



kubernetes

Kubernetes
Arquitectura y fundamentos

David Pestana Perdomo

Version 1.1.0 2023-07-09





Cofinanciado por
la Unión Europea



MINISTERIO
DE EDUCACIÓN
Y FORMACIÓN PROFESIONAL



Fondos Europeos



Junta de
Castilla y León

ACCIONES FORMATIVAS DIRIGIDAS AL PROFESORADO Y FORMADORES DE FP (MEDIDA 3.E.17.)

Formación Profesional de Castilla y León

Operación "PROF 2024" financiada por el Ministerio de Educación, Formación Profesional y Deportes y cofinanciado por el Fondo Social europeo Plus, en el marco financiero plurianual 2021-2027.



Contenidos

1. Kubernetes (k8s)	1
1.1. Origen	1
1.2. Introducción	2
1.3. Características	3
1.4. Restricciones	4
2. Arquitectura	5
2.1. Visión general	5
2.2. Master	6
2.2.1. Kube-apiserver	7
2.2.2. Cluster store (etcd)	9
2.2.3. Kube-controller-manager	9
2.2.4. Resumen	10
2.3. Nodos	10
2.3.1. Kubelet	11
2.3.2. Motor de contenedores	12
2.3.3. kube-proxy	12
2.4. Modelo declarativo y estado del cluster	13
2.5. PODS	13
2.5.1. Características	14
2.5.2. Ciclo de vida:	14
2.5.3. Despliegues	15
2.6. Services	15
2.7. ReplicationControllers	15
2.8. Deployments	15
3. Arquitectura KubeScheduler	17
3.1. Paso 1: Análisis de recursos hardware	18
3.2. Paso 2: Análisis de recursos libres	18
3.3. Paso 3: Análisis de nodo específico	19
3.4. Paso 4: Análisis de etiquetas	20
3.5. Paso 5: Análisis de red	21
3.6. Paso 6: Análisis de volúmenes	22
3.7. Paso 7: Análisis de la tolerancia	23
3.8. Paso 8: Análisis de la afinidad	24
4. API Primitives	26
4.1. Manifiestos YAML	26
4.2. Estructura básica de un manifiesto YAML	27
4.3. Ventajas del uso de manifiestos YAML	28
4.4. Obtención de YAML de objetos desplegados en kubernetes	28

5. Node	29
5.1. Requisitos Hardware	31
5.2. Requisitos Software	31
5.3. Requisitos de Red	32
5.4. Tipos de Nