local Setup)

🔹 على Linux :

- ممكن تعمل Manual Installation (يعني تنزل الـ binaries وتبني الكلاستر بنفسك)
  ⚠️ الطريقة دي متعبة ومش عملية للمبتدئين.
- الأسهل = تستخدم أدوات أوتوماتيكية.

🔹 على Windows :

- مفيش Windows Binaries رسمية لـ Kubernetes
- الحل : تشغل Linux VMs باستخدام:
  - VirtualBox
  - VMware Workstation
  - Hyper-V
- حتى لو استخدمت Docker Desktop أو حلول مشابهة : هو تحت الهودز بيشغل Linux Kernel صغير عشان الـ Containers أصلاً Linux-based

---

أدوات تساعدك محليًا

- Minikube :
  - Cluster من Node واحدة.
  - مناسب للتعلم.
  - بيعتمد على Virtualization Software زي VirtualBox

📌 أمر تشغيل سريع : minikube start --driver=virtualbox

- kubeadm :
  - تقدر تبني Single-node أو Multi-node Cluster
  - لكن لازم أنت اللي تجهز الـ VMs الأول.
  - بيديك مرونة أكتر من Minikube

📌 مثال:

❖ kubeadm init

❖ kubeadm join <master-ip>:6443 --token <token> --discovery-token-ca-cert-hash sha256:<hash>

👉 الفرق:

- Minikube & مشابهاته : بيعملوا Provisioning للـ VM بنفسهم.
- kubeadm : محتاج توفر الـ VMs بنفسك.

الإنتاج (Production Use)

في البزنس فيه نوعين رئيسيين من الحلول:

🔹 **Turnkey Solutions :**

- إنت بتجهز الـ VMs بنفسك (سواء كان On-Prem أو Cloud)
- بتستخدم Tool/Script يجهزلك الـ Cluster
- أنت المسؤول عن الـ VMs (تحديث - Patching – Upgrading)

🔸 أمثلة:

- KOPS على AWS
- OpenShift (RedHat – GUI + CI/CD Integration)
- Cloud Foundry Container Runtime باستخدام BOSH
- VMware PKS (للي عندهم بيئة VMware)
- Vagrant Scripts لعمل Clusters على Cloud Providers

🔹 **Hosted / Managed Solutions**

- "Kubernetes as a Service"
- الـ Provider بيجهزلك الـ Cluster + VMs + Maintenance
- أقل مجهود منك، وبالتالي أسرع في الـ Setup

🔸 أمثلة:

- GKE (Google Kubernetes Engine)
- EKS (Amazon Elastic Kubernetes Service)
- AKS (Azure Kubernetes Service)
- OpenShift Online (نسخة Managed من RedHat)

📌 الميزة : توفير وقت وجهد + High Availability جاهز.

📌 العيب : أقل تحكم + مرتبط بالـ Cloud Provider

هانختار طريقة (Learning Purpose) في هذا السكشن :

- بما إن الهدف هنا تعلم، ومش كل الناس معاها Cloud Account ،
- تم اختيار بناء Cluster محلي على VirtualBox

📌 Design المستخدم :

- 3 Nodes (1 Master + 2 Workers)
- كلهم VMs شغالة على VirtualBox

✅ الخلاصة

- Kubernetes يشتغل على Laptop, On-Prem, Cloud
- محليًا: استخدم Minikube أو kubeadm
- للإنتاج : عندك اختيارين:
  - Turnkey Solutions : مسئوليتك كاملة لكن عندك Tools تساعد.
  - Hosted Solutions : Provider بيعمل كل حاجة عنك.

🚀 استخدامنا هنا والتطبيق العملي بتاعنا : **Local Setup** على **VirtualBox (3 Nodes)**

Configure High Availability & Networking

ليه محتاج HA أصلاً ؟

- لو عندك Master واحد:
  - الـ Workers والـ Pods هيكملوا يشتغلوا بشكل طبيعي لحد ما يحصل Failures
  - المشكلة تبدأ لما Pod يقع : محتاج Master (ReplicationController / Scheduler) يعيد تشغيله.
  - فلو الـ Master مش شغال : مفيش Controller ولا Scheduler وبالتالي الـ Pod مش هيتعاد تشغيله.
  - كمان، مش هتقدر تدير الكلاستر من بره من خلال (kubectl / API)

👉 الحل : Multiple Masters + Redundancy

مكونات لازم يتعمل لها HA

- Master Nodes (أكتر من واحد)
- Control Plane Components :
  - kube-apiserver
  - kube-scheduler
  - kube-controller-manager
  - ETCD
- Workers & Applications (وده عندنا فعلًا عن طريق ReplicaSets و Services)

إزاي الـ Master يتكرر؟

1. 🔹 kube-apiserver :

- بيشتغل Active-Active : أي API Server ممكن يرد على أي Request
- نقدر نشغل أكتر من API Server في نفس الوقت.
- المشكلة : kubectl بيروح عادة لـ Master محدد على Port 6443
- الحل : ارززع Load Balancer قدام الـ Masters زي (NGINX, HAProxy ...)
- وبالتالي هتلاقي ان kubectl بيتوصل مع الـ Load Balancer بدل ماسترات معينة.

📌 أمر إعداد kubeconfig عشان يوجّه للـ LB :

❖ `kubectl config set-cluster my-cluster --server=https://<LB-IP>:6443`

2. 🔹 kube-scheduler & kube-controller-manager :

- دول ما ينفعش يشتغلوا Active-Active .. لأنهم ممكن يعملوا Duplicate Actions
  - مثال : ممكن Controller Managers 2 يشوفوا Pod Fail فالاتنين يخلقوا Pods زيادة.
- الحل : بيشتغلوا Active-Passive (Standby)
- طاب اللي بيحدد مين ؟؟ Active = Leader Election

📌 إعداد Leader Election :

- بيحصل أوتوماتيك عن طريق الـ Option : --leader-elect=true
- بيتنافسوا على Lock في Object اسمه : kube-controller-manager endpoint
- اللي ياخد الـ Lock يبقى Leader
- Lease Time = 15 ثانية (ممكن يتغير بـ --leader-elect-lease-duration)
- Renewal كل 10 ثواني (--leader-elect-renew-deadline)
- محاولات جديدة كل 2 ثانية (--leader-elect-retry-period)

👉 لو الـ Leader وقع : فالـ Standby يبقى Active خلال ثواني.

3. ETCD (الـ Database بتاع Kubernetes) :

❖ هنا عندنا 2 Topologies :

1) Stacked (داخل الماستر)
   - etcd + Control Plane على نفس السيرفر.
   - Setup أسهل + أقل عدد Nodes
   - العيب : لو Master وقع : فأنا كدا فقدت etcd Member + Control Plane معاه.

2) External ETCD Cluster
   - etcd منفصل على سيرفرات مخصصة.
   - أكتر أمان و Resiliency
   - العيب : أعقد في Setup + محتاج Nodes زيادة .

```
cat /etc/systemd/system/kube-apiserver.service
[Service]
ExecStart=/usr/local/bin/kube-apiserver \\
  --advertise-address=${INTERNAL_IP} \\
  --allow-privileged=true \\
  --apiserver-count=3 \\
  --etcd-cafile=/var/lib/kubernetes/ca.pem \\
  --etcd-certfile=/var/lib/kubernetes/kubernetes.pem \\
  --etcd-keyfile=/var/lib/kubernetes/kubernetes-key.pem \\
  --etcd-servers=https://10.240.0.10:2379,https://10.240.0.11:2379
```

📌 ملاحظة:

- الـ kube-apiserver بيتواصل مع ETCD مباشرة.
- بنحدد في Config : --etcd-servers=https://etcd1:2379,https://etcd2:2379
  بحيث نتأكد ان الـ API Server بيشاور علي الـ Right Address of ETCD Server
- أي ETCD Instance يقدر يستقبل Requests (Read/Write) لأنه Distributed System

---

التصميم النهائي اللي المفروض نوصله بعد تطبيق كل الكلام السابق

- بدأنا بـ :
  - Master + 2 Workers = 3 Nodes
- بعد تطبيق الـ HA :
  - 2 Masters
  - 2 Workers
  - 1 Load Balancer قدام الـ Masters
- المجموع = 5 Nodes

> Depending on your environment & network ecosystem you have a wide variety of networking options to choose from. A list of supported network solutions and their implementation details are available here:
> https://kubernetes.io/docs/concepts/cluster-administration/networking/#how-to-implement the-kubernetes-networking-model

---

✅ الخلاصة

- HA : تمنع نقطة فشل واحدة.
- API Server → Active-Active : مع Load Balancer
- Scheduler + Controller Manager → Active-Passive (Leader Election)
- etcd : ممكن Stacked أو External Cluster (التاني أفضل للـ Production)
- التصميم النهائي : Multiple Masters + LB + Redundancy في كل Component

ETCD In High Availability

مقدمة

- في المحاضرة دي هنتكلم عن ETCD في وضع الـ HA
- وده جزء أساسي قبل ما نتكلم عن تشغيل Kubernetes نفسه في وضع الـ HA
- بمعنى : قبل ما تظبط الـ Control Plane في HA .. فلازم تفهم ETCD HA

📌 أوامر عملية:

تشوف الـ etcd الموجود في الـ kube-system namespace : kubectl get pods -n kube-system -l component=etcd

---

ما هو ETCD ؟

- هو عبارة عن Key-Value Store موزع (Distributed KV Store)
- بيخزن بيانات الـ Cluster زي الـ (Configs, State, Secrets)
- سريع، آمن، موثوق.
- البيانات بتتخزن كـ Key=Value بدل جداول Relational
- بيدعم تخزين البيانات بصيغ زي JSON أو YAML

📌 تقدر تشوف نسخة etcdctl : etcdctl version

ليه الـ ETCD لازم يكون Distributed ؟

- تخيل ان عندك ETCD على سيرفر واحد بس .. فلو السيرفر ده وقع يبقي راحت كل بيانات الكلاستر.
- الحل : نشغل ETCD على 3 سيرفرات أو أكتر بحيث كل Node عندها نسخة من البيانات.
- كده حتى لو Node واحدة وقعت .. الكلاستر يفضل شغال.

📌 Visualization :

- Node1 (ETCD)
- Node2 (ETCD)
- Node3 (ETCD)

كلهم عندهم نفس الداتا

مشكلة الـ Writes

- القراءة سهلة : أي Node ممكن تديك نفس النتيجة.
- المشكلة : لو جالك 2 Write Requests في نفس اللحظة على Nodes مختلفة .. ياتري إيه اللي يتكتب؟

🔑 الحل:

- مش كل Nodes بتكتب.
- فيه Node واحدة بتبقى Leader
- الباقي بيبقوا Followers
- الـ Writes بتتعمل عند الـ Leader .. وبعدها يوزع التحديث للباقي.

📌 أمر عملي :

تشوف مين الـ Leader : ETCDCTL_API=3 etcdctl --endpoints=<ip>:2379 endpoint status --write-out=table

---

RAFT Protocol (خوارزمية الإجماع)

- الـ ETCD بيستخدم بروتوكول اسمه RAFT عشان يحقق الـ Consensus بين الـ Nodes ..
  و يضمن إن كل Cluster شايف نفس الداتا بشكل متسق (Consistent).
- إزاي؟
  1. كل Node بتبدأ كـ Follower وعندها Timer عشوائي.
  2. أول Node يخلص التايمر .. تقدر يرشح نفسها علشان تبقى Leader
  3. باقي الـ Nodes يصوتوا لها.
  4. اللي ياخد أغلبية الأصوات يبقى Leader
  5. لو Leader وقع .. باقي الـ Nodes يعيدوا الانتخاب.

📌 دور الـ Leader :

- كل Write Request يروح للـ Leader
- الـ Leader يكتب الـ Log Entry و يبعتها للـ Followers
- لازم الأغلبية (Quorum) من الـ Nodes تعمل Acknowledge إنها استقبلت.
- بعد كده الـ Leader يعمل Commit ويعتبر الـ Write ناجح.

Quorum (الأغلبية المطلوبة)

- في أي نظام Distributed (زي etcd, Consul, أو أي Database موزعة)، البيانات بتتخزن على أكتر من Node.
- لما تيجي تعمل Write (تكتب Data جديدة) .. النظام محتاج يتأكد إن الكتابة دي وصلت لأغلبية الـ Nodes، مش مجرد Node واحدة ... الأغلبية دي اسمها Quorum.
- الـ Quorum هي اقل عدد من الـ Nodes لاااازم يكون موجود علشان الـ Cluster يشتغل مظبوط

❖ طريقة حساب الـ Quorum :

- **Quorum = (عدد الـ Nodes ÷ 2) + 1**
- يعني لازم يوافق أكتر من نص الـ Nodes.
- الهدف: تمنع التناقض (Split Brain)، بحيث مايبقاش عندك نسختين مختلفتين من نفس البيانات
- مثال:
  - لو عندك Cluster فيه 3 Nodes .. فبتطبيق المعادلة السابقة هايطلع ان الـ Quorum يساوي 2
    يعني لازم على الأقل 2 Nodes يوافقوا على الـ Write .. ولو Node واحدة بس وافقت .. فالكتابة غير صالحة.

📌 علشان تشوف صحة الكلاستر :

❖ ETCDCTL_API=3 etcdctl --endpoints=<ip1>:2379,<ip2>:2379,<ip3>:2379 endpoint health

---

عدد الـ Nodes المناسب

- سواء كان عدد الـ Nodes واحد او اتنين فكده مالهمش قيمة .. ده مش Cluster أصلًا .
  ولو Node وقعت .. يبقي مفيش Quorum .. اذا السيستم كله يقع
- أقل حاجة 3 Nodes :
- الأفضل : عدد فردي (3, 5, 7)
- طيب ليه مش Even ؟
  - لو عندك 6 ← ممكن يتقسموا 3 و3 يبقي مفيش Quorum
  - لو عندك 7 ← هيتقسموا 4 و3 يبقي مهما حصل وطبقا للمعادلة فالـ Quorum موجود.

🔑 الخلاصة:

- 3 كويس كبداية.
- 5 لو عايز Fault Tolerance أعلى .
- أكتر من 5 ← مش ضروري وغالي .

Instances	Quorum	Fault Tolerance
1	1	0
2	2	0
3	2	1
4	3	1
5	3	2
6	4	2
7	4	3

تركيب ETCD عمليًا

خطوات تشغيله على سيرفر:

1. نزّل البايناري :

```
wget -q --https-only --timestamping \
  "https://github.com/etcd-io/etcd/releases/download/v3.5.0/etcd-v3.5.0-linux-amd64.tar.gz"
```

2. فك الضغط وانقل الملفات . `tar -xvf etcd-v3.3.9-linux-amd64.tar.gz`
3. حضّر Certificates (TLS)
4. شغّل الـ Service مع تحديد الـ Peers في `--initial-cluster`

```
wget -q --https-only \
    "https://github.com/coreos/etcd/releases/download/v3.3.9/etcd-v3.3.9-linux-amd64.tar.gz"

tar -xvf etcd-v3.3.9-linux-amd64.tar.gz

mv etcd-v3.3.9-linux-amd64/etcd* /usr/local/bin/

mkdir -p /etc/etcd /var/lib/etcd

cp ca.pem kubernetes-key.pem kubernetes.pem /etc/etcd/
```

أوامر etcdctl

1. تفعيل API v3 : `export ETCDCTL_API=3`
2. إضافة بيانات : `etcdctl put name “john”`
3. قراءة بيانات : `etcdctl get name`
4. تشوف كل الـ Keys : `etcdctl get "" --keys-only --prefix`

✅ الخلاصة

- ETCD : قلب الكلاستر، بيسجّل كل الـ Data
- عشان يكون Highly Available : لازم أكتر من Node
- الـ Leader بيكتب، والـ Followers بياخدوا نسخ.
- RAFT Protocol مسؤول عن الانتخابات.
- Quorum (أغلبية) = الـ (N/2 + 1) Nodes
- الأفضل دايمًا عدد فردي (3 أو 5).
- أوامر etcdctl بتديك إدارة وتست للداتا.

📌 اختبار عملي سريع:

#احفظ قيمة `etcdctl put cluster "k8s-ha”`

#استرجعها `etcdctl get cluster`

Demo - Provision VMs

https://github.com/mmumshad/kubernetes-the-hard-way/blob/master/VirtualBox/docs/02-compute-resources.md

الفكرة العامة

- عشان نعمل Kubernetes Cluster (بـ HA Mode) محتاجين نجهز Infrastructure
- وبالتالي هانستخدم VirtualBox VMs + Vagrant
- السبب: بيخلي الكل عنده نفس البيئة بنفس الإعدادات بضغطة واحدة.

عدد الـ VMs

- في المثال ده : 5 VMs :
  - 2 Masters
  - 2 Workers
  - 1 Load Balancer

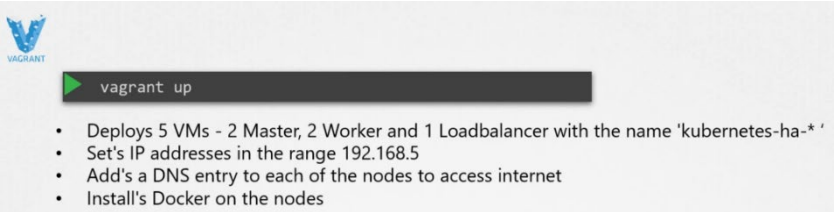
🖥️ Specs :

- كل VM : 1 vCPU + 1GB RAM
- ممكن تقلل RAM لـ 500 MB للـ Workers و Load Balancer

Vagrant vs Manual

- ممكن تعمل الـ VMs واحدة واحدة في VirtualBox
- لكن الأسهل : Vagrant → File واحد (Vagrantfile) بـ Deploy الكل.
- مجرد أمر واحد : vagrant up
  يخلق كل الـ VMs + يجهز Docker + IPs

اللي بيعمله الـ Vagrantfile



- بيعمل 5 VMs على VirtualBox
- يديهم Static IPs في (192.168.5.x) subnet
- يضيف DNS entries عشان يخرجوا على الإنترنت.
- ينزّل Docker 18.06 على الـ Workers
- يفعّل IP Bridge Forwarding

📌 ملاحظة: ده بياخد ~10 دقائق على حسب سرعة جهازك والنت.

الدخول للـ VMs (SSH)

❖ بعد ما تخلص:

- هيتخلق فولدر اسمه .vagrant (مخفي) ... جواه ملفات private_keys لكل VM

❖ عندك طريقتين للـ SSH :

1. باستخدام Vagrant : vagrant ssh worker-1 and vagrant ssh master-1
   ودا اسهل لأنه بيستخدم الـ private_key تلقائيًا
2. باستخدام أي SSH Tool (Putty, MobaXterm ...)
   - الـ User : Vagrant
   - الـ Private Key : موجود vagrant/machines/<vm-name>/virtualbox/private_key .
   - الـ IP : زي 192.168.5.11 للـ Master1

تنظيم الـ Sessions

- هاننظم الـ SSH Sessions في فولدر (زي Projects)
- عمل Session لكل Node باسم واضح (مثال master-1, worker-2)
- الهدف: يبقى سهل تدخل لأي VM بسرعة.

الخلاصة

- جهّزنا Infrastructure فيها:
  - 2 Masters + 2 Workers + 1 LoadBalancer
  - بـ VirtualBox + Vagrant
  - مع Docker already installed
- دلوقتي بقى عندنا أساس نبني عليه الـ Kubernetes Cluster

📌 أوامر مهمة:

تشغيل كل الـ VMs : vagrant up

الدخول لواحدة : vagrant ssh master1

إيقافهم : vagrant halt

مسحهم (لو عايز تبدأ من جديد) : vagrant destroy -f

Demo - Install Client Tools

kubernetes-the-hard-way/docs/03-client-tools.md at master · mmumshad/kubernetes-the-hard-way

اختيار الـ Administrative Client

- بعد ما جهّزنا الـ Infrastructure (الـ VMs بتاعة الكلستر)
- لازم نحدد Client واحد مسؤول عن الإدارة.
- ده اللي هيعمل:
  - Certificates
  - kubeconfig files
  - توزيع الملفات دي على باقي النودز.

📌 ممكن تختار جهازك (اللابتوب) أو أي نود.

واحنا هنا هاانختار master-1 يكون الـ Admin Client

شرط أساسي

- الـ Client اللي تختاره لازم يقدر يعمل SSH Access لكل النودز التانية.
- الهدف : تقدر تنسخ الملفات عليهم بسهولة.

تفعيل Passwordless SSH

- الحل : نعمل SSH Key-based Authentication من master-1 لكل النودز.

الخطوات:

1. على master-1 : ssh-keygen
   - دوس Enter على كل الخيارات وخليها Default
   - وبعدين ls .ssh هيطلعلك ملفات تحت:
     - ~/.ssh/id_rsa (Private key)
     - ~/.ssh/id_rsa.pub (Public key)
2. انسخ الـ Public Key لمكان authorized_keys على كل Node تانية :

- cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
  - أو بطريقة أسرع: ssh-copy-id vagrant@<node-ip>

تجربة الاتصال

- من master-1 جرّب تدخل على master-2 :: ssh vagrant@192.168.5.12
- لو دخل من غير ما يطلب Password : يبقى شغال تمام.
- كررها مع كل الـ Workers والـ LoadBalancer

تثبيت kubectl

- دلوقتي لازم يكون عندك kubectl على الـ Admin Client (سواء master-1 أو اللابتوب)
- ده هتستخدمه عشان:
  - تولّد الـ kubeconfig files
  - تتحكم في الكلستر.

أوامر (مثال Ubuntu) :

❖ curl -LO "https://dl.k8s.io/release/$(curl -s https: //dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"

❖ chmod +x kubectl

❖ sudo mv kubectl /usr/local/bin/

🎯 الخلاصة

- حددت جهاز مسؤول عن الإدارة (master-1)
- فعلت SSH key-based access عشان تنسخ الملفات بسهولة.
- ثبّتت kubectl على الجهاز ده.
- دلوقتي بقيت جاهز تكمل خطوات إعداد الكلستر (certificates + kubeconfigs)

Demo - Secure Cluster

kubernetes-the-hard-way/docs/04-certificate-authority.md at master · mmumshad/kubernetes-the-hard-way

المقدمة

الفيديو بيشرح إزاي نأمّن الاتصال داخل الكلاستر باستخدام TLS Certificates
الفكرة الأساسية إن كل مكوّن من مكونات الـ Control Plane والـ Worker Nodes يتواصل مع بعض بطريقة مشفرة وآمنة.

التحضير

- هتشتغل من جهاز إداري (Admin Client) اللي هو في حالتنا دي الـ master-1 Node
- لازم الجهاز ده يقدر يعمل SSH على باقي الأجهزة علشان ننسخ الملفات (الشهادات) بينهم.
- على الشمال في التيرمنال وعلى اليمين Documentation (اللينك بتاعه ف اول توبيك) علشان يسهّل نسخ الكوماندات.

إنشاء Certificate Authority (CA)

أول خطوة : نعمل Root CA اللي هي المرجع اللي بيوقّع باقي الشهادات.

خطوات:

1. إنشاء Private Key للـ CA : openssl genrsa -out ca.key 2048
2. إنشاء CSR (طلب شهادة) باسم kubernetes-CA
   - openssl req -x509 -new -key ca.key -subj "/CN=kubernetes-CA" -out ca.crt
3. توقيع الشهادة بنفسك (Self-sign)

```
# Create private key for CA
openssl genrsa -out ca.key 2048

# Create CSR using the private key
openssl req -new -key ca.key -subj "/CN=KUBERNETES-CA" -out ca.csr

# Self sign the csr using its own private key
openssl x509 -req -in ca.csr -signkey ca.key -CAcreateserial  -out ca.crt -days 1000
```

👉 الناتج عندك:

- ca.crt (الشهادة)
- ca.key (المفتاح الخاص)
- ملف CSR مالوش لازمة بعد التوقيع .

إنشاء شهادة Admin

The Admin Client Certificate

Generate the `admin` client certificate and private key:

```
# Geenrate private key for admin user
openssl genrsa -out admin.key 2048

# Generate CSR for admin user. Note the OU.
openssl req -new -key admin.key -subj "/CN=admin/O=system:masters" -out admin.csr

# Sign certificate for admin user using CA servers private key
openssl x509 -req -in admin.csr -CA ca.crt -CAkey ca.key -CAcreateserial  -out admin.crt
```

- نولّد Private Key جديد.
- نعمل CSR مع تحديد:
  - CN=admin
  - O=system:masters (جروب مسؤولين Kubernetes)
- نوقّع الشهادة باستخدام CA

👉 النتيجة:

- admin.key
- admin.crt

باقي مكونات الـ Control Plane

نكرر نفس الخطوات السابقة لكل مكوّن (Controller Manager – Scheduler – Kube-proxy) (و لاحظ اننا لسا في الـ master-1 Node) :

شهادة الـAPI Server

الـ Kube-API Server محتاج شهادة خاصة، ليه؟؟!!
لأنه بيتواصل بعدة أسماء (DNS names, IPs)
خطوات:

1. نعمل ملف config ونكتب فيه كل الأسماء اللي ممكن الـ API Server يتعرف بيها.
   زي kubernetes, kubernetes.default, IPs بتاعة الـ cluster
2. نولّد CSR باستخدام الملف ده.
3. نوقّعه باستخدام CA

kubernetes-the-hard-way/docs/04-certificate-authority.md at master · mmumshad/kubernetes-the-hard-way

---

شهادة ETCD

- برضه نعمل ملف config ونحدد اسم الشهادة etcd-server
- نولّد CSR
- نوقّعه بالـ CA

kubernetes-the-hard-way/docs/04-certificate-authority.md at master · mmumshad/kubernetes-the-hard-way

---

Service Accounts

- نولّد key pair مخصوص علشان الـ Service Account Tokens
- الـ Controller Manager بيستخدمه علشان يولّد التوكنز.

kubernetes-the-hard-way/docs/04-certificate-authority.md at master · mmumshad/kubernetes-the-hard-way

---

Certificates للـ Worker Nodes (Kubelets)

- لسه ماعملناش شهادات الـ kubelets بتوع الـ Worker Nodes
- هنأجلها لوقت ما نيجي نكوّن الـ Worker Nodes

توزيع الشهادات

- ناخد كل الشهادات اللي عملناها ونوزعها على باقي الـ Master Nodes (في حالة لو عندنا أكتر من واحد)
- لو شغال على Master واحد بس، يبقى التوزيع مش ضروري.

النتيجة النهائية

- كده كل شهادات الـ Control Plane جاهزة.
- الشهادات بتاعة الـ Worker Nodes هتتعمل لاحقًا.
- دلوقتي نقدر نبدأ تنصيب وتهيئة الـ Master Nodes

✅ الخلاصة:

إحنا أنشأنا CA أساسية، ووقّعنا منها شهادات لكل المكونات زي ..
(Admin, Controllers, Proxy, Scheduler, API Server, ETCD, Service Accounts) ..
ثم وزّعناها على الماستر نودز .. باقي بس Certificates الخاصة بالـ Worker Nodes اللي هنعملها في مرحلة تانية.

Demo - Kube Config

kubernetes-the-hard-way/docs/05-kubernetes-configuration-files.md at master · mmumshad/kubernetes-the-hard-way

kubeconfig Files

- وظيفتها:

أي Client (زي kubectl أو مكونات الـ Control Plane) بيحتاج يتواصل مع kube-apiserver وبالتالي لازم يكون عنده ملف kubeconfig

الملف ده بيحتوي على:

  - عنوان الـ API Server
  - الـ Credentials أو Tokens
  - الـ Context اللي بيربط User بـ Cluster بـ Namespace

من هم الـ Clients اللي بيحتاجوا kubeconfig ؟

1. Admin User : اللي هيستخدم kubectl لإدارة الـ Cluster
2. kube-controller-manager : علشان يتكلم مع الـ API Server
3. kube-scheduler : برضه بيتواصل مع الـ API Server
4. kube-proxy : اللي بيشتغل على الـ Worker Nodes

كل دول لازم نولّد لهم kubeconfig files

الوصول للـ API Server

في التصميم اللي عندنا فيه Load Balancer قدام الـ Master Nodes :

- المكونات اللي على نفس الـ Master Node (زي Controller-Manager & Scheduler)
  بيتواصلوا مع الـ API Server عن طريق Loopback (127.0.0.1)
- الـ Clients الخارجية (زي Admin User أو kube-proxy على الـ Workers) :
  بيتواصلوا مع الـ API Server من خلال Load Balancer

خطوات إنشاء kubeconfig Files

1 حدد عنوان الـ Load Balancer :

مثلاً : LOADBALANCER_ADDRESS=192.168.5.30

2 إنشاء kubeconfig لـ kube-proxy :

```
kubectl config set-cluster kubernetes-the-hard-way \
  --certificate-authority=ca.pem \
  --embed-certs=true \
  --server=https://${LOADBALANCER_ADDRESS}:6443 \
  --kubeconfig=kube-proxy.kubeconfig

kubectl config set-credentials kube-proxy \
  --client-certificate=kube-proxy.pem \
  --client-key=kube-proxy-key.pem \
  --embed-certs=true \
  --kubeconfig=kube-proxy.kubeconfig

kubectl config set-context default \
  --cluster=kubernetes-the-hard-way \
  --user=kube-proxy \
  --kubeconfig=kube-proxy.kubeconfig

kubectl config use-context default --kubeconfig=kube-proxy.kubeconfig
```

3 إنشاء kubeconfig لـ Controller-Manager

هنا الـ API Server بيتكلم داخليًا عبر Loopback :

```
kubectl config set-cluster kubernetes-the-hard-way \
  --certificate-authority=ca.pem \
  --embed-certs=true \
  --server=https://127.0.0.1:6443 \
  --kubeconfig=kube-controller-manager.kubeconfig

kubectl config set-credentials kube-controller-manager \
  --client-certificate=kube-controller-manager.pem \
  --client-key=kube-controller-manager-key.pem \
  --embed-certs=true \
  --kubeconfig=kube-controller-manager.kubeconfig

kubectl config set-context default \
  --cluster=kubernetes-the-hard-way \
  --user=kube-controller-manager \
  --kubeconfig=kube-controller-manager.kubeconfig

kubectl config use-context default --kubeconfig=kube-controller-manager.kubeconfig
```

4 نفس الكلام بيتكرر مع Scheduler و Admin User

الفرق بس في الـ certificates والـ user name

توزيع الملفات

- kube-proxy.kubeconfig : يتوزع على كل الـ Worker Nodes
- kube-controller-manager.kubeconfig + kube-scheduler.kubeconfig + admin.kubeconfig : يتوزعوا على الـ Master Nodes

---

ملاحظة

إحنا لسه ماعملناش kubeconfig للـ kubelet اللي موجود على الـ Workers
ده هيتعمل لاحقًا لما نيجي نـ Bootstrap الـ Worker Nodes وفيه طريقتين هنشوفهم مع بعض

Demo – Data Encryption

kubernetes-the-hard-way/docs/06-data-encryption-keys.md at master · mmumshad/kubernetes-the-hard-way

فكرة الـ Data Encryption في Kubernetes

- أي داتا في الـ Cluster (زي Configurations, Secrets, Application data) : بتتخزن في قاعدة البيانات الخاصة بـ etcd
- بشكل افتراضي ، الـ etcd بيخزن البيانات بصيغة نص عادي (Plaintext) ، يعني أي حد عنده وصول مباشر للـ etcd يقدر يشوف الـ Secrets من غير أي تشفير.
- عشان كدا Kubernetes بيوفر ميكانيزم اسمه Encryption at Rest، واللي بيشفر البيانات قبل ما تتخزن جوه etcd

---

إنشاء مفتاح التشفير (Encryption Key)

- أول خطوة : نولّد مفتاح (Key) عشوائي باستخدام أمر بسيط : head -c 32 /dev/urandom | base64

📌 الشرح:

- head -c 32 /dev/urandom : يولّد 32 بايت عشوائي.
- base64 : يحوّل الـ Bytes دي لـ string قابل للاستخدام.

إنشاء ملف إعدادات التشفير (Encryption Config)

- الملف ده بيقول للـ kube-apiserver : " لما تيجي تخزن Secrets في etcd ، استخدم المفتاح الفلاني."
- مثال عملي:

```
cat > encryption-config.yaml <<EOF
kind: EncryptionConfig
apiVersion: v1
resources:
  - resources:
      - secrets
    providers:
      - aescbc:
          keys:
            - name: key1
              secret: ${ENCRYPTION_KEY}
      - identity: {}
EOF
```

📌 الشرح:

- resources: secrets : معناه شفر الـ Secrets بس (ممكن تزود أنواع تانية لو عاوز).
- aescbc : نوع التشفير (AES-CBC)
- fallback option identity: {} : يعني لو التشفير مش متاح هيخزنها عادي من غير تشفير.

استخدام ملف التشفير في الـ kube-apiserver

لما تيجي تشغل الـ kube-apiserver ، لازم تضيف الفلاج ده:

❖ --encryption-provider-config=/etc/kubernetes/encryption-config.yaml

علشان ده اللي بيخلي الـ kube-apiserver يقرأ ملف التشفير ويستخدمه.

توزيع الملف على الـ Master Nodes

- لأن كل Master Node بيشغّل kube-apiserver خاص بيه، فلازم الملف يتنسخ على كل Master Node في نفس المكان.
- مثال : scp encryption-config.yaml user@master-node:/etc/kubernetes/

📌 **إضافات مهمة:**

- الـ Secrets القديمة مش هتتشفّر أوتوماتيك بعد ما تشغّل الميزة. الحل : تعمل re-encryption عن طريق:

❖ **kubectl get secrets --all-namespaces -o json | kubectl replace -f -**

- اختيار نوع التشفير:
  - aescbc : آمن جدًا (موصى به).
  - aesgcm : أسرع لكن أقل استخدام.
  - identity : بدون تشفير (فقط للـ fallback)

Demo – Install ETCD Cluster On Master Nodes

kubernetes-the-hard-way/docs/07-bootstrapping-etcd.md at master · mmumshad/kubernetes-the-hard-way

بداية نشر مكونات الـ Master Plane

- هنا بنبدأ بعملية نشر المكوّنات الأساسية للـ Control Plane
- أول حاجة ETCD Cluster :
ليه؟ لأنه الـ database اللي بيخزن كل بيانات الـ cluster (Nodes, Pods, Configs, Secrets...)

اختيار Topology (Stacked vs External)

- احنا هنا هانشتغل بطريقة الـ stacked topology .. يعني هيشغلوا الـ ETCD cluster على نفس الماستر نودز.
- بس خلي بالك ان في بديل : تعمل External ETCD cluster (عبارة عن Cluster منفصل عن الماستر نودز)
  - ✅ stacked : أسهل لكن لو الماستر وقع → ممكن تخسر الـetcd.
  - ✅ external : أكثر أمان ومرونة لكن محتاج إدارة إضافية.

Deploy الـ ETCD كـ Cluster

- ETCD مش بيشتغل كـ single instance فقط، هنا هيشتغل cluster (مثلاً master-1 و master-2)
- لازم نفس الخطوات تتعمل على كل الـ nodes عشان يبقوا synchronized

---

إدارة الـ SSH لتنفيذ الأوامر

- هانعمل SSH مفتوح على كل الماستر نودز.
- عندنا tool بتسمح إننا نكتب الأمر مرة واحدة ويتنفذ على كل النودز (مثلاً tmux, cssh, أو Ansible)
- أو ممكن تنفذ يدوي node by node

تحميل الـ ETCD Binaries

```
wget -q --https-only --timestamping \
"https://github.com/etcd-io/etcd/releases/download/v3.5.6/etcd-v3.5.6-linux-amd64.tar.gz"
tar -xvf etcd-v3.5.6-linux-amd64.tar.gz
sudo mv etcd-v3.5.6-linux-amd64/etcd* /usr/local/bin/
```

- هنا بننزّل binaries ونفك الضغط وننقلها للـ PATH عشان تبقى متاحة.

تهيئة فولدرات ETCD ونقل الـ Certificates

```
sudo mkdir -p /etc/etcd /var/lib/etcd
sudo cp ca.pem kubernetes-key.pem kubernetes.pem /etc/etcd/
```

- بينقل الـ certs اللي جهزناها قبل كده (CA, server certs, keys)
- دي مهمة عشان الـ etcd communication يكون مشفّر (TLS)

جلب الـ Internal IP لكل VM

```
INTERNAL_IP=$(hostname -I | awk '{print $1}')
```

Then `echo $INTERNAL_IP`

- لازم نحدد internal IP بتاع كل master node
- بيتخزن في variable عشان يستخدمه في config

تحديد اسم الـ Node

- ETCD_NAME=$(hostname -s)
- echo $ETCD_NAME

- اسم النود مهم جداً لأن دا اللي بيميزه داخل الـ cluster
-

كتابة الـ ETCD systemd Service File

```
cat <<EOF | sudo tee /etc/systemd/system/etcd.service
[Unit]
Description=etcd
Documentation=https://github.com/coreos

[Service]
ExecStart=/usr/local/bin/etcd \\
  --name ${ETCD_NAME} \\
  --cert-file=/etc/etcd/etcd-server.crt \\
  --key-file=/etc/etcd/etcd-server.key \\
  --peer-cert-file=/etc/etcd/etcd-server.crt \\
  --peer-key-file=/etc/etcd/etcd-server.key \\
  --trusted-ca-file=/etc/etcd/ca.crt \\
  --peer-trusted-ca-file=/etc/etcd/ca.crt \\
  --peer-client-cert-auth \\
  --client-cert-auth \\
  --initial-advertise-peer-urls https://${PRIMARY_IP}:2380 \\
  --listen-peer-urls https://${PRIMARY_IP}:2380 \\
  --listen-client-urls https://${PRIMARY_IP}:2379,https://127.0.0.1:2379 \\
  --advertise-client-urls https://${PRIMARY_IP}:2379 \\
  --initial-cluster-token etcd-cluster-0 \\
  --initial-cluster controlplane01=https://${CONTROL01}:2380,controlplane02=https://${CONTROL02}:2380 \\
  --initial-cluster-state new \\
  --data-dir=/var/lib/etcd
Restart=on-failure
RestartSec=5

[Install]
WantedBy=multi-user.target
EOF
```

مثال:

- هنا بيعرّف:
  - certificates عشان الـ TLS
  - IPs بتاعة كل أعضاء الـ cluster
  - data-dir : المكان اللي يخزن فيه الـ etcd البيانات.

تشغيل الخدمة

- sudo systemctl daemon-reload
- sudo systemctl enable etcd
- sudo systemctl start etcd
- sudo systemctl status etcd

- لازم تتأكد إنها Active و Running

التحقق من تشغيل الـ Cluster

- ETCDCTL_API=3 etcdctl member list --endpoints=https://127.0.0.1:2379 \
  --cacert=/etc/etcd/ca.pem --cert=/etc/etcd/kubernetes.pem \
  --key=/etc/etcd/kubernetes-key.pem

- لازم يطلعلك أعضاء الـ cluster (مثلاً master-1 و master-2)

إضافات مهمة :

1. الـ etcd هو قلب الـ Kubernetes : يعني أي corruption فيه .. cluster = كله يتأثر.
   و دايماً خد backup (باستخدام etcdctl snapshot save)
2. Performance : يفضل يكون etcd على SSD عشان سرعة القراءة/الكتابة.
3. Scaling : في production خلي etcd cluster فردي العدد (3, 5, 7) عشان الـ consensus (Raft algorithm)
4. Monitoring : تابع الـ etcd metrics (:2379/metrics) عشان تشوف latency و health

Demo - Install Control Plane Components

kubernetes-the-hard-way/docs/08-bootstrapping-kubernetes-controllers.md at master · mmumshad/kubernetes-the-hard-way

هناده هانتكلم عن تنصيب الـ Control Plane Components في Kubernetes على الـ master nodes

---

إنشاء مجلد لإعدادات Kubernetes

على كل Master Node : sudo mkdir -p /etc/kubernetes/config

تنزيل الـ Kubernetes Binaries

المكونات الأساسية:

- kube-apiserver
- kube-controller-manager
- kube-scheduler
- kubectl

مثال:

```
wget -q --https-only --timestamping \
  "https://dl.k8s.io/v1.27.0/bin/linux/amd64/kube-apiserver" \
  "https://dl.k8s.io/v1.27.0/bin/linux/amd64/kube-controller-manager" \
  "https://dl.k8s.io/v1.27.0/bin/linux/amd64/kube-scheduler" \
  "https://dl.k8s.io/v1.27.0/bin/linux/amd64/kubectl"
```

خليهم executable وانقلهم :

```
chmod +x kube-apiserver kube-controller-manager kube-scheduler kubectl
sudo mv kube-apiserver kube-controller-manager kube-scheduler kubectl /usr/local/bin/
```

نسخ الشهادات وملف التشفير (encryption config)

```
sudo mkdir -p /var/lib/kubernetes/
sudo cp ca.pem ca-key.pem kube-apiserver.pem kube-apiserver-key.pem \
  service-account.pem service-account-key.pem encryption-config.yaml /var/lib/kubernetes/
```

الحصول على الـ Internal IP

```
INTERNAL_IP=$(hostname -I | awk '{print $1}')
echo $INTERNAL_IP
```

إنشاء ملف إعداد kube-apiserver

/etc/systemd/system/kube-apiserver.service

مثال :

```
[Unit]
Description=Kubernetes API Server
Documentation=https://github.com/kubernetes/kubernetes

[Service]
ExecStart=/usr/local/bin/kube-apiserver \
  --advertise-address=${INTERNAL_IP} \
  --allow-privileged=true \
  --authorization-mode=Node,RBAC \
  --client-ca-file=/var/lib/kubernetes/ca.pem \
  --enable-admission-plugins=NodeRestriction \
  --encryption-provider-config=/var/lib/kubernetes/encryption-config.yaml \
  --etcd-servers=https://127.0.0.1:2379 \
  --etcd-cafile=/var/lib/kubernetes/ca.pem \
  --etcd-certfile=/var/lib/kubernetes/kubernetes.pem \
  --etcd-keyfile=/var/lib/kubernetes/kubernetes-key.pem \
  --kubelet-client-certificate=/var/lib/kubernetes/kubernetes.pem \
  --kubelet-client-key=/var/lib/kubernetes/kubernetes-key.pem \
  --service-account-key-file=/var/lib/kubernetes/service-account.pem \
  --service-cluster-ip-range=10.32.0.0/24 \
  --tls-cert-file=/var/lib/kubernetes/kubernetes.pem \
  --tls-private-key-file=/var/lib/kubernetes/kubernetes-key.pem \
  --v=2
Restart=on-failure
RestartSec=5

[Install]
WantedBy=multi-user.target
```

إعداد الـ Controller Manager

انسخ ملف الكونفيج لمكانه :

- sudo cp kube-controller-manager.kubeconfig /var/lib/kubernetes/

ملف الخدمة /etc/systemd/system/kube-controller-manager.service ::

مثال:

```
[Unit]
Description=Kubernetes Controller Manager
Documentation=https://github.com/kubernetes/kubernetes

[Service]
ExecStart=/usr/local/bin/kube-controller-manager \
  --kubeconfig=/var/lib/kubernetes/kube-controller-manager.kubeconfig \
  --leader-elect=true \
  --service-cluster-ip-range=10.32.0.0/24 \
  --root-ca-file=/var/lib/kubernetes/ca.pem \
  --service-account-private-key-file=/var/lib/kubernetes/service-account-key.pem \
  --use-service-account-credentials=true \
  --v=2
Restart=on-failure
RestartSec=5

[Install]
WantedBy=multi-user.target
```

إعداد الـ Scheduler

انسخ الكونفيج :

- sudo cp kube-scheduler.kubeconfig /var/lib/kubernetes/

ملف الخدمة /etc/systemd/system/kube-scheduler.service ::

```
[Unit]
Description=Kubernetes Scheduler
Documentation=https://github.com/kubernetes/kubernetes

[Service]
ExecStart=/usr/local/bin/kube-scheduler \
  --kubeconfig=/var/lib/kubernetes/kube-scheduler.kubeconfig \
  --leader-elect=true \
  --v=2
Restart=on-failure
RestartSec=5

[Install]
WantedBy=multi-user.target
```

تفعيل وإدارة الخدمات

- sudo systemctl daemon-reload
- sudo systemctl enable kube-apiserver kube-controller-manager kube-scheduler
- sudo systemctl start kube-apiserver kube-controller-manager kube-scheduler

---

التحقق من الحالة

- kubectl get componentstatuses --kubeconfig admin.kubeconfig

هتشوف حاجة زي:

```
NAME                 STATUS    MESSAGE             ERROR
scheduler            Healthy   ok
controller-manager   Healthy   ok
etcd-0               Healthy   {"health":"true"}
```

👉وبكده يبقى الـ Control Plane شغال.

Demo - Install Control Plane Load Balancer

kubernetes-the-hard-way/docs/08-bootstrapping-kubernetes-controllers.md at master · mmumshad/kubernetes-the-hard-way

الغرض الأساسي

- احنا عاوزين نعمل Load Balancer يوزّع الترافيك على أكتر من kube-apiserver لأن عندنا أكتر من Node Master في الـ HA setup
- لو الـ client (زي kubectl أو أي مكون في الكلاستر) حب يكلم الـAPI Server ، بدل ما يختار Master واحد ممكن يقع، هيكلم Load Balancer اللي يوزّع الترافيك عالسليم.

تنصيب HAProxy

- هنستخدم أداة HAProxy عشان نعمل الـ Load Balancing
- أمر التثبيت (في Ubuntu مثلاً) : sudo apt-get install -y haproxy **then** sudo apt-get update

---

تجهيز ملف الكونفيج الخاص بـ HAProxy

- بننشئ الملف /etc/haproxy/haproxy.cfg
- شرح مبسط :
  - Frontend : الـ Load Balancer بيسمع على بورت 6443 (اللي هو بورت الـ API Server)
  - Backend : بيوزع الترافيك على الـ Master Nodes (مثلاً master1 و master2)
  - Check : معناها يعمل health check عالـ API Server

تشغيل الخدمة

- بعد التعديل على الكونفيج:
  - sudo systemctl enable haproxy
  - sudo systemctl start haproxy
  - sudo systemctl status haproxy

التحقق من الوصول للـ API Server عبر Load Balancer

- دلوقتي بدل ما أكلم الـ Master Node مباشرة، أكلم الـ Load Balancer IP
  مثال : curl -k https://<LOAD_BALANCER_IP>:6443/version
- لو كل حاجة تمام .. هيرجع JSON فيه نسخة الـ Kubernetes

الاستخدام بعد كده

- في أي مكان هكتب عنوان الـ kube-apiserver (في kubeconfig أو configs تانية)
  هستعمل IP بتاع الـ Load Balancer بدل ما أستخدم IPs الـ Master Nodes

---

إضافات مهمة

- يفضل دايمًا تستخدم TCP mode مع HAProxy عشان الـ API Server بيشتغل على TLS
- ممكن تستخدم أدوات تانية للـ Load Balancing زي Nginx أو Keepalived + HAProxy لو عاوز VIP (Virtual IP) عالي الاعتمادية.
- لازم تفتح بورت 6443 في الفايروول عشان الـ Clients يقدروا يوصلوا لـ Load Balancer

Demo - Bootstrap Worker Node

kubernetes-the-hard-way/docs/10-bootstrapping-kubernetes-workers.md at master · mmumshad/kubernetes-the-hard-way

🎯 الفكرة العامة

إحنا خلصنا من إعداد الماستر نودز + LoadBalancer... دلوقتي جايين نضيف الـ Worker Nodes
لكن بدل ما نجهزهم بنفس الطريقة، هنا هنشوف طريقتين مختلفتين لإضافة Worker :

1. Manual Certificates : إحنا اللي نعمل ونوقّع وننسخ السيرتيفيكيت بإيدينا طريقة Manual
2. TLS Bootstrap : النود نفسه يولّد ويجدد السيرتيفيكيت أوتوماتيك (طريقة أوتوماتيكية).

📌 ده بيركز على الطريقة الأولى (Manual Certificates)

📝 خطوات التنفيذ

1. **توليد Certificates للـ Worker :**
   - بنبدأ من Master Node (مش من الـ Worker نفسه)
   - نعمل OpenSSL config file ونكتب فيه الـ DNS + IP بتوع الـ Worker
   - نولّد Private Key + Certificate Signing Request (CSR)
   - بعد كده نستخدم CA (Certificate Authority) اللي جهزناه قبل كده ونوقّع بيه السيرتيفيكيت.

🔑 الفكرة : كده عندنا شهادة تثبت هوية الـ Worker وموثوقة من الـ CA بتاع الكلاستر.

---

2. **تجهيز kubeconfig للـ Worker :**
   - الـ Kubelet محتاج يعرف :
     - يتكلم مع مين (يعني محتاج يعرف الـ IP بتاع Load Balancer)
     - يلاقي الشهادة فين ؟!
   - فبننشئ kubeconfig file مخصوص للـ Kubelet على الـ Worker
     بعدها ننسخ:
     - السيرتيفيكيت
     - Kubeconfig من الـ Worker Node الي الـ Master

---

3. **إعداد الـ Binaries بتاعة الـ Kubelet**
   على الـ Worker Node نفسه:
   - ننزل kubelet binaries
   - نجهز الـ folder structure (مثلاً /var/lib/Kubernetes)
   - نخلي الـ binaries executable وننقلها للـ /usr/bin
   - ننسخ السيرتيفيكيت والـ kubeconfig جوه /var/lib/Kubernetes

---

4. **إعداد Config Files**
   - نعمل Kubelet config file : فيه حاجات زي:
     - Cluster Domain
     - Cluster DNS Server
   - نعمل Systemd service file للـ Kubelet

---

5. **إعداد kube-proxy**
   - نكرر نفس الفكرة للـ kube-proxy (config + service)

---

6. **تشغيل الخدمات**

- systemctl daemon-reload
- systemctl enable kubelet
- systemctl start kubelet

نتأكد بالخدمات شغالة (systemctl status)

لو ظهر Errors عن الشبكة ففي المرحلة دي ممكن نطنش دلوقتي عادي (لأننا لسه ما ركبناش الـ Networking Plugin)

---

7. **التحقق من انضمام الـ Worker**
   - نروح على الماستر ونكتب : kubectl get nodes
   - هنلاقي الـ Worker موجود.
   - بس هيكون حالته NotReady : لأن الـ Networking مش جاهز

🔑 **ملخص مبسط**

- الطريقة دي manual :
  - إحنا نولّد السيرتيفيكيت
  - نوقّعها بالـ CA
  - ننسخها للـ Worker
  - نجهز kubeconfig + binaries + services
- النود بيظهر في الكلاستر، بس هيبقى NotReady لحد ما نكمل إعداد الشبكات.

TLS Bootstrap worker node

🎯 الفكرة العامة:

- ❖ إحنا دلوقتي خلصنا من الماستر نودز واللود بالانسر وكمان عاملين Worker Node واحد بالطريقة اليدوية (عملنا شهادات TLS بإيدينا)
- ❖ في Worker Node التاني هنستخدم طريقة أذكى وأسهل اسمها TLS Bootstrapping
الفكرة إن الـ Kubelet نفسه هو اللي هيطلب ويجدد الشهادات من الـ API Server أوتوماتيك من غير ما إحنا نعمل كل مرة العملية المرهقة دي (زي السكشن اللي فاتت).

🧩 المكونات اللي هتفرق معانا هنا:

1. أنواع الشهادات اللي محتاجينها على Worker Node :
   - Server Certificate : ده عشان الـ API Server يقدر يتصل بالـ Kubelet (مثلاً لما تجيب Logs من Pod)
   - Client Certificate : ده عشان الـ Kubelet نفسه يقدر يتصل بالـ API Server

📌 في Worker-1 إحنا جهزنا الاتنين بإيدينا. لكن في Worker-2 مش عايزين نعمل كده.

📌 So our Goal with TLS BootStrapping is to automate the cert. management so the Kubelet can take care of it by itself

📌 For using the Cert. API, the kubelet needs to be Authenticated into the kube-apiserver with the right set of permissions through "Bootstrap Token"

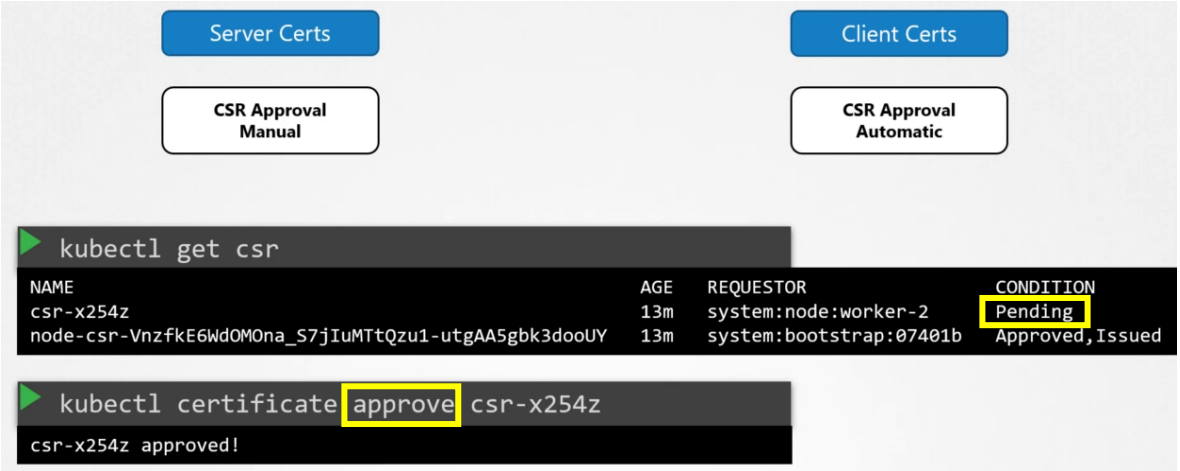
2. Bootstrap Token :
ده عبارة عن Token مؤقت بندهوله للـ Kubelet عشان يدخل يكلم الـ API Server لأول مرة.
   - بيتسجل في Group اسمه system:bootstrappers
   - وبما ان الـ Token ده معندهوش أي Permission فبالتالي بيتربط بـ Role اسمها system:node-bootstrapper : وظيفتها انها بتدي صلاحية محدودة بحيث إن الـ Kubelet يقدر يعمل CSR (Certificate Signing Request)

3. CSR Requests :
أول ما Kubelet يشتغل، هيولّد Certificate Signing Request ويبعتُه للـ API Server
   - إحنا نقدر نشوف الطلبات دي بالأمر: kubectl get csr
   - ممكن نوافق عليها يدويًا : kubectl certificate approve <csr-name>
   - أو نخليها تتوافق أوتوماتيك لو ربطنا Group system:bootstrappers بـ Role إضافي اسمه : system:certificates.k8s.io:certificatesigningrequests:nodeclient

📌 كده أي Worker جديد هيدخل لوحده من غير تدخل.

🔄 تجديد الشهادات:

- Client Certificates : تتجدد أوتوماتيك (لو فعلنا الـ Flag --rotate-certificates=true)
- Server Certificates : لازم موافقة يدوية لأسباب أمنية. بنشوفها Pending ونوافق عليها يدويًا بنفس أمر الـ approve

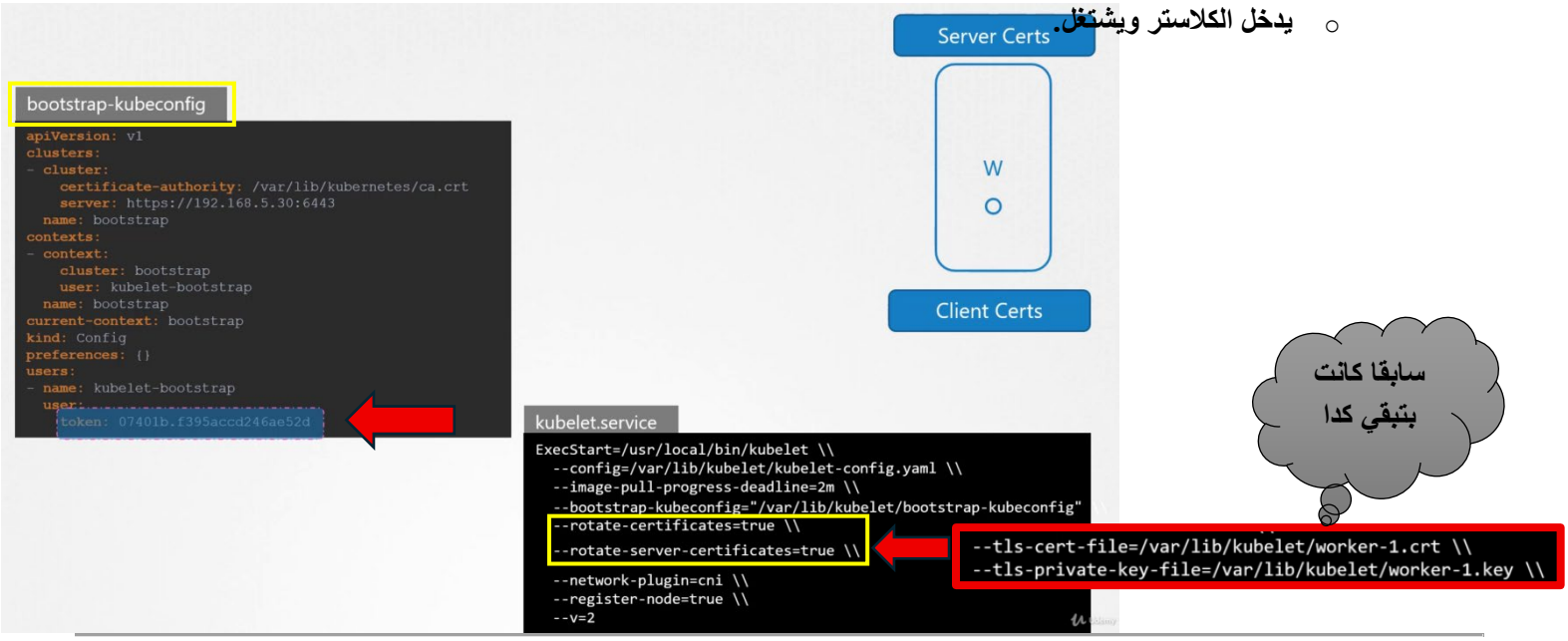


⚙️ خطوات التفعيل عمليًا:

1. على Master Node :
   - نعمل الـ Bootstrap Token : kubeadm token create --print-join-command
     هايطلعلك توكن جاهز تستخدمه
   - نربطه بالـ Group المناسب (system:bootstrappers)
   - نربطه بالـ Role (system:node-bootstrapper) عشان يقدر يبعت CSR

2. على Worker Node الجديد:
   - نعدل Config الـ Kubelet ونضيف Flag :
   - --bootstrap-kubeconfig=/var/lib/kubelet/bootstrap-kubeconfig
   - --rotate-certificates=true
   - --rotate-server-certificates=true
   - لاحظ ان الملف bootstrap-kubeconfig بيحتوي التوكن ده بدل الشهادات.

3. النتيجة:
   - أول ما النود يشتغل .. علطول يبعت CSR .. يا يتوافق يدويًا، يا أوتوماتيك (حسب إعدادنا).
   - يدخل الكلاستر ويشتغل.



📝 الخلاصة:

- TLS Bootstrapping يخلي النود يجهّز نفسه بنفسه بدل ما إحنا نعمل Certificates بإيدينا.

- التوكن مؤقت بس للبداية .. بعد كده النود نفسه يدخل Group تاني (system:nodes) ويجدد شهاداته بنفسه.
- الـ Client Certificates بتتوافق أوتوماتيك، لكن الـ Server Certificates لازم Approval يدوي.

Demo - TLS Bootstrap worker node

kubernetes-the-hard-way/docs/11-tls-bootstrapping-kubernetes-workers.md at master · mmumshad/kubernetes-the-hard-way

TLS Bootstrapping a Worker Node (worker-2)

الموضوع هنا عن إضافة Worker Node جديد (worker-2) للـ Kubernetes Cluster باستخدام TLS Bootstrapping
الهدف : إن الـ kubelet يقدر يجيب Certificate من الـ API Server بشكل تلقائي بدل ما نعمله مانيوال زي ما عملنا مع worker-1

المتطلبات (Prerequisites)

❖ لازم نتاكد من حاجتين في الـ Control Plane :

1. Enable bootstrap token auth
   - لازم الـ API Server يكون شغال بخيار : --enable-bootstrap-token-auth=true
   - بنأكد ده بـ : ps -aux | grep kube-apiserver

ونشوف الفلاج موجود ✅ .

2. Cluster signing certificate/key
   - الـ Controller Manager لازم يكون متوصل بـ Certificate Signing Key :
   - --cluster-signing-cert-file
   - --cluster-signing-key-file
   - بنأكد برضه بـ : ps -aux | grep kube-controller-manager

تجهيز الـ Worker Node

على worker-2 :

- نزّل الـ binaries : (kubectl – kube-proxy - kubelet)
- اعمل الـ Directories وحط الملفات في مكانها:
- mkdir -p /var/lib/kubelet /var/lib/kube-proxy /etc/kubernetes/manifests
- mv kubelet kube-proxy kubectl /usr/bin/
- انقل CA Certificate عشان نثق في الـ API Server : cp ca.crt /var/lib/kubernetes/

Bootstrap Token

❖ لإننا ما عندناش Certificate للـ kubelet ، محتاجين نعمل Bootstrap Token

1. نعمل Secret للـ Token :

❖ kubectl create secret bootstrap token <token> \
  --namespace=kube-system \
  --type="bootstrap.kubernetes.io/token"

2. ندي Permission للـ Worker إنه يعمل CSR (Certificate Signing Request) :

❖ kubectl create clusterrolebinding kubelet-bootstrap \
  --clusterrole=system:node-bootstrapper \
  --user=system:bootstrap:<token-id>

3. ندي Permission برضه إنه يـ Approve CSRs و Renew Certificates :

❖ kubectl create clusterrolebinding node-autoapprove-csr \
  --clusterrole=system:certificates.k8s.io:certificatesigningrequests:nodeclient \
  --group=system:bootstrappers

Bootstrap Kubeconfig

بدل ما نعمل kubeconfig بـCertificates ، هنا بنعمله بـ Bootstrap Token

```
❖ kubectl config set-cluster kubernetes \
    --certificate-authority=/var/lib/kubernetes/ca.crt \
    --embed-certs=true \
    --server=https://<LOADBALANCER_IP>:6443 \
    --kubeconfig=bootstrap.kubeconfig

❖ kubectl config set-credentials kubelet-bootstrap \
    --token=<bootstrap-token> \
    --kubeconfig=bootstrap.kubeconfig

❖ kubectl config set-context default \
    --cluster=kubernetes \
    --user=kubelet-bootstrap \
    --kubeconfig=bootstrap.kubeconfig

❖ kubectl config use-context default --kubeconfig=bootstrap.kubeconfig
```

Configure Kubelet + Kube-proxy

- اعمل ملف Config للـ kubelet يحدد الـ environment
- شغّل الخدمة بـ Systemd وخليها تستخدم : --bootstrap-kubeconfig=/var/lib/kubelet/bootstrap.kubeconfig
  - بدل : --kubeconfig=/var/lib/kubelet/kubeconfig
- نفس القصة للـ kube-proxy بس بالطريقة العادية.

شغل الخدمات

```
❖ systemctl daemon-reload
❖ systemctl enable kubelet kube-proxy
❖ systemctl start kubelet kube-proxy
❖ systemctl status kubelet
```

تأكيد من الـ Master

- شوف الـ CSRs : kubectl get csr
- معظم الـ kubelet CSRs بيتوافق عليها أوتوماتيك. لكن لو لسه Pending :
- kubectl certificate approve <csr-name>

- اتأكد من الـ Nodes : kubectl get nodes

اخيرا هتلاقي worker-2 اتضاف للـ Cluster

🔑 الملخص والفكرة الأساسية:

- Worker-1 = Manual Certificates
- Worker-2 = TLS Bootstrapping (يطلب Certificate بنفسه باستخدام Bootstrap Token)

Configure Kubectl for Remote Access

kubernetes-the-hard-way/docs/12-configuring-kubectl.md at master · mmumshad/kubernetes-the-hard-way

إزاي نعمل Kubeconfig file عشان نقدر نوصل للـ Kubernetes cluster من جهاز خارجي (Remote Access)

✅ الخطوات

1. **إنشاء ملف Kubeconfig**
   - الـ Kubeconfig بيحتوي على:
     - Cluster info (اسم الكلاستر + الـ API Server IP أو Load Balancer IP)
     - User credentials (Admin user + Certificate/Key)
     - Context (الربط بين user والـ cluster)

مثال لإنشاء الملف يدويًا:

```
kubectl config set-cluster my-cluster \
  --server=https://<LOADBALANCER-IP>:6443 \
  --certificate-authority=/etc/kubernetes/pki/ca.crt \
  --embed-certs=true
```

بعدين نضيف اليوزر (admin user) :

```
kubectl config set-credentials admin \
  --client-certificate=/etc/kubernetes/pki/admin.crt \
  --client-key=/etc/kubernetes/pki/admin.key \
  --embed-certs=true
```

نعمل Context يربط اليوزر بالـcluster:

```
kubectl config set-context my-context \
  --cluster=my-cluster \
  --user=admin
```

وأخيرًا نحدد الـ context كـ default : `kubectl config use-context my-context`

---

2. **مكان حفظ الملف**
   - بيتخزن في: `~/.kube/config`
   - لازم نتأكد إنه موجود هناك: `ls -l ~/.kube/config`

---

3. **التحقق من الاتصال بالـ Cluster**

- نستخدم أوامر test :
  - ده بيعرض حالة المكونات الأساسية.(scheduler, controller-manager, etcd) :

```
kubectl get componentstatuses
```

  - ده بيتأكد إننا شايفين النودز اللي في الكلاستر : `kubectl get nodes`

📌 الخلاصة

- أنشأنا ملف Kubeconfig فيه Certificates + User + Cluster info
- خزنّاه في ~/.kube/config.
- عملنا test باستخدام `kubectl get componentstatuses` و `kubectl get nodes` للتأكد إن الاتصال بالـ cluster ناجح.

Demo - Deploy POD Networking Solution – Weave

kubernetes-the-hard-way/docs/13-configure-pod-networking.md at master · mmumshad/kubernetes-the-hard-way

تجهيز الـ Networking للـ Kubernetes Cluster باستخدام Weave CNI plugin، بحيث نقدر نشغّل الـ Pods على الـ worker nodes بشكل سليم.

✅ الخطوات العملية

التأكد من حالة الكلاستر : kubectl get nodes

- هتشوف إن عندك 2 Nodes (Master + Worker)
- حالة الـ Worker Node = NotReady

معرفة سبب المشكلة : kubectl describe node <worker-node-name>

- هتلاقي الرسالة هتقول: **Network plugin is not configured**

يعني مفيش CNI plugin متثبت، وده السبب إن النود مش جاهزة.

تجهيز الـ CNI Plugins الافتراضية

قبل ما نستخدم Weave ، لازم ننزل الـ CNI plugins الأساسية (اللي Weave بيعتمد عليها)

- ننزل الـ CNI binaries :

❖ wget https://github.com/containernetworking/plugins/releases/download/v1.5.0/cni-plugins-linux-amd64-v1.5.0.tgz

- نفك الضغط ونحط الملفات في المكان المطلوب:

❖ sudo mkdir -p /opt/cni/bin

❖ sudo tar -C /opt/cni/bin -xzf cni-plugins-linux-amd64-v1.5.0.tgz

- نكرر نفس الخطوات دي على كل Worker Node

نشر Weave CNI Plugin

- من أي Master Node نطبق الـ Weave manifest الجاهز:

❖ kubectl apply -f "https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version | base64 | tr -d '\n')"

📌 إيه اللي بيحصل؟

- بيتعمل Deploy لـ Weave Agents كـ DaemonSet (يعني أوتوماتيك بيتنزل Pod على كل Node في الكلاستر)

مراقبة حالة الـ Pods : kubectl get pods -n kube-system

📌 هتشوف الـ Weave Pods نازلة، وتستنى لحد ما تبقى Running.

التحقق من حالة النودز : kubectl get nodes

📌 دلوقتي حالة الـ Worker Nodes = Ready ✅.

يبقى الشبكة اتظبطت والـ Pods يقدروا يتواصلوا مع بعض جوه الكلاستر.

🔑 الخلاصة

- **المشكلة : النودز مش جاهزة بسبب عدم وجود Network Plugin**
- **الحل:**
  1. **نزلنا CNI plugins الأساسية في /opt/cni/bin**
  2. **نشرنا Weave CNI كـ DaemonSet**
- **النتيجة : النودز بقت Ready والشبكة شغالة.**

Demo - Kube-api to Kubelet connectivity

kubernetes-the-hard-way/docs/14-kube-apiserver-to-kubelet.md at master · mmumshad/kubernetes-the-hard-way

التحقق من الـ Cluster بعد التجهيز

- بعد ما خلصنا إعداد الـ cluster (Master + Workers + Networking)
  - بنعمل : kubectl get nodes

لو كله تمام هنلاقي النودز جاهزة (Ready)

  - بعدين : kubectl get pods -A

بنراجع إن كل الـ system pods (زي kube-dns, weave, etc شغالة)

مشكلة عرض الـ Logs

- دلوقتي عايزين نعرض اللوجات الخاصة بـ Weave Pod أو أي Pod
- الـ Weave Pod فيه أكتر من Container ، فلازم نحدد أي Container :

❖ kubectl logs <pod-name> -c <container-name> -n kube-system

- لما جربنا، ظهر. 🚫 Error: Forbidden !!!!

السبب

- السؤال : إزاي الـ kube-apiserver اللي على الـ (Master) بيجيب Logs من Pod موجود على Worker Node ؟!
- الإجابة: بيكلم kubelet اللي على كل Worker Node
- وبالتالي فالمشكلة إن الـ API Server لسه ما عندوش صلاحيات إنه يتكلم مع الـ kubelet

---

الحل : RBAC (Role-Based Access Control)

لازم ندي صلاحيات للـ API Server يتواصل مع الـ kubelet :

✅ (أ) نعمل ClusterRole :

- الدور ده يسمح بالوصول لموارد زي:
  - nodes/proxy
  - stats
  - log
  - spec
  - metrics

مثال تعريف ClusterRole (yaml :

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: kube-apiserver-to-kubelet
rules:
  - apiGroups: [""]
    resources:
      - nodes/proxy
      - nodes/stats
      - nodes/log
      - nodes/spec
      - nodes/metrics
    verbs: ["*"]
```

✅ (ب) نعمل ClusterRoleBinding :

- نربط الدور ده بالـ kube-apiserver

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: kube-apiserver-to-kubelet
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: kube-apiserver-to-kubelet
subjects:
  - kind: User
    name: kube-apiserver
    apiGroup: rbac.authorization.k8s.io
```

التحقق

- بعد ما نعمل الدور والـ binding :

❖ **kubectl logs <pod-name> -c <container-name> -n kube-system**

هتشتغل وتعرض الـ logs عادي. ✅

ملاحظة مهمة

- لو في أي وقت لقيت مشكلة في :
  - kubectl logs
  - kubectl exec

يبقى أول حاجة تفكر فيها : هل الـ kube-apiserver عنده الصلاحيات يتكلم مع الـ kubelet ولا لأ ؟!؟!؟!

Deploy core DNS

kubernetes-the-hard-way/docs/15-dns-addon.md at master · mmumshad/kubernetes-the-hard-way

🎯 هانتكلم عن آخر خطوة وهي إضافة الـ DNS Cluster Add-on (CoreDNS) في الكلاستر

الفكرة

زي ما عملنا مع الـ Networking (Weave أو Flannel) فإحنا محتاجين نوفر DNS داخلي للكلاستر بحيث أي Pod يقدر يعمل Resolve لأي Service بالـ ClusterIP بتاعها.

📌 الحل الافتراضي في Kubernetes هو CoreDNS

الخطوات العملية

1. نطبّق المانيفست الجاهز بتاع CoreDNS :

❖ kubectl apply -f https://storage.googleapis.com/kubernetes-the-hard-way/coredns.yaml

📌 المانيفست ده بينشئ الحاجات الآتية:

- ServiceAccount خاص بـ coredns
- ClusterRole & RoleBinding علشان يديله صلاحيات.
- ConfigMap فيه إعدادات CoreDNS
- Deployment بيشغل الـ CoreDNS Pods
- Service علشان يخلي الـ CoreDNS متاح جوه الكلاستر.

---

2. ننتظر لحد ما الـ Pods تشتغل : kubectl get pods -n kube-system
   - هتشوف حاجة زي:

coredns-xxxx Running
coredns-xxxx Running

---

3. نعمل Pod تجريبي (BusyBox) علشان نختبر الـ DNS :

❖ kubectl run busybox --image=busybox:1.28 -- sleep 3600

---

4. ندخل جوه الـ Pod : kubectl exec -it busybox -- sh

---

5. نعمل اختبار DNS Resolution : nslookup kubernetes

📌 الناتج هيكون حاجة زي:

```
Server:    10.96.0.10
Address 1: 10.96.0.10 kube-dns.kube-system.svc.cluster.local
Name:      kubernetes
Address 1: 10.96.0.1
```

- الـ 10.96.0.10 ده الـ ClusterIP بتاع CoreDNS
- والـ 10.96.0.1 ده Kubernetes "Service" نفسه.

الخلاصة

- CoreDNS هو DNS الداخلي للكلاستر.
- أي Pod يقدر يحل أسماء السيرفيسات (service discovery)
- بكده الكلاستر جهز بالكامل (Control plane + Worker nodes + Networking + DNS)

Install Kubernetes with Kubeadm – Introduction

🎯 الموضوع الأساسي

هانتكلم عن kubeadm ودي أداة بتسهل إعداد Kubernetes Cluster .
بدل ما تنزل كل مكوّن من مكونات Kubernetes (API Server, etcd, Controller Manager, Scheduler, Kubelet ...)
(يدويًا وتضبط الـ Certificates والـConfigurations)
➕ فالـ kubeadm بيعمل ده كله بالـ Best Practices بشكل أوتوماتيك.

🔑 الخطوات الرئيسية لإنشاء Cluster باستخدام kubeadm

1 تجهيز الـ Systems (الـ VMs أو Physical Machines)

- لازم يكون عندك كذا جهاز أو Virtual Machines
- واحد هتسميه Master (Control Plane Node) والباقي هيبقوا Worker Nodes

📌 ملحوظة:
ممكن تجرب على لابتوبك باستخدام VirtualBox أو VMware أو حتى Minikube لو للتجارب.

---

2 تثبيت Container Runtime

- Kubernetes مش بيشغّل الـ Containers لوحده ، محتاج Container Runtime
- هنا اختارنا Docker (بس خلي باااالك .. ممكن برضه containerd أو CRI-O)

🔧 أمر عملي لتثبيت Docker مثال على Ubuntu :

```
sudo apt update
sudo apt install -y docker.io
sudo systemctl enable docker
sudo systemctl start docker
```

---

3 تثبيت kubeadm على كل Node

- لازم تنزل kubeadm + kubelet + kubectl على كل الأجهزة.
- kubeadm : بيعمل bootstrap للـ Cluster
- kubelet : بيدير الـ Pods على كل Node
- kubectl : أداة التحكم من الـ CLI

🔧 أوامر عملية على Ubuntu :

```
sudo apt update && sudo apt install -y apt-transport-https curl
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -
echo "deb https://apt.kubernetes.io/ kubernetes-xenial main" | sudo tee \
    /etc/apt/sources.list.d/kubernetes.list
sudo apt update
sudo apt install -y kubelet kubeadm kubectl
sudo apt-mark hold kubelet kubeadm kubectl
```

4️⃣ تهيئة Master Node

- على الماستر بتعمل Initialize للـ Cluster : sudo kubeadm init --pod-network-cidr=10.244.0.0/16

📌 الشرح:

- --pod-network-cidr : مدى الـ IPs اللي هيتوزع على الـ Pods (هتحتاجه للـ CNI مثل Flannel/Weave)
- الأمر ده بيعمل Install لكل مكونات Control Plane (API Server, etcd, Controller Manager, Scheduler). بعد ما يخلص، هيطبعلك أمر عشان تضيف الـ Worker Nodes للـ Cluster

---

5️⃣ إعداد kubectl على Master

- عشان تستخدم kubectl من الماستر نفسه:

```
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

6️⃣ تثبيت Pod Network (CNI Plugin)

- عشان الـ Pods تتواصل مع بعض، لازم Network Plugin
- مثل Flannel أو Weave أو Calico

🔧 مثال لتثبيت Weave Net :

```
kubectl apply -f "https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version | base64 | tr -d '\n')"
```

7️⃣ إضافة Worker Nodes للـ Cluster

- من الأمر اللي kubeadm طبعه بعد الـ init :

```
kubeadm join <master-ip>:6443 --token <token> --discovery-token-ca-cert-hash sha256:<hash>
```

📌 ده بيربط الـ Worker مع الـ Master

8️⃣ تحقق من حالة الـ Cluster

```
kubectl get nodes
kubectl get pods -A
```

هتشوف الماستر والوركرز Up and Running

📝 ملحوظات إضافية

- لازم تفتح الـ Ports المطلوبة في الـ Firewall (6443 للـ API Server, 10250 للـ kubelet .....إلخ)
- الأفضل تبدأ بـ Cluster صغير (1 Master + 2 Workers) للتجارب.
- بعد كده ممكن تزود Workers أو تعمل HA بإنك تزود أكثر من Master

Deploy with Kubeadm – Provissioning

🎯 الموضوع الأساسي

المحاضرة دي عبارة عن Demo عشان يورينا إزاي نجهز الـ Virtual Machines (1 Master + 1 Workers)
باستخدام Vagrant + VirtualBox قبل ما نبدأ بقى نعمل Bootstrapping للـ Kubernetes Cluster بـ kubeadm

---

🔑 الخطوات اللي هانمشي عليها

1 المتطلبات الأساسية

- لازم يبقى عندك VirtualBox (Virtualization Software)
- وكمان Vagrant (ودي أداة Automation بتسهّل إنشاء الـ VMs بدل ما تعملها يدوي)

🔧 أوامر عملية لتثبيت على Ubuntu :

- ❖ sudo apt update
- ❖ sudo apt install -y virtualbox vagrant

---

2 استخدام Vagrantfile

- بدل ما تكتب إعدادات كل VM يدويًا (RAM, CPU, Network, IP ...) ، بيكون عندنا Vagrantfile فيه كل الـ Configurations
- الملف ده موجود في GitHub Repo تابع للكورس.

kodekloudhub/certified-kubernetes-administrator-course: Certified Kubernetes Administrator - CKA Course

🔧 أوامر عملية:

#نسخ المشروع من GitHub : git clone <repo-url>
#الدخول جوة المشروع : cd <repo-directory>

---

3 مراجعة إعدادات Vagrantfile

- في الـ Vagrantfile ، معمولة إعدادات زي:
  - عدد الـ Nodes (1 Master + 2 Workers)
  - IP Range → 192.168.56.0/24 (يعني الـ VMs هياخدوا IPs من هنا)

📌 مثال من الملف:

- ❖ master.vm.network "private_network", ip: "192.168.56.10"
- ❖ worker1.vm.network "private_network", ip: "192.168.56.11"
- ❖ worker2.vm.network "private_network", ip: "192.168.56.12"

---

4 التحقق من حالة الـ VMs قبل الإنشاء

- نشوف حالة الـ VMs الافتراضية (لسه مش متخلقة) : vagrant status

📌 النتيجة : كلهم "not created"

- لكن بتظهر أساميهم اللي هيتخلقوا بيها:
  - kube-master
  - kube-node-01
  - kube-node-02

5 تشغيل وإنشاء الـ VMs

- لإنشاء الـ VMs وتشغيلها : vagrant up

📌 اللي بيحصل:

- بيستخدم Ubuntu Bionic 64 كـ Base Image
- يسحب الصورة (لو مش موجودة) ويبدأ يجهز الماستر وبعده الوركرز.
- ممكن ياخد وقت حسب سرعة النت + إمكانيات الجهاز.

---

6 التحقق بعد الإنشاء

- نرجع نعمل : vagrant status

📌 المره دي كلهم هيظهروا "running"

---

7 الدخول على أي VM

- ندخل على الماستر مثلاً : vagrant ssh kube-master
- كده انت جوة الماستر.

🔧 جرب أي أمر بسيط زي: uptime
عشان تتأكد إن كل حاجة شغالة.

---

8 اختبار الاتصال بين الـ Nodes

- من الماستر تقدر تعمل ping على الـ Workers :

```
ping 192.168.56.11      # worker1
ping 192.168.56.12      # worker2
```

📌 الهدف: نتأكد إن الشبكة الداخلية بين الـ VMs شغالة كويس.

📝 ملحوظات إضافية

- ليه Vagrant مفيد ؟
  - بيوفرلك بيئة جاهزة بـ "config file" : يعني لو عايز تعيد الإنشاء أو تدي نفس الإعدادات لحد تاني، سهل جدًا.
- ممكن تعمل destroy وإعادة تشغيل بسهولة : vagrant up Then vagrant destroy -f
- ده مجرد إعداد للبنية التحتية (الـ VMs) : في التوبيك الجاي هانبدأ نجهز الـ Kubernetes Cluster نفسه باستخدام kubeadm

Demo - Deployment with Kubeadm

https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/

🎯 الموضوع الأساسي

- الشرح ده هو Bootstrapping لـ Kubernetes Cluster باستخدام kubeadm بعد ما خلصنا إعداد الـ VMs (Master + Workers)
- الخطوات ماشية من :

➢ **Prerequisites → Install Docker → Install kubeadm/kubelet/kubectl → Init Cluster → Add Networking → Join Workers → Test Pod.**

🔑 الخطوات بالتفصيل

1 الـ Prerequisites

- لازم نظام تشغيل مدعوم Ubuntu 16.04 أو أعلى
- Hardware : على الأقل 2GB RAM + Network connectivity بين الـ Nodes
- لازم تتأكد إن Firewall و iptables modules شغالة عشان الـ Nodes يعرفوا يكلموا بعض.

🔧 أوامر عملية:

- تأكد إن module بتاع bridge-nf-call-ip6tables/ip4tables شغال : `lsmod | grep br_netfilter`
- لو مش موجود ففعّله علي كل الـ Nodes : `sudo modprobe br_netfilter`

---

2 تثبيت Container Runtime (Docker)

- Kubernetes محتاج Container Runtime
- هنا بيستخدم Docker

🔧 أوامر عملية (Ubuntu) :

- تحديث النظام : `sudo apt update`
- تثبيت الـ repos :

```
sudo apt install -y apt-transport-https ca-certificates curl software-properties-common
```

- إضافة GPG Key بتاع Docker :

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
```

- إضافة الـ repo :

```
sudo add-apt-repository \
  "deb [arch=amd64] https://download.docker.com/linux/ubuntu  $(lsb_release -cs)  stable"
```

- تثبيت Docker :
  - ❖ `sudo apt update`
  - ❖ `sudo apt install -y docker-ce`
- بعد التثبيت، يتأكد إن الخدمة شغالة : `systemctl status docker`

---

3 تثبيت Kubernetes Components

- لازم نثبت الـ Tools دي :
  - kubeadm : بيعمل Bootstrap للـ Cluster
  - kubelet : الـ Agent اللي بيشتغل على كل Node ويدير الـ Pods
  - kubectl : CLI للتعامل مع الـ Cluster

🔧 أوامر عملية:

- إضافة الـ repo الرسمي

❖ sudo curl -fsSLo /usr/share/keyrings/kubernetes-archive-keyring.gpg https://packages.cloud.google.com/apt/doc/apt-key.gpg

❖ echo "deb [signed-by=/usr/share/keyrings/kubernetes-archive-keyring.gpg] \
https://apt.kubernetes.io/ kubernetes-xenial main" | \
sudo tee /etc/apt/sources.list.d/kubernetes.list

- تثبيت الأدوات

❖ sudo apt update
❖ sudo apt install -y kubelet kubeadm kubectl

- قفل الإصدار (مهم عشان ميترفعش فجأة)

❖ sudo apt-mark hold kubelet kubeadm kubectl

4️⃣ تهيئة الـ Master (Initialize Cluster)

- على الماستر بس نشغل:

❖ sudo kubeadm init \
--pod-network-cidr=10.244.0.0/16 \
--apiserver-advertise-address=192.168.56.2

📌 شرح الـ Parameters بتاع الأمر السابق :

- --pod-network-cidr : الـ IP range اللي الـ CNI plugin (زي Weave, Flannel) هيستخدمه.
- --apiserver-advertise-address : الـ IP بتاع الـ Master اللي الـ Workers هيكلموه عليه.
- الأمر ده بيعمل :
  - Pre-flight checks.
  - Certificates.
  - Kubernetes Control Plane (API Server, etcd, Scheduler, Controller Manager).

5️⃣ تهيئة kubectl للـ User

- بعد ما يخلص init ، يديك أوامر:

❖ mkdir -p $HOME/.kube
❖ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
❖ sudo chown $(id -u):$(id -g) $HOME/.kube/config

📌 الهدف : تقدر تشغل أوامر kubectl كـ User عادي.

6️⃣ إضافة Networking Plugin (Weave Net)

- Kubernetes مش بيشتغل من غير CNI Plugin
- هنا هانستخدم Weave Net :

kubectl apply -f "https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version | base64 | tr -d '\n')"

📌 ده بيعمل Deploy لـ DaemonSet على كل Node عشان يربط الـ Pods ببعض.

7️⃣ Join Workers للـ Cluster

- بعد الـ kubeadm والـ init هايطلعلك أمر kubeadm join
مثال :

❖ kubeadm join 192.168.56.2:6443 --token abcdef.0123456789abcdef \
--discovery-token-ca-cert-hash sha256:xxxxxxxxxxxxxxxxxxxxxxxxxxxx

- نشغله على كل Worker

8️⃣ التحقق من الـ Nodes

- على الماستر : kubectl get nodes

📌 هتشوف master + workers ، الأول بيبقوا "NotReady" لحد ما الـ CNI يشتغل، بعدين يبقوا "Ready"

9️⃣ اختبار بإنشاء Pod

- جرب تعمل Pod عادي : kubectl run nginx --image=nginx
- راقب الحالة : kubectl get pods -o wide
- ولما تخلص امسحه : kubectl delete pod nginx

📝 ملحوظات إضافية

- لو نسيت أمر join تقدر تجيبه تاني : kubeadm token create --print-join-command
- اختيار الـ CNI مهم جدًا(Weave, Flannel, Calico) والـ Weave مناسب للتجارب.
- رقم فردي من الـ Masters (3 أو 5) مطلوب لو بتعمل HA Cluster

Solution - Deploy a Kubernetes Cluster using kubeadm (Optional)

- إحنا هنا هانعمل حل عملي (Solution) بيجاوب على شوية أسئلة/تمارين (8 أسئلة).
- الهدف : تعمل Cluster بسيط فيه Master Node و Worker Node(s)

---

1. ◆ تثبيت kubeadm, kubelet, kubectl

- أول خطوة : نثبت الباكجات الأساسية على الـ Master والـ Worker :
  - kubeadm : الأداة اللي بتعمل Initialize للـ Cluster
  - kubelet Agent : موجود على كل Node بيدير الـ Pods
  - kubectl : أداة CLI اللي بنستخدمها نتكلم مع Kubernetes

📌 أوامر عملية (Ubuntu/Debian) :

- ❖ sudo apt-get update
- ❖ sudo apt-get install -y apt-transport-https ca-certificates curl
- ❖ sudo curl -fsSLo /usr/share/keyrings/kubernetes-archive-keyring.gpg https://packages.cloud.google.com/apt/doc/apt-key.gpg
- ❖ echo "deb [signed-by=/usr/share/keyrings/kubernetes-archive-keyring.gpg] \ https://apt.kubernetes.io/ kubernetes-xenial main" | sudo tee \ /etc/apt/sources.list.d/kubernetes.list
- ❖ sudo apt-get update
- ❖ sudo apt-get install -y kubelet kubeadm kubectl

---

2. ◆ تهيئة الـ Master Node

- بعد ما نثبت الحزم :
  - نروح على الـ Master Node
  - نعمل Initialize للـ Cluster باستخدام kubeadm

📌 أمر عملي : sudo kubeadm init

- ده بيعمل :
  - Pre-flight checks : يتأكد إن كل حاجة متجهزة
  - يجهز Certificates
  - يشغل الـ Control Plane components (API Server, etcd, Controller Manager, Scheduler)
  - يطبعلك kubeadm join command (مهم جدًا) لأن دا اللي من خلاله هتقدر تضيف Worker Nodes

```
Your Kubernetes control-plane has initialized successfully!

To start using your cluster, you need to run the following as a regular user:

  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
  https://kubernetes.io/docs/concepts/cluster-administration/addons/

Then you can join any number of worker nodes by running the following on each as root:
```

3. إعداد ملف kubeconfig

- بعد ما نعمل init ، لازم نظبط الـ kubectl (بمعني انه يشتغل من حساب الـ User مش root)

📌 أوامر عملية:

- ❖ mkdir -p $HOME/.kube
- ❖ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
- ❖ sudo chown $(id -u):$(id -g) $HOME/.kube/config

- الملف ده (admin.conf) فيه معلومات الاتصال بالـ API Server

---

4. إضافة Worker Node

- من النتيجة بتاعة init بيطلعلك أمر شبه ده:

kubeadm join <Master-IP>:6443 --token <token> --discovery-token-ca-cert-hash sha256:<hash>

- وممكن تجيب الامر السابق ده من خلال : kubeadm token create –print-join-command هايطلعلك نفس النتيجة

وبعدين تشغل الـ o/p بتاع الأمر السابق على كل Worker Node علشان يدخلوا في الـ Cluster

---

5. إضافة Network Plugin

- الـ Nodes بعد الانضمام بيبقوا NotReady لحد ما نضيف CNI Plugin (مثلاً Weave أو Calico)

📌 مثال (Weave) :

kubectl apply -f "https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version | base64 | tr -d '\n')"

- ده بينزل Deployment + DaemonSet يشغل الـ Weave Net على كل Node
- بعد كده الـ Nodes هتبقى Ready

---

6. التحقق من الـ Cluster

- نتاكد إن كل حاجة شغالة : kubectl get nodes

```
NAME      STATUS   ROLES           AGE   VERSION
master    Ready    control-plane   10m   v1.27.3
node01    Ready    <none>          5m    v1.27.3
```

---

7. اختبار الـ Cluster

- نعمل Pod بسيط (nginx) ونتأكد إنه اشتغل.

📌 أمر عملي:

- ❖ kubectl run nginx --image=nginx
- ❖ kubectl get pods

✅ الخلاصة

- ❖ Master Node : kubeadm init
- ❖ Worker Node(s) : kubeadm join
- ❖ Network Plugin : Weave/Calico
- ❖ Check : kubectl get nodes

End To End Tests

المقدمة

هانشرح إزاي نتحقق (Validate) من أن الـ Kubernetes Cluster شغال كويس وكل الـ Components فيه تمام.

وطبعا التحقق مهم لأن كوبرنيتس معقد وبيجمع خدمات كتير (API Server, Scheduler, etcd ...)

---

1 Manual Validation (التحقق اليدوي)

قبل ما نستخدم Tools جاهزة، ممكن نبدأ باختبارات يدوية:

- Check Nodes : kubectl get nodes .. بحيث نشوف كل الـ Nodes صحيتهم (Ready / NotReady).
- Check Pods : kubectl get pods -A
- خاصة الـ Pods اللي في Namespace : kube-system .. لأن فيها Control Plane Components.
- Check Services :
  - لو الـ Control Plane مش متثبت كـ Pods زي في Kops أو Managed Clusters يبقى نشوف الـ Services : service kube-apiserver status مع استبدال apiserver بكل مُكون
- Try Deploy App :
  - نعمل Deploy لـ Nginx : kubectl create deploy nginx --image=nginx
  - بعدين نعمل Scale : kubectl scale deploy nginx --replicas=3
    نتأكد إن البودات توزعت على كل النودز.
- Expose Service : kubectl expose deploy nginx --port=80 --type=NodePort
  ونجرب نوصل للتطبيق.
- اختبارات إضافية : Secrets, ConfigMaps, Volumes, Networking ...

---

2 End-to-End Tests (E2E Tests) (هانشرحه بالتفصيل في التوبيك الجاي)

- الاختبارات اليدوية بتكبر جدًا (مئات الاختبارات).
- هنا بيجي دور الـ Test Suites
- كوبرنيتس عنده Tool اسمه Kube-Test (ودا جزء من test-infra repo على GitHub)
- بيشغل حوالي 1000 Test في مجالات مختلفة:
  - API
  - CLI
  - Networking
  - Scheduling
  - Storage & Security

---

3 أمثلة على الاختبارات

- Networking :
  - Pod يقدر يكلم Pod تاني عبر HTTP
  - الـ Services لازم ترد من الـ Endpoints
  - Latency لازم يكون قليل.
  - DNS يحل أسماء الـServices
- Storage :
  - Secrets و ConfigMaps تتقرأ من الـ Pods
  - Mounted Volumes شغالة.

4 طريقة عمل الاختبار

- بيعمل Namespace خاص بالاختبار.
- ينشئ فيه الـ Objects (Pods, Services)
- يستنى لحد ما يقوموا.
- يجري الاختبار مثلاً يستخدم Curl من Pod لـ Pod
- يسجل النتيجة.
- يمسح الـ Namespace (Cleanup)

---

5 Use Cases (حالات الاستخدام)

- ممكن تستخدمه عشان تختبر Cluster موجود.
- أو تبني Kubernetes Binaries خاصة بيك ثم تعمل Deploy للـ Cluster و تشغل الاختبارات ثم تمسحه.
- ده مفيد لو بنيت حل خاص للـ Cluster (Automation Tool مثلًا)

---

6 Conformance Tests

- من الـ 1000 Test ، فيه حوالي 160 Test أساسيين.
- دول اسمهم Conformance Tests
- لازم أي Cluster يدّعي إنه متوافق مع Kubernetes يعدي الاختبارات دي.
- بعد ما تنجح، ترفع النتائج على Test-Grid وتبقى Kubernetes Certified Solution Provider

⏱️ الوقت:

- Full E2E (1000 Test) = ~12 Hours
- Conformance (~160 Test) = 1-2 Hours

---

7 أدوات أخرى

- Sonobuoy : أسهل وأسرع من Kube-Test ، واجهته أوضح.
- بس احنا هنا هانكمل مع الـ Native Kubernetes Testing Infra

🟡 الخلاصة

- Manual Validation : جيد كبداية.
- E2E Tests : شامل جدًا.(1000 Test)
- Conformance Tests : الحد الأدنى اللازم عشان الـ Cluster يكون معتمد رسميًا.
- Tools زي Sonobuoy موجودة للتسهيل.

End to End Tests - Run and Analyze

فكرة المحاضرة

التوبيك ده بيركّز على إزاي تشغّل وتفسّر نتائج اختبارات الـ E2E (End-to-End Tests) على الكلاستر بتاعك.

---

1 تثبيت Kubetest

- محتاج يكون عندك Golang متسطب على الـ Master Node
- بعدين تعمل : go get k8s.io/test-infra/kubetest .. ده بينزّل أداة kubetest محليًا.

---

2 تحميل الـ Binaries المطلوبة

- تستخدم الأمر: kubetest extract --extract=<k8s_version>
  - ده بينشئ فولدر اسمه /Kubernetes فيه كل الـ (... ,kube-apiserver ,kubectl) binaries
- تدخل الفولدر : cd kubernetes/

---

3 تشغيل الاختبارات

- لتشغيل كل الـ Tests (حوالي 1000 Test) : kubetest --test --provider=skeleton > result.txt
- هنا :
  - --provider بيحدد مكان الكلاستر (Local / GCP / AWS)
  - لو شغال محلي، تستخدم skeleton

---

4 إعدادات إضافية لو Local Provider

- لازم تعرّف Variables فيها تفاصيل الـ API Server :
  - export KUBE_MASTER_IP=<master_ip>
  - export KUBE_MASTER=<master_url>
- بعد كده تشغّل الاختبارات.

---

5 نوعين من الاختبارات

- Full E2E Tests :
  - حوالي 1000.Test
  - بياخدوا 12 ساعة تقريبًا.
- Conformance Subset Tests أو Features :
  - تستخدم Argument إضافي : --ginkgo.focus=Conformance
  - يشغّل بس الـ (~164 Test) Conformance Tests
  - بياخدوا حوالي ساعة ونص.

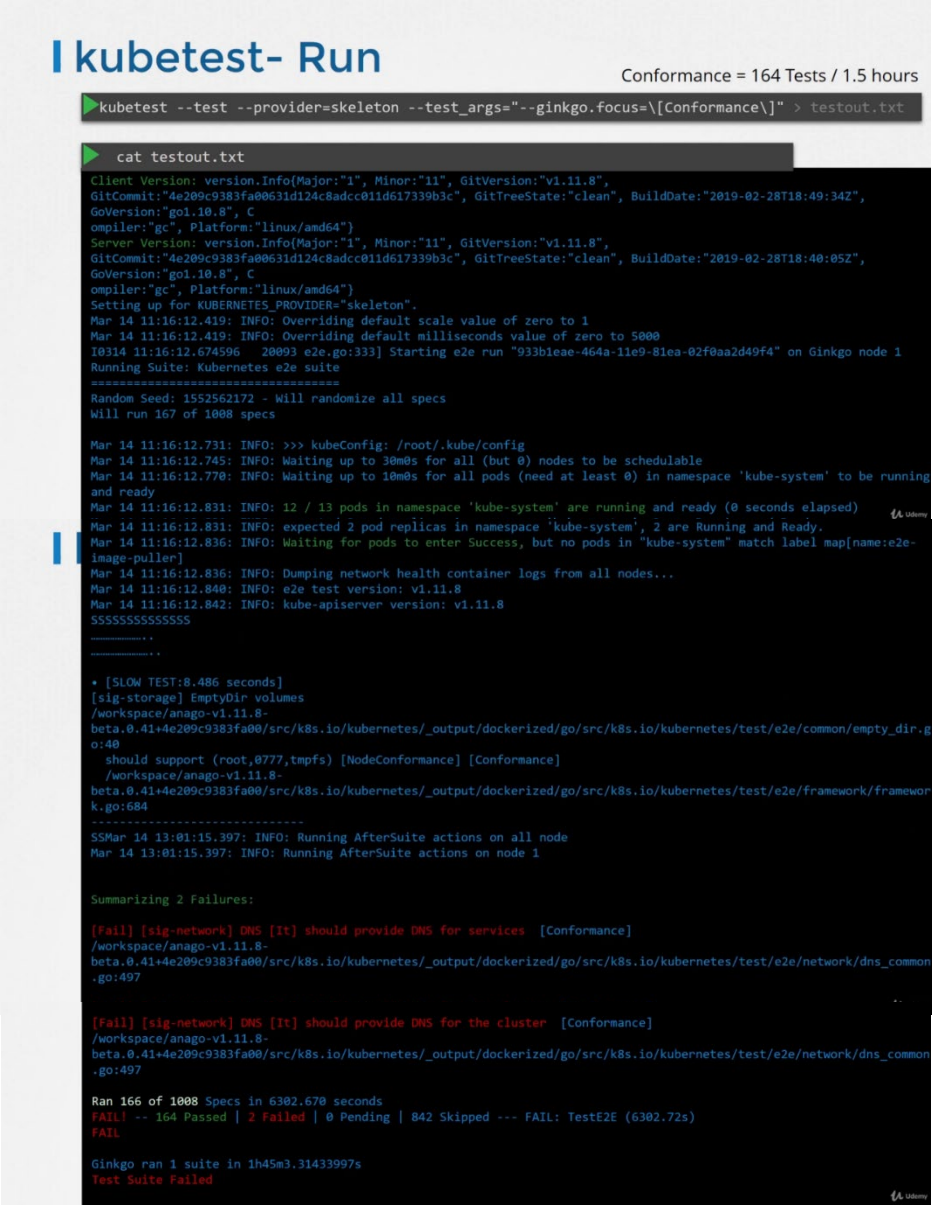
---

6 تفسير النتائج

- أول الاختبارات بسيطة:
  - مقارنة kubectl client مع server version
  - التحقق من حالة الـ Pods
- في الآخر بيظهر Summary :

- Tests run: 166 / 1008
- Passed: 164
- Failed: 2

- المثال هنا : فيه اختبارين فشلوا خاصين بالـ DNS يعني الكلاستر مش بيحل أسماء الـ Services صح



الخلاصة

- الـ Kubetest : : الأداة الرسمية لتشغيل اختبارات E2E
- تقدر تختبر كل حاجة أو تركز على جزء محدد (Conformance)
- النتيجة بتقولك Passed / Failed وتحدد مكان المشكلة.
- أي فشل زي ماحصل في المثال مع الـ DNS معناه إن الكلاستر مش ملتزم تمامًا بالمعايير.

Demo - Run End To End Test using Kubetest

kubernetes-the-hard-way/docs/17-e2e-tests.md at master · mmumshad/kubernetes-the-hard-way

المتطلبات (Requirements)

- مش لازم تشغل الاختبار من الـ Master Node، ممكن أي Linux Server
- يفضل يكون عندك Go (Golang) متسطب . .. لازم يكون عندك RAM + Disk كفاية:
  - Master Node : 2GB على الأقل.
  - Worker Nodes : 1GB أو أكتر لكل واحد && Free Disk Space : 7-10GB

Install Go

- wget https://go.dev/dl/go1.20.5.linux-amd64.tar.gz
- sudo tar -C /usr/local -xzf go1.20.5.linux-amd64.tar.gz
- export PATH=$PATH:/usr/local/go/bin

كدا نزّلنا Go وظبطنا الـ PATH .. ولو حابب تتأكد إنه اتسطب : go version

---

Get Kubetest

نزّل أداة kubetest : go get -u k8s.io/test-infra/kubetest

Extract Kubernetes Binaries

اختار نسخة Kubernetes اللي هتختبرها (مثال v1.13) : kubetest --extract=v1.13

ده هيعمل فولدر اسمه Kubernetes/ فيه كل الـ binaries المطلوبة.

Configure Environment Variables

لازم تعرف kubetest إزاي يتواصل مع الـ API Server :

- export KUBE_MASTER_IP=<Master-or-LB-IP> **and** export KUBE_MASTER=<Master-or-LB-URL>

Run Tests

تشغيل كل الـ E2E Tests (حوالي 1000 - Test) بياخد 12 ساعة : kubetest --test --provider=skeleton

تشغيل Conformance Tests (فقط حوالي 160 - Test) بياخد 1-2 ساعة :

kubetest --test --provider=skeleton --ginkgo.focus="\[Conformance\]"

Viewing Results

- بعد ما يخلص، هتلاقي Summary زي:
  - Ran 166 of 1008 Specs **and** SUCCESS! -- 164 Passed | 2 Failed
- بيوريك إيه اللي Pass و إيه اللي Fail (مثال DNS Test Fail) (زي اللي في التوبيك اللي فات)

الخلاصة

- kubetest بيعمل E2E Validation على Cluster
- Conformance Tests : أهم 160 Test) أساسيين ولازم يعدّوا علشان تبقى Kubernetes Certified)

Smoke Test

kubernetes-the-hard-way/docs/16-smoke-test.md at master · mmumshad/kubernetes-the-hard-way

- مثال عملي لجزء Basic Functionality Testing

1. Health Check (Nodes + Pods)

أول حاجة لازم نتاكد إن الكلاستر شغال كويس.

أوامر : kubectl get pods -A then kubectl get nodes

- get nodes : يوريك حالة كل الـ Nodes (Ready / NotReady)
- get pods -A : يوريك كل البودز في كل الـ namespaces

لو كله ✅ يبقى الكلاستر صحي.

---

2. Test Data Encryption (ETCD)

هنا بنتأكد إن البيانات في الـ ETCD مش متخزنة كـ Plain Text

خطوات:

- نعمل Secret : kubectl create secret generic my-secret --from-literal=mykey=mydata
- نجيب القيمة من داخل etcd (لو عندك access للـ etcd) :

❖ ETCDCTL_API=3 etcdctl get /registry/secrets/default/my-secret \
--endpoints=https://127.0.0.1:2379 \
--cacert=/etc/kubernetes/pki/etcd/ca.crt \
--cert=/etc/kubernetes/pki/etcd/server.crt \
--key=/etc/kubernetes/pki/etcd/server.key

المفروض تلاقي القيمة مش متخزنة نص عادي، لكن متكوّدة / مشفرة.

- بعد التأكد نعمل Cleanup : kubectl delete secret my-secret

---

3. Deploy Nginx

نعمل Deployment بسيط :

❖ kubectl create deployment nginx --image=nginx
❖ kubectl get pods -o wide

ونتاكد إن البود اتعمل وجاهز (Running)

---

4. Expose as NodePort

نفتح الـ Deployment على العالم الخارجي :

❖ kubectl expose deployment nginx --type=NodePort --port=80
❖ kubectl get svc nginx

- هنا هتلاقي Port زي 31518
- ده هو الـ NodePort اللي هتستخدمه للاتصال.

---

5. Test Connectivity

لو عندك IP للـ Worker Node (من kubectl get nodes -o wide): curl http://<worker-node-ip>:31518

❖ المفروض يرد عليك بصفحة Nginx الافتراضية.

6. Logs + Exec

تأكد إنك تقدر:

- تجيب الـ Logs : kubectl logs <nginx-pod-name>
- تدخل جوه البود وتشغل أوامر: kubectl exec -it <nginx-pod-name> -- /bin/bash

---

الخلاصة

الاختبار البسيط ده بيأكد إن :

- الـ Cluster صحي (Nodes + Pods)
- الـ Data مش متخزنة Plain Text في ETCD (Encryption Enabled)
- الـ Deployments بتتعمل.
- الـ Services (NodePort) بتشتغل وتوصلك للتطبيق.
- Logs + Exec شغالين على البود.

Trroubleshooting – Introduction

🔹 **المقدمة :**

هانبدأ قسم جديد خاص بـ Troubleshooting Kubernetes (استكشاف الأخطاء وإصلاحها في الكلاستر)
الفكرة إنك بعد ما عرفت إزاي تبني الكلاستر وتديره، لازم كمان تبقى قادر تشخص وتصلح الأعطال اللي بتحصل سواء في الـ Applications أو في مكونات الكلاستر نفسه.

🛠️ **محتوى القسم:**

1. **Troubleshooting Application Failures**
   - إزاي تعرف ليه التطبيق (Pod/Deployment/Service) مش شغال كويس.
   - هنشوف أدوات زي kubectl describe, kubectl logs و kubectl exec
2. **Troubleshooting Control Plane Failures**
   - لو الـ API Server أو الـ Scheduler أو أي Component من الـ Control Plane وقف.
   - هنتعلم إزاي ندور في الـ systemd services زي
   (systemctl status kubelet, journalctl -u kube-apiserver ...)
3. **Troubleshooting Worker Node Failures**
   - لما الـ Node يبان NotReady أو فيه مشاكل في kubelet أو container runtime
   - هنا بيكون فيه خطوات زي التشييك على kubelet logs أو إن السيرفر نفسه يكون عليه ضغط موارد.
4. **Troubleshooting Network Issues**
   - مشاكل الاتصال بين الـ Pods (تحديدا CNI plugins زي flannel/calico)
   - أو مشاكل Service/DNS resolution

📏 **فكرة القسم:**

- هيكون مليان Hands-on Labs : هيديك "Cluster مكسر" ويطلب منك تصلحه.
- ده بيدربك على السيناريوهات الحقيقية اللي ممكن تقابلها في Production

---

📌 **خلاصة:**

القسم ده مهم جدًا لأنه بيخليك مش بس تعرف تديرKubernetes ، لكن كمان تكون SRE/Administrator قوي يقدر يلاقي المشكلة ويصلحها بسرعة.
وده جزء أساسي من أي شغل في مجال الـ DevOps أو System Administration

Application Failure

◆ المقدمة

- بما إننا خلال الكورس عملنا شوية troubleshooting exercises (استكشاف أخطاء) في سياقات مختلفة.
- دلوقتي هانبص نظرة عامة منظمة عن طرق الـ troubleshooting + كمان practice tests
- البداية هتكون مع أخطاء التطبيقات (Application Failures)

Check Accessibility (مثال (Two-Tier Application))

- Database Pod : بيشغّل قاعدة البيانات، وبيخدم الويب سيرفر من خلال DB Service
- Web Pod : بيشغّل الويب سيرفر، وبيخدم المستخدمين من خلال Web Service

📌 نصيحة : قبل ما تبدأ debugging ، اعمل خريطة أو رسم توضيحي للـ architecture

(العناصر اللي عندك والخطوط اللي بينهم زي الترتيب ده) (user → web service → web pod → db service → db pod)

لأن ده هيسهّل عليك إنك تمشي خطوة بخطوة وتحدد مكان المشكلة.

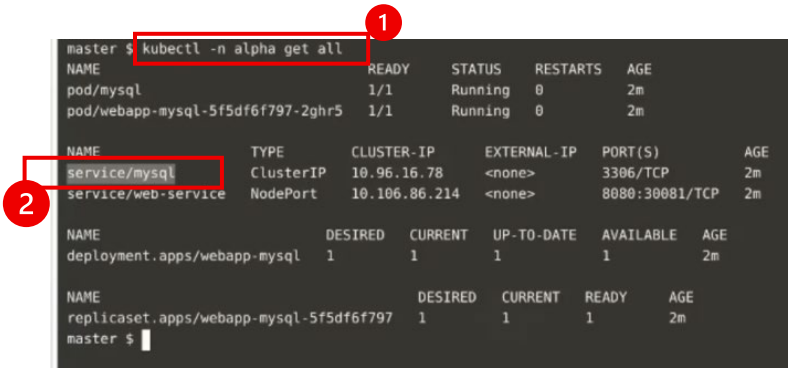
خطوات استكشاف الأخطاء (Troubleshooting Flow)

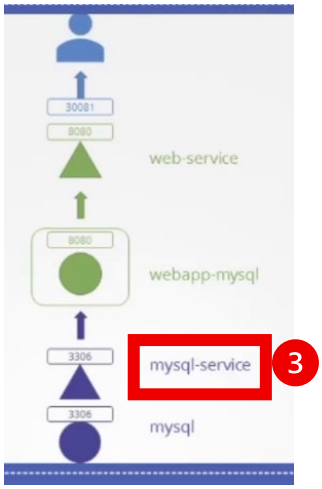
1. ابدأ من الـ Frontend (مثلا لو المستخدم بلغ إن فيه مشكلة في الوصول للتطبيق) :
   - لو Web App : جرّب توصّل على الـ NodePort أو الـ Service باستخدام :

```
curl http://<NodeIP>:<NodePort>
```

2. تشييك على الـ Service :
   - هل اسم الـ service هو هواه نفس اللي في الـ Architecture بتاعي ؟؟!

```
kubectl describe svc web-service
```





✓ طاب ازاي نحل المشكلة دي ؟!

```
master $ kubectl -n alpha get svc mysql -o yaml > mysql-service.yaml
master $ kubectl -n alpha delete svc mysql
service "mysql" deleted
master $ vi mysql-service.yaml
master $ kubectl create -f mysql-service.yaml
service/mysql-service created
```

4

In this Example :

Name space : is “alpha”

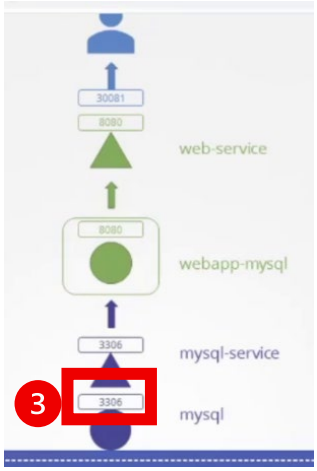
the service : it’s name should be “mysql-service”. But It’s name is “mysql”

وبالتالي كل اللي عملناه اننا نسخنا الـ yaml. بتاع السيرفس القديم لـ yaml. جديد بس بنفس اسم السيرفس الصحيح (بدل ماقعد اعمل ملف الـ yaml بتاع السيرفس الجديدة من الزيرو يعني فبنشغل الشغل :D)

وبعدين مسحنا الـ yaml. القديم بتاع السيرفس القديمة .. ثم دخلنا نعدل اسم السيرفس بقا في الـ yaml. الجديد بتاع السيرفس الجديدة

- هل الـ Service شايفة الـ endpoints (يعني هل السيرفس شايف الـ pods اللي المفروض يوجه لها الترافيك) ؟
لأن لو مفيش Target endpoints هايديك Connection Refused : غالبًا فيه mismatch بين selectors
في الـ service و labels على الـ pod او اختلاف في الـ Target Ports between svc and Architecture

```
master $ kubectl describe svc mysql-service -n beta
Name:                     mysql-service
Namespace:                beta
Labels:                   <none>
Annotations:              <none>
Selector:                 name=mysql
Type:                     ClusterIP
IP:                       10.96.143.161
Port:                     <unset>  3306/TCP
TargetPort:               8080/TCP
Endpoints:                10.32.0.2:8080
Session Affinity:         None
Events:                   <none>
master $
```



✓ طاب ازاي نحل المشكلة دي ؟!

```
master $ kubectl -n beta get svc mysql-service -o yaml > mysql-service.yaml
master $ kubectl -n beta delete  svc mysql-service
service "mysql-service" deleted
master $ vi mysql-service.yaml
master $ kubectl create -f mysql-service.yaml
```

In this Example :
Name space : is "beta"
the service : it's name is "mysql-service". Which is True
But the problem in this service is "TargetPorts is 8080" and In The Architeture, It's "3306"

وبالتالي كل اللي عملناه اننا نسخنا الـ yaml. بتاع السيرفس القديم لـ yaml. جديد بنفس اسم السيرفس عادي (لانه في الحالة دي صح)
وبعدين مسحنا الـ yaml. القديم بتاع السيرفس القديمة .. ثم دخلنا نعدل الـ TargetPorts بقا في الـ yaml. الجديد

```
master $ kubectl -n gamma get ep
NAME            ENDPOINTS          AGE
mysql-service   <none>             1m
web-service     10.32.0.3:8080     1m
master $
```

```
master $ kubectl get svc mysql-service -n gamma -o yaml
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: 2019-10-20T05:42:49Z
  name: mysql-service
  namespace: gamma
  resourceVersion: "2772"
  selfLink: /api/v1/namespaces/gamma/services/mysql-service
  uid: 73a1cb0a-f2fc-11e9-99c9-0242ac110008
spec:
  clusterIP: 10.102.16.99
  ports:
  - port: 3306
    protocol: TCP
    targetPort: 3306
  selector:
    name: sql00001
  sessionAffinity: None
  type: ClusterIP
status:
  loadBalancer: {}
master $
```

```
master $ kubectl -n gamma describe pod mysql
Name:               mysql
Namespace:          gamma
Priority:           0
PriorityClassName:  <none>
Node:               node01/172.17.0.18
Start Time:         Sun, 20 Oct 2019 05:42:49 +0000
Labels:             name=mysql
Annotations:        <none>
Status:             Running
IP:                 10.32.0.2
Containers:
  mysql:
    Container ID:   docker://9d6fa24ed28cc8532df31238ce317c03695b4691cd5eaf2c018f9dea8a41635d
    Image:          mysql
    Image ID:       docker-pullable://mysql@sha256:7345ce4ce6f0c1771d01fa333b8edb2c606ca59d385f69575f8e3e2ec6695eee
    Port:           3306/TCP
    Host Port:      0/TCP
    State:          Running
      Started:      Sun, 20 Oct 2019 05:42:52 +0000
    Ready:          True
    Restart Count:  0
    Environment:
      MYSQL_ROOT_PASSWORD:  paswrd
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-cshmh (ro)
```

✓ طاب ازاي نحل المشكلة دي ؟!

```
master $ kubectl -n gamma  delete  svc mysql-service
service "mysql-service" deleted
master $ kubectl -n gamma expose pod mysql --name=mysql-service
service/mysql-service exposed
master $ kubectl -n gamma get ep
NAME            ENDPOINTS        AGE
mysql-service   10.32.0.2:3306   8s
web-service     10.32.0.3:8080   3m
master $
```

4

In this Example :

Name space : is “gamma”

the problem : It can’t reach the Endpoint of the Pod as shown in figure1

so we check the matches between Labels and Selectors of the svc and pod as shown in figure2,3

وبالتالي كل اللي عملناه اننا مسحنا الـ yaml. بتاع السيرفس القديم .. وبعدين هانستخدم kubectl expose علشان نعمل service جديدة شغالة عالـ pod علطول ## بس هو هنا غشيم شوية لان الطريقة دي مش مستحبة عشان لو الـ pod حصله restart مثلا فالـ service مش هاتقدر تتابعه)

والأفضل اننا نمشي زي الخطوات السابقة .. بمعني اننا نخش علي لـ yaml. بتاع السيرفس ونعدل الـ selector بحيث يكون متوافق مع الـ labels اللي عالـ pod ## دا الحل الأفضل ##

---

3. **تشييك على الـ Pod نفسه :**
   - اتأكد إن حالة مكونات الـ Control Plan وحالة الـ Pods تبقي Running مش **CrashLoopBackOff** أو **Error** :

     kubectl get pods -o wide
   - شوف عدد الـ **restarts** : ده بيديك فكرة لو التطبيق بيقع وبيعمل restart كتير.
   - شوف الـ events : **kubectl describe pod <pod-name>**

---

4. **تشييك الـ Logs :**
   - Logs عادية : **kubectl logs <pod-name>**
   - لو الـ pod بيعمل restart بسرعة ومش قادر تلحق تشوف السبب .. استخدم :

     kubectl logs --previous <pod-name>
   - أو تابع بشكل مباشر : **kubectl logs -f <pod-name>**

```
kubectl get pod
NAME          READY   STATUS    RESTARTS   AGE
Web           1/1     Running   5          50m
```

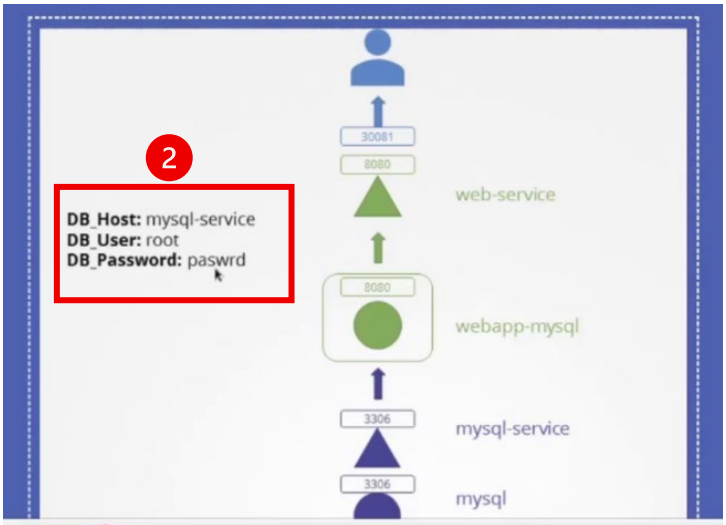
```
kubectl describe pod web
...

Events:
  Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
  Normal  Scheduled  52m   default-scheduler  Successfully assigned webapp-mysql to worker-1
  Normal  Pulling    52m   kubelet, worker-1  pulling image "simple-webapp-mysql"
  Normal  Pulled     52m   kubelet, worker-1  Successfully pulled image "simple-webapp-mysql"
  Normal  Created    52m   kubelet, worker-1  Created container
  Normal  Started    52m   kubelet, worker-1  Started container
```

```
kubectl logs web -f --previous
10.32.0.1 - - [01/Apr/2019 12:51:55] "GET / HTTP/1.1" 200 -
10.32.0.1 - - [01/Apr/2019 12:51:55] "GET /static/img/success.jpg HTTP/1.1" 200 -
10.32.0.1 - - [01/Apr/2019 12:51:55] "GET /favicon.ico HTTP/1.1" 404 -
10.32.0.1 - - [01/Apr/2019 12:51:57] "GET / HTTP/1.1" 200 -
10.32.0.1 - - [01/Apr/2019 12:51:57] "GET / HTTP/1.1" 200 -
10.32.0.1 - - [01/Apr/2019 12:51:58] "GET / HTTP/1.1" 200 -
10.32.0.1 - - [01/Apr/2019 12:51:58] "GET / HTTP/1.1" 200 -
10.32.0.1 - - [01/Apr/2019 12:51:58] "GET / HTTP/1.1" 400 - Some Database Error application exiting!
```

5. لو المشكلة في الـ Database Layer :
   - نفس الخطوات:
     - افحص الـ DB Service (endpoints)
     - افحص الـ DB Pod (status, events)
     - شوف الـ DB logs عشان تشوف errors زي (connection refused, authentication errors)

```
master $ kubectl -n delta describe deployments. webapp-mysql
Name:                   webapp-mysql
Namespace:              delta
CreationTimestamp:      Sun, 20 Oct 2019 05:47:09 +0000
Labels:                 name=webapp-mysql
Annotations:            deployment.kubernetes.io/revision=1
Selector:               name=webapp-mysql
Replicas:               1 desired | 1 updated | 1 total | 1 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  name=webapp-mysql
  Containers:
   webapp-mysql:
    Image:      mmumshad/simple-webapp-mysql
    Port:       8080/TCP
    Host Port:  0/TCP
    Environment:
      DB_Host:      mysql-service
      DB_User:      sql-user
      DB_Password:  paswrd
    Mounts:         <none>
  Volumes:          <none>
Conditions:
  Type           Status  Reason
```

1




```
master $ kubectl -n delta get deployments. webapp-mysql -o yaml > web.yaml
master $ kubectl delete deployments. webapp-mysql -n delta
deployment.extensions "webapp-mysql" deleted
master $
master $
master $
master $ vi web.yaml
master $ kubectl create -f web.yaml
```

3

4

✓ طاب ازاي نحل المشكلة دي ؟!

In this Example :
we specified values of (DB_Host, DB_User, DB_Password) as shown on figure2
the problem : was ACCESS DENIED FOR USER "sql-user'
So we start checking the Deployment to see database. As you can see we realize that "DB_User=sql-user" Not "root" as shown in figure1

وبالتالي كل اللي عملناه هو نفس الطرق السابقة وهو اننا بنعمل yaml جديد من القديم ثم نمسح القديم ونخش نعدل في الجديد

📚 ملاحظات إضافية

- Kubernetes Documentation فيه troubleshooting guide ممتاز لـ Applications
- ده مهم جدًا سواء للـ Practice Tests أو للـ CKA Exam

📌 الخلاصة:

التكنيك الأساسي هو إنك تمشي layer by layer :

1. User access (frontend)
2. Service discovery
3. Pod health & restarts
4. Logs (current + previous)
5. Dependencies (زي الـ DB)

Control Plane Failure

1 أول خطوة: فحص حالة الـ Nodes

- تستخدم : kubectl get nodes

🔎 الهدف : تتأكد إن كل الـ Nodes Ready وصحية.

لو في Node مش Ready يبقى عندك مشكلة في الاتصال أو في الـ kubelet

---

2 فحص حالة الـ Pods في الـ Cluster

- تستخدم : kubectl get pods -A

🔎 الهدف : تشوف كل الـ Pods شغالة كويس.

---

3 التأكد من مكونات الـ Control Plane

❖ فيه سيناريوهين:

📍 لو مستخدم kubeadm :

- مكونات الـ Control Plane بتتثبت كـ Pods داخل الـ namespace : kubectl get pods -n kube-system
  بتشوف :
  - kube-apiserver
  - kube-controller-manager
  - kube-scheduler
  - etcd

📍 لو مكونات الـ Control Plane متسطبة كـ خدمات (Services) على السيرفر مباشرة:

- تتأكد من الخدمات على الماستر:
  - systemctl status kube-apiserver
  - systemctl status kube-controller-manager
  - systemctl status kube-scheduler
- وعلى الـ Worker Nodes تتأكد من :
  - systemctl status kubelet
  - systemctl status kube-proxy

---

4 مراجعة الـ Logs

- لو cluster بـ kubeadm (Pods) : kubectl logs <pod-name> -n kube-system
- لو cluster بخدمات system :
  - journalctl -u kube-apiserver
  - journalctl -u kube-controller-manager
  - journalctl -u kube-scheduler

📌 ده هيساعدك تعرف السبب الأساسي للمشكلة (Config غلط .. أو Port مش مفتوح.. أو Certificate منتهي ... إلخ)

5️⃣ الرجوع للـ Documentation

- دايمًا Kubernetes docs فيها سكاشن troubleshooting : Troubleshooting Clusters | Kubernetes

---

✅ الخلاصة:

- تبدأ من مستوى الـ Nodes
- بعدين الـ Pods أو الـ Services
- بعدين Logs كل component
- وأخيرًا ترجع للـ docs

Demo – Control Plane Failure

🛠️ السيناريو

- عندك Cluster وفيه Application مش شغال (Pods في حالة Pending)
- المطلوب: تعمل Troubleshooting وتصلح مشاكل الـ Control Plane components (Scheduler, Controller Manager, ...).

1️⃣ المشكلة الأولى : الـ Pod مش بيتعمله Schedule

❖ تلاقي الـ Pod في حالة Pending :

- kubectl get pods
- kubectl describe pod <pod-name>

- السبب : kube-scheduler نفسه مش شغال.
- لما تراجع الـ kube-system namespace : kubectl get pods -n kube-system
  هاتلاقي إن kube-scheduler في حالة CrashLoopBackOff
- لما تبص على الـ logs : kubectl logs kube-scheduler-<node-name> -n kube-system
  هاتلاقي إن الـ binary مكتوب غلط (مثلًا مكتوب kube-shedulerrrrrr بدل kube-scheduler)

🔧 الحل:

- تفتح ملف تعريف الـ Pod لأنه static pod في :
  - cat /etc/system/system/kubelet.service.d/10-kubeadm.conf
    the o/p is "—config=/var/lib/kubelet/config.yaml" .. Copy the PATH
  - grep -i staticPodPath /var/lib/kubelet/config.yaml
    the o/p would be the path of (staticPodPath) which is "/etc/kubernetes/manifests"

لاحظ كل الخطوات دي علشان اعرف الـ path of staticpods بس الأفضل تكون حافظ الـ path علطول

وهو دا الـ path الإفتراضي اللي بيتعمل فيه الـ staticPods (اللي هي الـ control plane) بواسطة الـ kubeadm

- هانخش علي /etc/kubernetes/manifests/kube-scheduler.yaml وتصلح اسم الـ binary في section command
- تحفظ الملف : والـ kubelet هيعيد تشغيل الـ pod تلقائيًا.

✅ النتيجة : الـ scheduler يشتغل، والـ Pod يتعمله Schedule على Worker Node

---

2 المشكلة الثانية : Scaling مش شغال

❖ تحاول تعمل Scale للـ Deployment : kubectl scale deploy app --replicas=2
لكن عدد الـ Pods ما بيزيدش.

- السبب هنا : kube-controller-manager معطل (هو المسؤول عن الـ ReplicaSets والـ Deployments)
- تشوف حالته : kubectl get pods -n kube-system
هاتلاقيه في CrashLoop
- تشوف الـ logs : kubectl logs kube-controller-manager-<node-name> -n kube-system
فتكتشف إنه بيحاول يقرأ ملف Config اسمه غلط (مثلاً controller-manager-xxxx.conf
بدل controller-manager.conf)

🔧 الحل:

- تعدل ملف الـ static pod : /etc/kubernetes/manifests/kube-controller-manager.yaml
- تصلح اسم ملف الـ config
- تحفظ وبالتالي الـ pod يتعاد تشغيله.

---

3 مشكلة إضافية : Certificates Path غلط

- بعد شوية، نفس الـ controller-manager يرجع يـ CrashLoop تاني.
- لما تراجع الـlogs ، تلاقي إنه مش لاقي certificates في الـ path
- في ملف الـ (kube-controller-manager.yaml) pod ، الـ volume mount بيتجه لمكان غلط
(مثلاً /etc/kubernetes/wrong-pki بدل /etc/kubernetes/pki)

🔧 الحل:

- تصلح الـ path في الـ volumeMounts للـ certificates
- تحفظ الملف وبالتالي pod يتعاد تشغيله ويشتغل طبيعي.

✅ النتيجة النهائية

- بعد الإصلاح:
  - الـ Scheduler شغال : Pods تتعمل لها Scheduling
  - الـ Controller Manager شغال : Deployments تتعمل لها Scaling
  - الـ Cluster بيرجع يشتغل بشكل طبيعي.

📌 الخلاصة (Exam Tips) :

- أي Control Plane component (API Server, Scheduler, Controller Manager, ETCD) بيتسطب كـ static pod في /etc/kubernetes/manifests
- أي تعديل في ملفات الـ YAML هناك : kubelet بيلاحظ التغيير ويعيد تشغيل الـ Pod أو يعمله recreate
- دايمًا تبدأ troubleshooting بـ:

❖ kubectl get pods -n kube-system
❖ kubectl describe pod <name>
❖ kubectl logs <name>

1. لو مش واضح ايه سبب المشكلة .. فتراجع ملفات الـ manifests

Worker Node Failure

🖥️ البداية : حالة الـ Nodes

- أول حاجة نبدأ نعمل : kubectl get nodes
- لو لقيت Node مش Ready : تبدأ تحقق فيه.

🔍 استخدام describe node

- تشوف التفاصيل : kubectl describe node <node-name>
- هتلاقي عندك Conditions (بتكون True/False/Unknown)
  - OutOfDisk : لو الـ node خلصت مساحة.
  - MemoryPressure : لو فيه ضغط على الـ RAM
  - DiskPressure : لو فيه ضغط على القرص (I/O أو مساحة)
  - PIDPressure : لو فيه عمليات كتيرة جدًا (processes)
  - Ready : لو الـ node سليم.

✅ ده بيديك فكرة سريعة إيه المشكلة الأساسية.

⚠️ حالة Unknown

- لو الحالة Unknown : غالبًا الـ Node فقد الاتصال بالـ Master (يعني crashed أو network issue)
- تراجع آخر Heartbeat Time : kubectl describe node <node-name> | grep "LastHeartbeatTime"
  ده يوريك إمتى حصل آخر اتصال.

🛠️ تحقق من حالة الـ Node نفسه

- تدخل على السيرفر وتشوف:
  - هل الجهاز Online ولا Crashed
  - موارد النظام :
  - top : للـ CPU والـ Memory
  - df -h : للـ Disk usage

⚙️ kubelet Service

- أهم خدمة في Worker Node : kubelet
- تتحقق من حالته: systemctl status kubelet
- لو مش شغال فتشغّله : systemctl enable kubelet **Then** systemctl start kubelet

---

📜 Logs

- تشوف الـ logs بتاعت kubelet : journalctl -u kubelet -f
- هنا ممكن تكتشف مشاكل زي:
  - Certificates expired
  - غلط في config
  - Node مش قادر يكلم الـ API Server

Certificates 🔑

- الـ kubelet بيستخدم Certificates عشان يتواصل مع الـ Control Plane
- لازم تتحقق إنها:
  - مش منتهية الصلاحية :
    - ❖ openssl x509 -in /var/lib/kubelet/pki/kubelet-client-current.pem -noout -dates
  - صادرة من الـ CA الصحيح.
  - معمولة للـ Group المناسب (system:nodes)

✅ الخلاصة

عشان تعمل Troubleshoot للـ Worker Node :

1. kubectl get nodes : شوف Ready/NotReady
2. kubectl describe node : راقب Conditions
3. لو Unknown : بص على Heartbeat
4. افحص الجهاز نفسه (CPU, Memory, Disk)
5. افحص kubelet service : status + logs
6. راجع Certificates

Demo – Worker Node Failure

🟢 بداية اللاب

- في الـ Worker Nodes بيهمنا خدمتين أساسيين:
  - kubelet
  - kube-proxy

مشكلة الـ Certification

❖ هتلاقي ان (worker = NotReady) فنبدأ نعمل الـ Troubleshooting :

1. نعمل : kubectl describe node <worker-node>
   - ما طلعش سبب واضح.
2. نروح على الـ worker ونراجع الـ kubelet status :
   systemctl status kubelet
   - الخدمة شغالة، بس في logs فيه Error
3. نشوف الـ logs : journalctl -u kubelet -f
   - الخطأ Unable to load client cert : يعني الملف مش موجود.
4. ندور على ملف config :
   - عادة بيكون في :
     - /etc/systemd/system/kubelet.service.d/10-kubeadm.conf
     - /var/lib/kubelet/config.yaml
   - نلاقي إن path للـ cert مكتوب غلط.
5. نصححه : nano /var/lib/kubelet/config.yaml بحيث نخلي المسار يروح للـ: /etc/kubernetes/pki/

> لاحظ اننا بنعرف الـ path الصحيح بتاع الـ cert من خلال :
>
> - ❖ Cd /etc/kubernetes
> - ❖ ls which giving kubelet.conf & pki
> - ❖ cd pki/
> - ❖ ls which giving ca.crt and here it's.
> - ❖ pwd which giving the full path of cert. (/etc/kubernetes/pki)

6. نعمل reload + restart :

- systemctl daemon-reload
- systemctl restart kubelet

✅ وبعدها الـ node بيرجع Ready

مشكلة الـ Connection Refused بسبب الـ Port

❖ تاني الـ worker بيرجع NotReady

1. نراجع kubelet logs :
   - نلاقي Error : connection refused على Port 6553
   - لكن الـ API Server شغال أصلاً على Port 6443 (الـ default) : ممكن تعرف من kubectl cluster-info
2. نفتح config بتاع kubelet :
   - ونلاقي إن مكتوب في --api-server endpoint : https://172.17.0.22:6553
   - وده غلط.
3. نصححه : https://172.17.0.22:6443
4. نعمل reload + restart زي قبل :

- systemctl daemon-reload
- systemctl restart kubelet

5. نرجع نعمل : kubectl get nodes

✅ نلاقي الـ node Ready

📌 الخلاصة

عشان تعمل Troubleshoot للـ Worker Node :

1. لو kubelet Dead : Restart service
2. لو فيه Error في cert path : صححه في /etc/kubernetes/pki
3. لو API Server Port غلط : صححه للـ 6443.

Kubectl with JSONPath Queries

1. Pre-Requisites - JSON PATH.html

◆ المقدمة

هانتكلم عن استخدام JSONPath مع kubectl (أداة kubectl control utility)
ده بيساعدك إنك تعرض وتفلتر البيانات اللي بترجع من الـ API Server بشكل مخصص بدل ما تقرأ كل التفاصيل الكبيرة اللي بتكون في الـ JSON raw output

---

🔍 ليه JSONPath مهم؟

- في بيئات Production ممكن يبقى عندك مئات الـ Nodes وآلاف الـ Objects (Pods, Deployments, Services, Secrets ...)
- الأوامر العادية زي: kubectl get nodes بتديك معلومات مختصرة (مختارة عشان تكون readable).
- لو عايز تفاصيل أكتر : ممكن تعمل: kubectl get nodes -o wide بس برضه مش بيعرض كل حاجة.
- أحيانًا محتاج:
  - تعمل report صغير : مثلاً اسم كل Node وعدد الـ CPU بتاعه
  - أو تعرض معلومات عن الـ images اللي مستخدمها الـ Pods
  - أو تشوف labels/annotations معينة.
- هنا بييجي دور الـ JSONPath : تقدر تفلتر وتعرض أي جزء من الـ JSON raw data اللي بيرجع من الـ API

🏗 الفكرة العامة

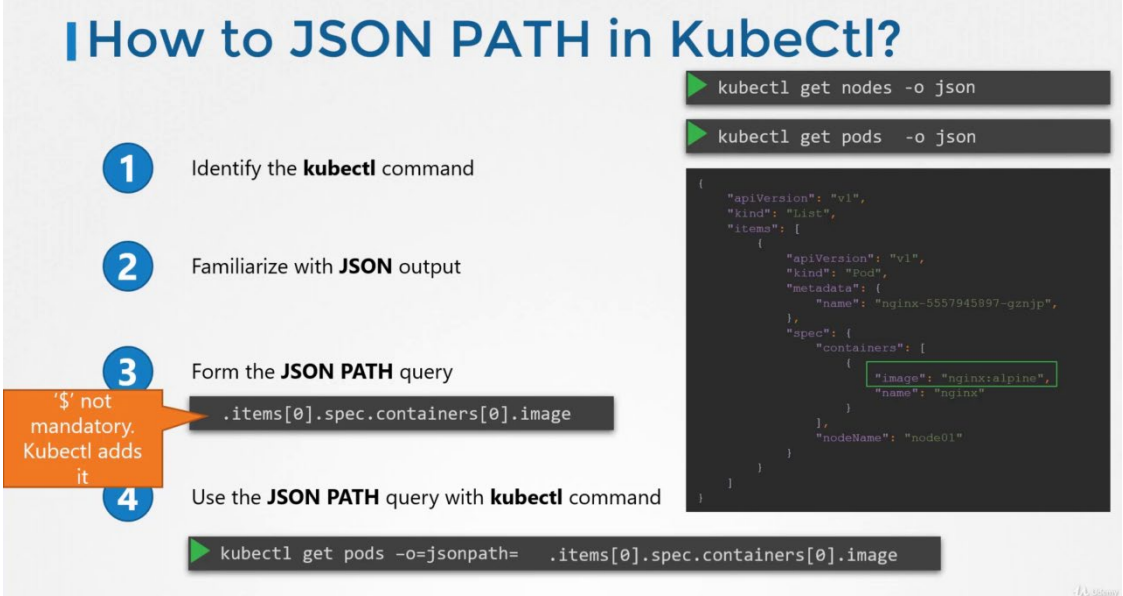
- kubectl لما بيكلم الـ API Server بيرجع بيانات بـ JSON
- الـ output اللي بيظهرلك على الشاشة هو نسخة مختصرة ومتنضفة.
- لو عايز تفاصيل أكتر هاتستخدم: kubectl get nodes -o json
- من هنا تقدر تكتب JSONPath Queries علشان تجيب بالظبط الفيلدز اللي محتاجها.

مثلاً : عايز تجيب أسماء الـ Nodes + عدد الـ CPU عند كل Node : ده اللي هنتعلمه.

🛠 خطوات استخدام JSONPath مع Kubectl

- لو عايز معلومات عن الـ Nodes : kubectl get nodes -o json
- لو عايز معلومات عن الـ Pods : kubectl get pods -o json

🚁 ده بيوريك كل التفاصيل زي ما الـ API Server بيرجعها.



2 افحص الـ JSON Structure

- بعد ما تجيب الـ output بـ -o json ، بص على التركيب (items → metadata → status ...)
- افهم بنية الـ JSON :
  - شوف الـ keys والـ objects جوه الـ JSON عشان تحدد الـ path.
  - مثال:

items[0].metadata.name

items[0].status.capacity.cpu

---

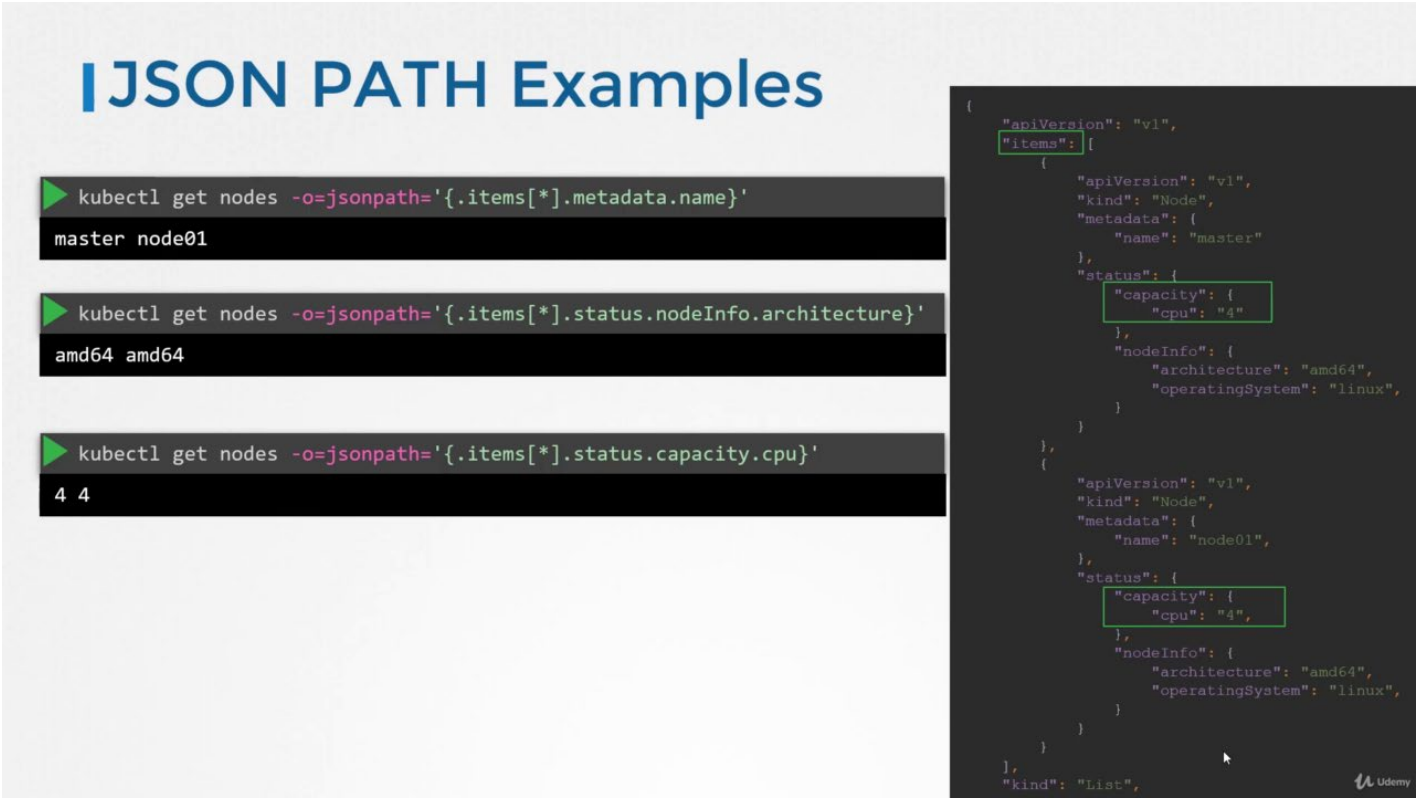
3 اكتب JSONPath Query

- تجيب أسماء الـ Nodes :

❖ kubectl get nodes -o jsonpath='{.items[*].metadata.name}'

- تجيب CPU Capacity :

❖ kubectl get nodes -o jsonpath='{.items[*].status.capacity.cpu}'



🌀 Loops مع JSONPath

- تقدر تعمل loop على كل item جوه الـ list باستخدام:
  - {range .items[*]} ... {end}
- وتستخدم:
  - \t → Tab
  - \n → New line

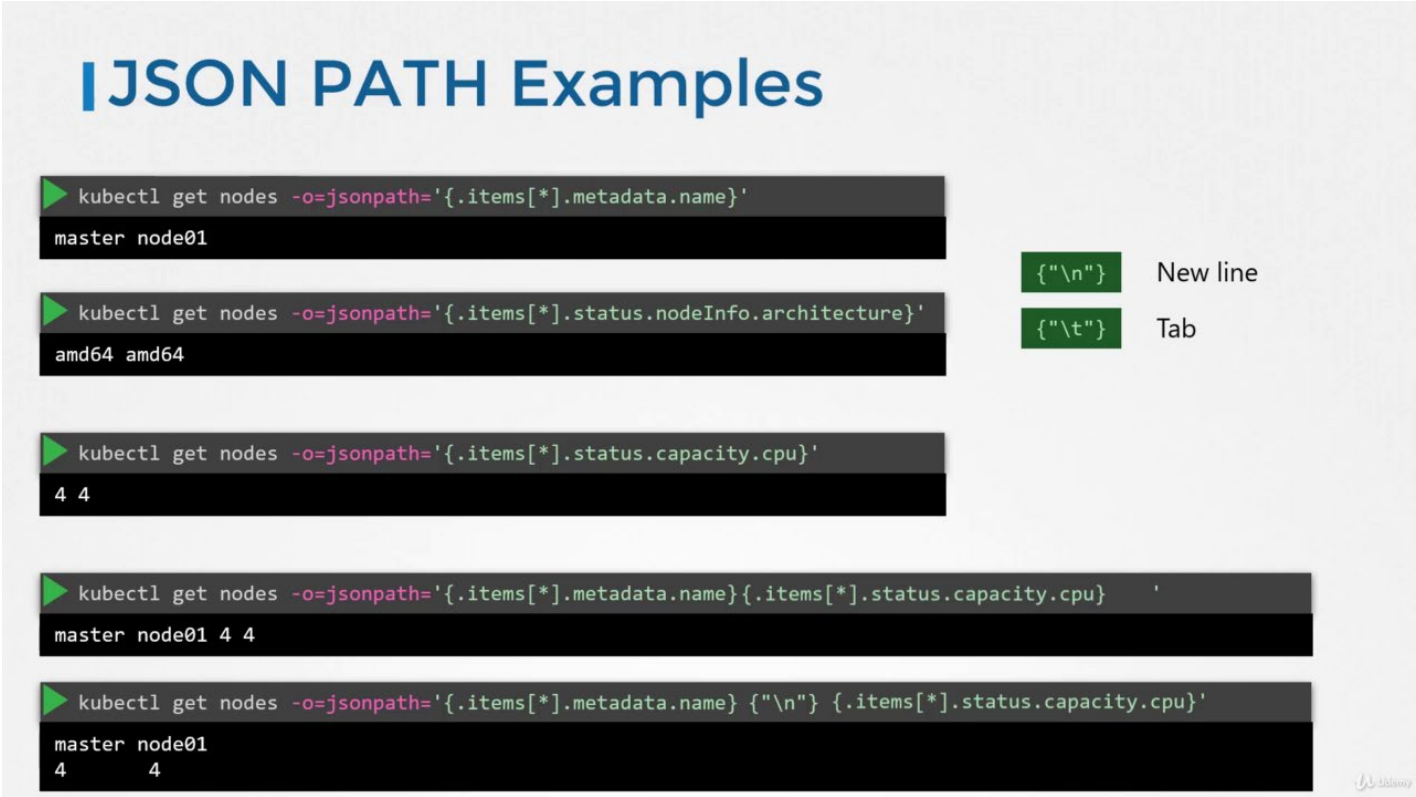
  عشان تنسق الـ output

مثال:

```
kubectl get nodes -o jsonpath='{range.items[*]}{.metadata.name}{"\t"}{.status.capacity.cpu}{"\n"}{end}'
```

📌 الناتج هيبقى بالشكل ده:

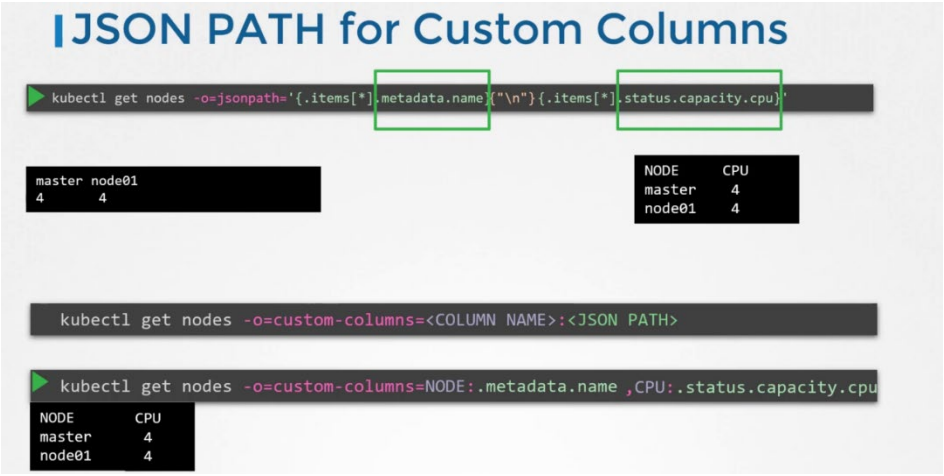
```
node1   4
node2   8
```



📄 Custom Columns

لو مش عايز تكتب JSONPath manually ، ممكن تستخدم : --custom-columns .. مثال :

```
kubectl get nodes -o custom-columns=NAME:.metadata.name,CPU:.status.capacity.cpu
```



5️⃣ Sorting

ممكن كمان تعمل sort حسب قيمة معينة باستخدام --sort-by مع JSONPath :

- kubectl get nodes --sort-by=.metadata.name
- kubectl get pods --sort-by=.status.startTime

📚 ملاحظات إضافية

- جرب Queries الأول على مواقع زي jsonpath.com عشان تتأكد إنها شغالة.
- في الامتحان (CKA/CKAD) هيجيلك Tasks زي" *بطبع أسماء كل الـ Pods اللي في namespace معين مع الـ Node اللي شغالين عليه* " : هنا JSONPath أو Custom Columns هيوفروا وقت رهيب.
- لازم تحط الـ query بين ' ' single quotes عشان الـ shell ما يلغبطش الأقواس.

📌 الخلاصة

التكنيك العملي لاستخدام JSONPath مع Kubectl :

1. حدد الكوماند اللي يجيب الـ Data (kubectl get ... -o json)
2. افحص تركيب الـ JSON.
3. جرّب JSONPath Query تجيب الفيلدز المطلوبة.
4. استخدم range + \t + \n لو عايز جدول.
5. أو استخدم -o custom-columns للوضوح.
6. استخدم --sort-by للترتيب.

Readiness Probe

🔹 المقدمة

- Kubernetes بيوفرلك آليات تقدر تتابع بيها حالة الـ Pods
- هانركز هنا على الـ Readiness Probe : إزاي النظام يعرف إن التطبيق جوه الـ Pod جاهز يستقبل ترافيك ولا لسه.
- ده مهم علشان الـ Service مايبعتش ترافيك لبرنامج لسه بيعمل startup.

📋 ملخص دورة حياة الـ Pod

- Pending : لسه بيتحدد مكان الـ Pod على أي Node
- ContainerCreating : بيبدأ يسحب الـ Image ويشغل الـ Containers
- Running : كل الـ Containers اشتغلت والتطبيق شغال.
- Terminated/Completed : التطبيق خلص أو الـ Pod اتقفل.

📌 تشيك الحالة:

❖ kubectl get pods

❖ kubectl describe pod <pod-name>

📌 مشكلة الـ Readiness

- بشكل افتراضي، Kubernetes بيفترض إن أي Container بمجرد ما يشتغل يبقى Ready
- لكن ده مش صح دايمًا:
  - Script بسيط ممكن يشتغل فورًا.
  - Database ممكن ياخد ثواني.
  - Web Server : زي (Jenkins ممكن ياخد 15 ثانية أو أكتر عشان يبقى جاهز)
- النتيجة ؟ الـ Service يوجّه ترافيك للـ Pod اللي لسه مش جاهز وبالتالي الـ Users يشوفوا Errors

---

🛠️ الحل Readiness Probe :

❖ تقدر تقول لـ Kubernetes إزاي يعرف إن التطبيق جوه الـ Pod بقى جاهز.

الخيارات:

1. HTTP Probe : يعمل request لـ API/Endpoint ويتأكد إن في response
2. TCP Probe : يتأكد إن فيه socket مفتوح على Port معين.
3. Exec Probe : يشغل أمر (script/command) جوه الـ Container

⚙️ إعدادات إضافية

- initialDelaySeconds : وقت تأخير قبل ما يبدأ الفحص (مثلاً 10s)
- periodSeconds : كل كام ثانية يعمل probe (default = 10s)
- failureThreshold : كام محاولة فاشلة قبل ما يعتبر الـ Pod مش جاهز.

🌀 سيناريو متعدد الـ Pods

- عندك Deployment بـ 3 Pods
- لو ضفت Pod جديد بياخد دقيقة عشان يجهز:
  - ❌ بدون Readiness : الـ Service هيوجه ترافيك له فورًا وبالتالي الـ Users يتأثروا.
  - ✅ مع Readiness : الـ Service يستنى الـ Pod الجديد يبقى جاهز وبعدين يضيفه للترافيك وبالتالي مفيش انقطاع.

🟩 ملاحظات إضافية

- الـ Readiness Probes بتركّز على جاهزية استلام الترافيك مش مجرد إن الـ Pod شغال.
- في Documentation الخاصة بـ Kubernetes فيه أمثلة متقدمة وتفاصيل عن كل الإعدادات.

📌 الخلاصة

- الـ Pod ممكن يبان "Running" لكن التطبيق لسه مش جاهز.
- الـ Readiness Probe بتربط الحالة الفعلية للتطبيق بالـ Pod readiness
- تقدر تختبر باستخدام HTTP, TCP أو Exec
- إعدادات زي delay, period, threshold بتديك تحكم كامل.
- مهمة جدًا خاصة في Deployments & ReplicaSets عشان تمنع downtime للمستخدمين.

Liveness Probes

🎯 الموضوع الأساسي

Liveness Probes في Kubernetes

إزاي تتأكد إن التطبيق جوه الـ Container لسه شغال بشكل صحي، مش مجرد الـ Container واقف على رجليه

◆ المقدمة

- **Docker لوحده مش orchestrator : يعني لو الـ Container وقع بيقف ومايرجعش غير لما تعمله restart يدوي.**
- **Kubernetes مختلف : يعني بيراقب الـ Containers ولو وقع بيعمله restart أو يعيد إنشاءه.**

📌 **لكن في مشكلة :**

- **أحيانًا الـ Container يفضل شغال (Running) بس التطبيق جوه معلق**
  (مثلاً infinite loop : من برّه باين إنه "سليم" بس فعليًا بيبطل يرد على المستخدمين.
- **هنا ييجي دور Liveness Probe**

🛠️ إيه هي الـ Liveness Probe ؟

- **هي اختبار (Probe) بيتعمل بشكل دوري للتأكد إن التطبيق جوه الـ Container لسه صحي.**
- **لو فشل : Kubernetes بيعتبر الـ Container ميت وبالتالي يقتله ويشغل واحد جديد مكانه.**

📋 إزاي بتتعرّف الصحة (Health) ؟

إنت كـ Developer بتحدد:

1. **HTTP Check : مثلاً API لازم يرد.**
2. **TCP Socket : يتأكد إن port مفتوح (مثلاً DB listening)**
3. **Exec Command : يشغل command/script جوه الـ Container**

⚙️ إعدادات إضافية

- **initialDelaySeconds : وقت تأخير قبل أول اختبار.**
- **periodSeconds : كل كام ثانية يعمل probe (default = 10s)**
- **successThreshold : كام مرة لازم ينجح عشان يتعتبر Healthy**
- **failureThreshold : كام مرة يفشل قبل ما يتقتل الـ Container**

الفرق بين Readiness و Liveness

- **Readiness Probe : يحدد هل الـ Pod جاهز يستقبل ترافيك ولا لأ . لو مش جاهز فالـ Service مايبعتلوش ترافيك**
- **Liveness Probe : يحدد هل التطبيق لسه صحي . لو مش صحي فالـ Kubernetes هايقتله ويجيب واحد جديد**

الخلاصة

- **الـ Container ممكن يبان شغال لكن التطبيق جوه معلق.**
- **Liveness Probes بتضمن إن Kubernetes يراقب صحة التطبيق فعليًا مش بس حالة الـ Container**
- **تقدر تختبر باستخدام HTTP / TCP / Exec**
- **إعدادات زي initialDelay, periodSeconds, failureThreshold بتديك تحكم كامل.**

Jobs

🎯 الموضوع الأساسي

Jobs في Kubernetes : تشغيل Batch Workloads (مهام بتنتهي بعد ما تخلص) زي:

- معالجة بيانات.
- توليد تقرير.
- إرسال إيميل.
- تشغيل Script صغير.

◆ الفرق بين الـ Workloads

- Web Apps / Databases :
  - شغالة دايمًا (Long Running)
  - مش بتنتهي إلا لو عملت لها Stop
- Batch Jobs :
  - بتعمل مهمة معينة وتخلص.
  - مش محتاجة تفضل شغالة على طول.

◆ نفس الفكرة فيDocker

- تعمل Container يشغل Script (مثلاً يجمع رقمين)
- بعد التنفيذ → يطبع النتيجة وبعدين الـ Container يعمل Exit
- لو شغلت: docker ps -a
  - هتلاقيه في حالة Exited + Exit Code = 0 .. يعني نجح

◆ المشكلة في Kubernetes

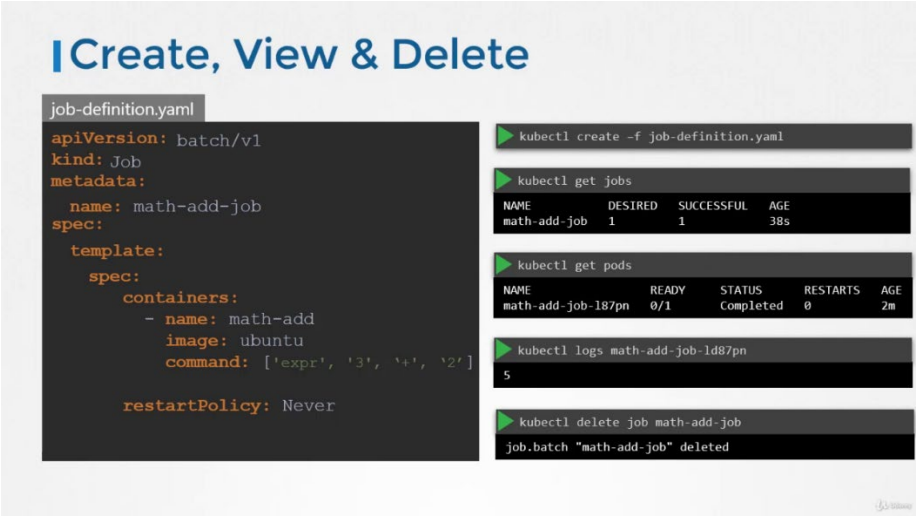
- لو عملت نفس الكلام بـ Pod عادي:
  - الـ Pod يعمل المهمة وبعدين يعمل Exit
  - لكن Kubernetes عنده restartPolicy = Always : بيرجع يشغل الـ Pod تاني.
  - والعملية تفضل تتكرر.

✅ الحل : تغير الـ restartPolicy لـ :

- Never : ميعدش تشغيل الـ Pod
- OnFailure : يشغله بس لو فشل.

◆ الحل المناسب Job :

- ReplicaSet : بيضمن عدد Pods دايمًا شغال.
- Job : بيضمن إن Pods تكمل المهمة للنهاية.
- تقدر تحدد :
  - عدد مرات التكرار(completions)
  - عدد البودز اللي تشتغل في نفس الوقت (parallelism)



◆ إنشاء Job

1. تبدأ من Pod YAML عادي.
2. تغير الـ Kind → Job
3. تضيف apiVersion : batch/v1
4. تحط Pod Spec جوة template
5. تنشئه : kubectl create -f job.yaml

- تشوف الـ Jobs : kubectl get jobs
- تشوف الـ Pods : kubectl get pods

هتلاقيهم في حالة Completed

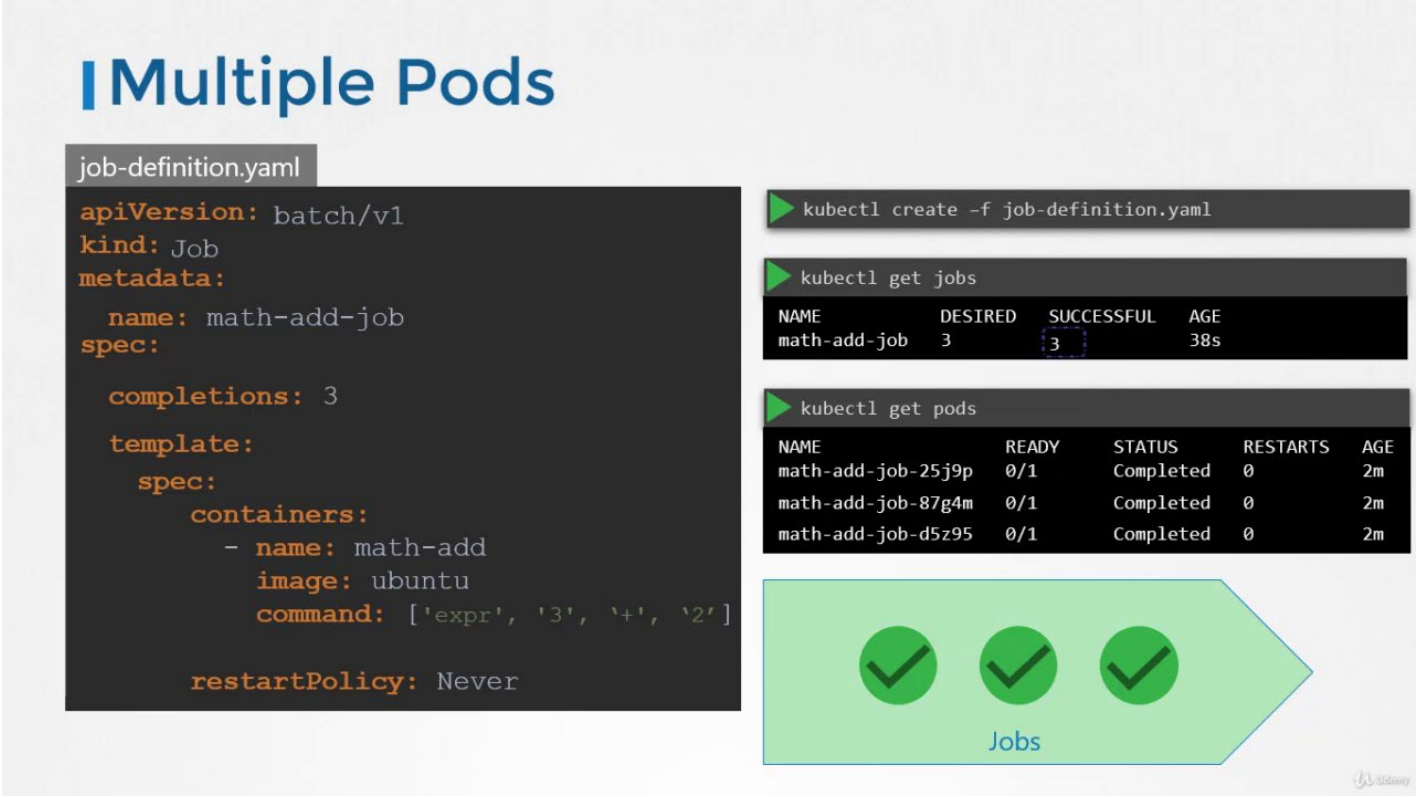
- تشوف النتيجة (stdout) : kubectl logs <pod-name>
- تمسح الـ Job : kubectl delete job <job-name>

وده يمسح الـ Pods اللي عملها كمان

◆ أمثلة عملية

❖ completions: 3

○ يعني لازم 3 Pods يكملوا بنجاح.



- ❖ **parallelism: 3**
  - يشغل **3 Pods** مرة واحدة في نفس الوقت.

📌 والذكاء في الموضوع إنك لو كنت عايز **3 Completions** و **2 Pods** ينجحوا من أول **3** محاولات ..
فهايفضل يعمل **Run** لغاية ما يحقق العدد المطلوب.

📌 الخلاصة

- **Pod** لوحده مش مناسب للـ **Batch Tasks** (هيفضل يعيد التشغيل)
- **Job** = أفضل حل:
  - يـ **Run Pod(s)**
  - يتأكد إنهم خلصوا المهمة.
  - يتوقف.
- ينفع تحدد:
  - **restartPolicy**
  - **completions**
  - **parallelism**

CronJobs

الموضوع الأساسي

CronJobs في Kubernetes : هي زي الـ Jobالعادي بس بيشتغل بشكل مجدول (Scheduled) زي crontab في Linux

---

🔹 الفكرة

- **Job : بينفّذ مرة واحدة بس (Instant)**
- **CronJob : بينفذ بجدول زمني (كل دقيقة، كل ساعة، يوميًا ... إلخ)**

📌 **مثال:**

لو عندك Job بيولّد تقرير ويبعت إيميل.

- **بـ Job عادي : هيتنفذ مرة واحدة.**
- **بـ CronJob : تقدر تخليه يتنفذ يوميًا الساعة 9 صباحًا.**

🔹 طريقة إنشاء CronJob

1. **تبدأ بـ Template فاضي :**
   - **apiVersion: batch/v1beta1 : في الإصدارات القديمة، حاليًا بقت (batch/v1)**
   - **kind: CronJob**
   - **metadata: name: reporting-cron-job**
2. **تحت spec :**
   - **تضيف schedule بصيغة cron expression**
     مثلا : schedule: "0 9 * * *" .. يعني كل يوم الساعة 9:00.
   - **جوة jobTemplate تحط تعريف الـ Job (اللي هو بدوره فيه template للـ Pod)**

```
# ┌──────────── minute (0 - 59)
# │ ┌──────────── hour (0 - 23)
# │ │ ┌──────────── day of the month (1 - 31)
# │ │ │ ┌──────────── month (1 - 12)
# │ │ │ │ ┌──────────── day of the week (0 - 6) (Sunday to Saturday;
# │ │ │ │ │                                   7 is also Sunday on some systems)
# │ │ │ │ │
# * * * * * command to execute
```

⚠️ **خليك واخد بالك:**

في CronJob فيه 3 مستويات من spec :

- **spec بتاعة الـ CronJob**
- **Spec بتاعة الـ Job**
- **Spec بتاعة الـ Pod**

🔹 أوامر التعامل مع CronJobs

- إنشاء CronJob : kubectl create -f cronjob.yaml
- عرض الـ CronJobs : kubectl get cronjob

📌 Kubernetesأوتوماتيك هيعمل Jobs من الـ CronJob ، والـ Jobs هتعمل Pods

---

📌 الخلاصة

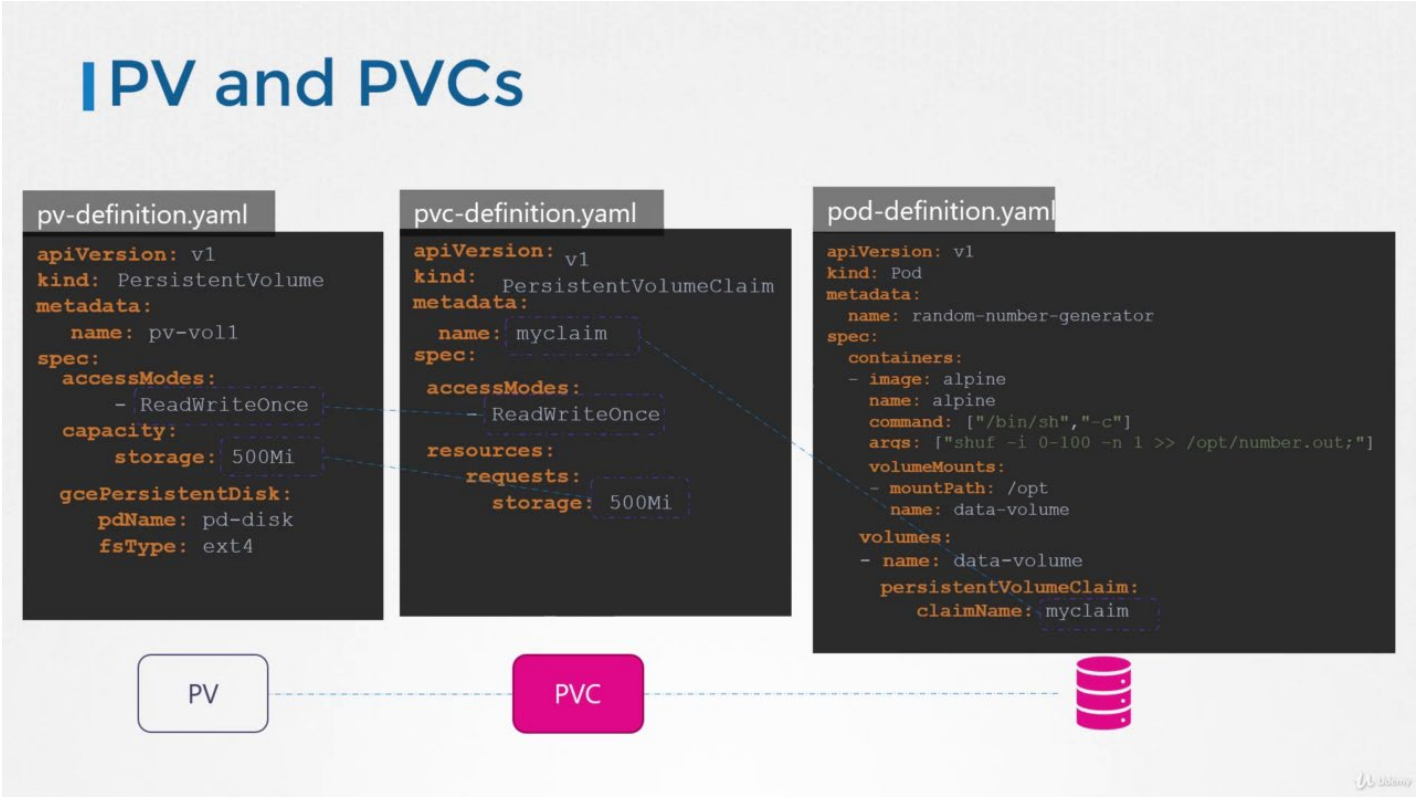
- Job = مهمة لمرة واحدة.
- CronJob = Job بس بمواعيد مجدولة.
- مفيد جدًا في :
  - تقارير يومية و الـ Tasks الدورية.
  - Backups

📌 لازم تاخد بالك من تعدد مستويات specعشان الـ YAML يطلع صحيح

Storage Classes

🎯 الموضوع الأساسي

Storage Classes في Kubernetes : بتخلّي عملية إنشاء الـ Persistent Volumes (PVs) أوتوماتيكية بدل ما تعملها يدوي.



◆ المشكلة قبل StorageClass

- **في البداية كنا بنعمل :**
  1. **نجهّز الـ Persistent Disk (مثلاً على Google Cloud يدويًا)**
  2. **نكتب ملف PV ونربطه بالـ Disk ده.**
  3. **نعمل PVC (PersistentVolumeClaim) ونربطه بالـ PV**

📌 **ده اسمه Static Provisioning (توفير يدوي)**

المشكلة : كل مرة محتاجين تخزين جديد .. لازم لازم نكرر الخطوات يدويًا.

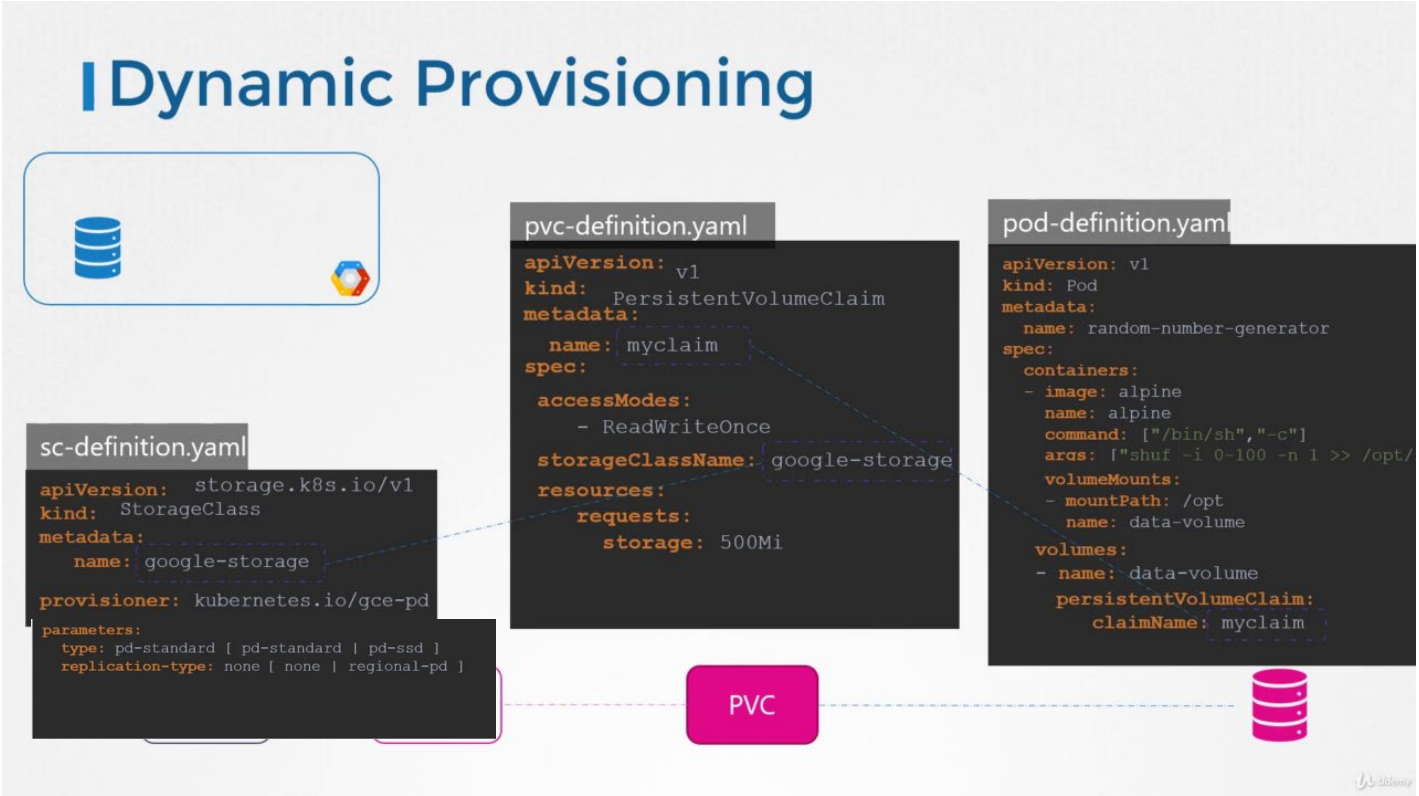
◆ الحل StorageClass

- **مع StorageClass :**
  - **بتحدد نوع الـ Provisioner (زي... Google Cloud, AWS EBS, Azure Disk, NFS)**
  - **أول ما يتعمل PVC : Kubernetes أوتوماتيك ينشئ PV و Disk جديد بناءً على الـ StorageClass**

📌 **ده اسمه Dynamic Provisioning**

◆ خطوات الاستخدام

1. أنشئ StorageClass :
   - Provisioner : بيحدد نوع التخزين (هنا Google Persistent Disk)
   - Parameters : تحدد مواصفات إضافية.(SSD/Standard, Replication...)
2. في الـ PVC :
   - storageClassName : بيربط الـ PVC بالـ StorageClass
3. Kubernetes أوتوماتيك يعمل:
   - Disk جديد.
   - PV جديد.
   - يربط الـ PVC بالـ PV



◆ Storage Classes متعددة (Service Tiers)

- ممكن تعمل أكتر من StorageClass على حسب مستوى الخدمة:
  - Silver → Standard Disks
  - Gold → SSD Disks
  - Platinum → SSD + Replication

📌 وبالتالي المطور يحدد ببساطة أي Class يختار في الـ PVC بدل ما يدوّر في التفاصيل التقنية للتخزين.

📌 الخلاصة

- Static Provisioning : لازم تعمل الـ Disk والـ PV يدويًا.
- Dynamic Provisioning (StorageClass) : كل حاجة أوتوماتيك.
- StorageClass بتسهل إدارة أنواع وأصناف مختلفة من التخزين.

StatefulSets

🎯 الموضوع الأساسي

StatefulSets في Kubernetes : هي طريقة لإدارة الـ Pods اللي محتاجة ترتيب ثابت + هوية ثابتة (زي قواعد البيانات)

🔹 ليه محتاج StatefulSet ؟

- مع الـ Deployments :
  - الـ Pods بتطلع بأسماء عشوائية.
  - كلها بتبدأ في نفس الوقت وبالتالي مفيش ترتيب معين.
  - لو Pod حصله Crash واتعمله إعادة تشغيل .. هايطلع باسم جديد.

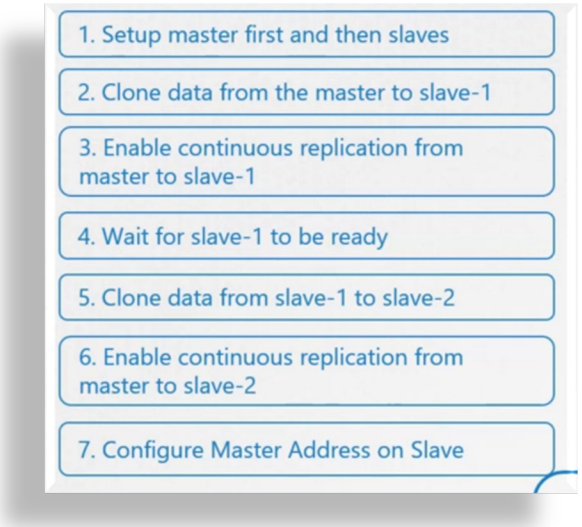
📌 ده ينفع مع Web Servers أو Apps عادية، لكن مش مع قواعد البيانات اللي محتاجة ترتيب وهوية ثابتة.

🔹 مثال MySQL Cluster (Master - Slaves) :

1. سيرفر واحد : شغلنا MySQL واشتغل.
2. عايزين High Availability : أضفنا سيرفرات زيادة كـ Slaves
3. المطلوب:
   - Master يتعمله Setup الأول .
   - Slave1 يـعمل clone من Master
   - Slave2 يـعمل clone من Slave1 وبعدين يرتبط بالـ Master
   - كل Slave لازم يعرف العنوان الثابت للـ Master

📌 مع Deployments مش هانقدر نضمن ده:

- مفيش ترتيب (يعني ممكن Slave يطلع قبل Master)
- أسماء الـ Pods بتتغير لو عملنا Restart



🔹 الحل StatefulSets

- زي الـ Deployment لكن بميزات إضافية:

1. الترتيب (Ordered Deployment)
   - StatefulSet يطلّع الـ Pods واحدة واحدة.
   - Pod0 لازم يبقى Ready قبل ما Pod1 يطلع، وهكذا.
   - ده يضمن : Master الأول .. بعدين Slave1 .. بعدين Slave2
2. هوية ثابتة (Stable Identity)
   - كل Pod بياخد اسم ثابت:
     - mysql-0 (Master)
     - mysql-1 (Slave1)
     - mysql-2 (Slave2)
   - لو Pod وقع ورجع .. بيرجع بنفس الاسم.
3. Sticky Identity
   - دايمًا mysql-0 هو الـ Master
   - Slaves يعرفوا العنوان ده ويعملوا Replication عليه.
   - حتى لو Master وقع ورجع .. يفضل اسمه mysql-0

🔹 مثال عملي (أسماء الـ Pods)

- StatefulSet باسم mysql :
  - Pod0 → mysql-0 (Master)
  - Pod1 → mysql-1 (Slave1 ينسخ من Master)
  - Pod2 → mysql-2 (Slave2 ينسخ من Slave1 ويربط بالـ Master)
- لو زودنا Pod جديد (Scale up) :
  - Pod3 → mysql-3 (هايعرف ينسخ من mysql-2)

📌 الخلاصة

- Deployment : مناسب للتطبيقات الـ Stateless (زي Web Servers)
- StatefulSet : مناسب للتطبيقات الـ Stateful (زي Databases, Kafka, Cassandra)
- مميزاته:
  1. الترتيب في الإنشاء.
  2. أسماء ثابتة للـ Pods
  3. ثبات الهوية حتى بعد إعادة التشغيل.

Creating StatefulSets with Options

🎯 الموضوع الأساسي

إزاي ننشئ StatefulSet وإيه الـ Options اللي ممكن نخصصها.

🔹 إمتى تستخدم StatefulSet ؟

- مش دايمًا محتاج StatefulSet
- تستخدمه لو:
  1. الـ Pods لازم تطلع بترتيب محدد.
  2. كل Pod محتاج اسم ثابت (Stable Identity)
  3. أو محتاج DNS ثابت لكل Pod عشان الـ Apps تقدر توصل لبعض . (عن طريق الـ Headless Service)

🔹 إنشاء StatefulSet

1. شبه الـ Deployment :
   - File فيه apiVersion + kind + metadata + spec
   - جوه الـ spec بتحط Pod template زي العادي.
2. الفرق:
   - kind: StatefulSet (بـ S كبيرة في الاتنين)
   - لازم تضيف serviceName (وده بيكون Headless Service) ..(هاتتشرح تاليا)

◆ سلوك StatefulSet الافتراضي

1. **Ordered, Graceful Deployment**
   - StatefulSet يطلّع الـ Pods بالترتيب Pod0 يشتغل الأول، بعدين Pod1 ، بعدين Pod2 ...
   - كل Pod لازم يبقى Ready قبل ماالللي بعده يطلع.

2. **Stable DNS Records**
   - كل Pod بياخد اسم ثابت في الشبكة (DNS entry) تقدر تستخدمه في التواصل.
   - مثال:
     - mysql-0.svc-name
     - mysql-1.svc-name

3. **Scaling Up (التوسع)**
   - لما تزود عدد الـ Pods : فالـ StatefulSet يطلّعهم بالترتيب.
   - مثال : في MySQL Cluster : هتلاقي ان الـ Slave الجديد يقدر ينسخ من اللي قبله.

4. **Scaling Down (التقليل)**
   - بيتشال بالترتيب العكسي : Pod الأخير هايتشال الأول، بعدين اللي قبل وهكذاا

5. **Deletion (الحذف)**
   - لما تمسح StatefulSet : الـ Pods برضه بيتعملها Delete بالترتيب العكسي.

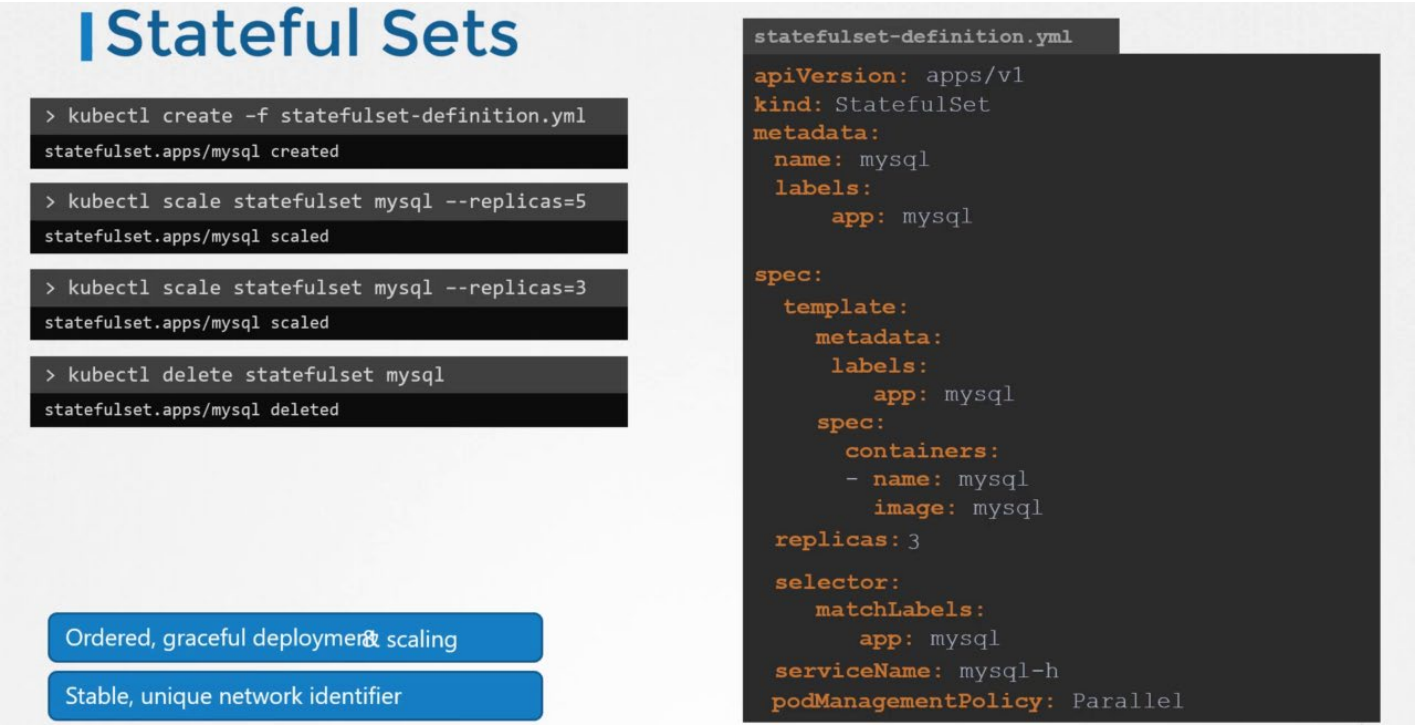
◆ التحكم في طريقة إنشاء الـ Pods

❖ في العادي (Default) :

- podManagementPolicy: OrderedReady
  - يعني لازم Pod0 يبقى Ready قبل ما Pod1 يطلع، وهكذا.

❖ لو عايز كل الـ Pods يطلعوا مرة واحدة:

- حط podManagementPolicy: Parallel
  - كده تستفيد من المميزات DNS ثابت + اسم ثابت (بس من غير الترتيب)



📌 الخلاصة

- StatefulSet زي Deployment بس بيوفر:
  - ترتيب في الإنشاء والحذف.
  - أسماء ثابتة للـ Pods
  - DNS Records ثابتة.
- Useful جدًا مع قواعد البيانات أو أي App محتاج ثبات.
- تقدر تخليه Ordered(افتراضي) أو Parallel حسب الـ Use case

Storage In StatefulSets

🎯 الموضوع الأساسي

Storage مع StatefulSets : إزاي نربط الـ StatefulSet بالـ Storage بحيث كل Pod ياخد الـ Volume الخاص بي.

🔹 Recap سريع (مراجعة)

- قبل كده اتكلمنا عن:
  1. PersistentVolume (PV) : بتتعامل مع الـ Storage الحقيقي.
  2. PersistentVolumeClaim (PVC) : الـ Pod بيطلب Storage من خلاله.
  3. StorageClass : بيخلي العملية Dynamic (أوتوماتيك يعمل Provision للـ PV بدل ما نعمله Manual)
- الطبيعي (Static أو Dynamic Provisioning) :
  - Pod واحد : PVC واحد و PV واحد.

🔹 المشكلة مع StatefulSet

- لو كتبت نفس الـ PVC في Pod Template :
  - كل الـ Pods اللي StatefulSet يطلعهم هتحاول تستخدم نفس الـ Volume
  - ينفع في بعض الحالات (لو محتاجين مشاركة نفس الـ Data بين أكتر من Pod)
  - لكن مش كل الـ Storage Types بيسمحوا بالـ Read/Write من كذا Pod في نفس الوقت.
- مثال:
  - MySQL Replication : كل Pod (Master, Slave1, Slave2) محتاج Volume خاص بي.
  - البيانات بتتنسخ بينهم على مستوى الـ DB مش على مستوى الـ Storage

---

🔹 الحل : VolumeClaimTemplates

- بدل ما تعمل PVC يدوي لكل Pod ،
- بتحط Template للـ PVC جوه StatefulSet
- يعني كل Pod StatefulSet يطلّعه .. يتعمل له PVC خاص به أوتوماتيك.
- الـ PVC بيروح يعمل Bind مع PV (اللي StorageClass وفره)

VolumeClaimTemplate

pvc-definition.yaml

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: data-volume
spec:
 accessModes:
    - ReadWriteOnce
 storageClassName: google-storage
 resources:
    requests:
      storage: 500Mi
```

statefulset-definition.yam

```yaml
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mysql
  labels:
    app: mysql
spec:
  replicas: 3
  selector:
    matchLabels:
      app: mysql
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
      - name: mysql
        image: mysql
        volumeMounts:
         - mountPath: /var/lib/mysql
           name: data-volume
      volumes:
        - name: data-volume
          persistentVolumeClaim:
            claimName: data-volume
```

VolumeClaimTemp

pvc-definition.yaml

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: data-volume
spec:
 accessModes:
    - ReadWriteOnce
 storageClassName: google-storage
 resources:
    requests:
      storage: 500Mi
```

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mysql
  labels:
    app: mysql
spec:
  replicas: 3
  selector:
    matchLabels:
      app: mysql
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
      - name: mysql
        image: mysql
        volumeMounts:
         - mountPath: /var/lib/mysql
           name: data-volume
      volumes:
        - name: data-volume
          persistentVolumeClaim:
            claimName: data-volume
```

VolumeClai

pvc-definition.yaml

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
```

```yaml
  name: mysql
  labels:
    app: mysql
spec:
  replicas: 3
  selector:
    matchLabels:
      app: mysql
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
      - name: mysql
        image: mysql
        volumeMounts:
         - mountPath: /var/lib/mysql
           name: data-volume
volumeClaimTemplates:
- metadata:
    name: data-volume
  spec:
   accessModes:
      - ReadWriteOnce
   storageClassName: google-storage
   resources:
      requests:
        storage: 500Mi
```

🔹 خطوات التشغيل (وراء الكواليس)

1. StatefulSet يطلع Pod0 : و يتعمله PVC0
   - StorageClass يـ Provision PV0 ويربطه بـ PVC0
2. StatefulSet يطلع Pod1 : و يتعمله PVC1
   - StorageClass يـ Provision PV1 ويربطه بـ PVC1
3. StatefulSet يطلع Pod2 : و يتعمله PVC2 ... وهكذا.

✅ في الاخر هتلاقي ان كل Pod بياخد Volume منفصل ومستقل.

🔹 Handling (الفشل)

- لو Pod فشل واتعاد تشغيله أو راح على Node تانية:
  - StatefulSet مايمسحش الـ PVC/PV
  - بيرجع يربط الـ Pod الجديد بنفس الـ PVC القديم.
  - وبالتالي نفس الـ Data بتفضل معاه.

📌 الخلاصة

- StatefulSet + VolumeClaimTemplates = Storage ثابت + مخصص لكل Pod
- بيديك:
  - Stable Storage : نفس الـ Volume بيرجع للـ Pod لو اتعاد تشغيله
  - Automatic PVC/PV Creation : عن طريق StorageClass
- ده اللي بيخلي StatefulSets مناسب جدًا للـ Databases (زي MySQL, MongoDB, Cassandra)

Headless Service

🎯 موضوع الدرس

Headless Services في Kubernetes : إزاي نستخدمها مع StatefulSets علشان نوصل لكل Pod باسمه
(يعني مش عن طريق Load Balancer)

◆ الفكرة الأساسية

- في StatefulSet :
  - كل Pod بياخد اسم ثابت زي mysql-0, mysql-1, mysql-2
- طبيعي في Kubernetes : أي App بيكلم App تاني .. بيستخدم Service
- المشكلة :
  - الـ Service العادي بيعمل Load Balancing : يعني يوزع الترافيك على كل الـ Pods
  - ده مش مناسب مع أنظمة زي MySQL Master-Slave .. طاب ليه ؟!؟!؟!؟
    - Reads ممكن تروح لأي Pod (Master أو Slave)
    - Writes لازم تروح للـ Master بس (mysql-0)

◆ الحل : Headless Service

- Service من غير ClusterIP : يعني مش بيعمل Load Balancing
- وظيفته : يدي لكل Pod الـ DNS Record الثابت . وذلك باستخدام Pod Name & Sub Domain

الشكل :

<PodName>.<HeadlessServiceName>.<Namespace>.svc.cluster.local

مثال:

- mysql-0.mysql-hs.default.svc.cluster.local
- mysql-1.mysql-hs.default.svc.cluster.local

💡 كده نقدر نخلي الـ WebApp يكتب دايمًا في : mysql-0.mysql-hs.default.svc.cluster.local

---

◆ إزاي ننشئ Headless Service ؟

1. نكتب YAML للـ Service عادي.
2. الفرق الوحيد :
3. لما نعمل Deploy : مش هياخدIP ، لكن هيتعمل DNS Records لكل Pod

headless-service.yml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: mysql-h
spec:
  ports:
   - port: 3306
  selector:
      app: mysql
  clusterIP: None
```

🔹 DNS Records والـ Pods

- Pod Definition ممكن ياخد:
  - hostname
  - subdomain
- مع Deployments :
  - ممكن كل الـ Pods بياخدوا نفس hostname/subdomain : DNS Records مكررة = ملهاش لازمة.
- علشان كدا مع StatefulSet :
  - بيعمل أوتوماتيك:
    - Hostname : اسم الـ Pod (mysql-0, mysql-1 ...)
    - Subdomain : اسم الـ Headless Service (mysql-hs)
  - النتيجة DNS Record : ثابت ومميز لكل Pod

```yaml
# headless-service.yml
apiVersion: v1
kind: Service
metadata:
  name: mysql-h
spec:
  ports:
    - port: 3306
  selector:
    app: mysql
  clusterIP: None
```

```yaml
# pod-definition.yml
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: mysql
spec:
  containers:
  - name: mysql
    image: mysql
  subdomain: mysql-h
  hostname: mysql-pod
```

🔹 StatefulSet وService Name

- لما تكتب StatefulSet YAML :
  - لازم تحدد serviceName:
    (اللي هو اسم الـ Headless Service)
- علشان يقدر يربط الـ Pods بالـ DNS الصح.

```yaml
# headless-service.yml
apiVersion: v1
kind: Service
metadata:
  name: mysql-h
spec:
  ports:
    - port: 3306
  selector:
    app: mysql
  clusterIP: None
```

```yaml
# deployment-definition.yml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mysql-deployment
  labels:
    app: mysql
spec:
  serviceName: mysql-h
  replicas: 3
  matchLabels:
    app: mysql
  template:
    metadata:
      name: myapp-pod
      labels:
        app: mysql
    spec:
      containers:
      - name: mysql
        image: mysql
```

📌 الخلاصة

- Service عادي = Load Balancer + ClusterIP
- Headless Service = مفيش ClusterIP ، بس بيدي DNS ثابت لكل Pod
- StatefulSet = Headless Service +
  - كل Pod ليه اسم ثابت.
  - نقدر نوصل للـ Master (mysql-0) أو أي Slave بسهولة عن طريق الـ DNS
- ده الحل المثالي للـ Databases (MySQL, Cassandra, MongoDB, ...)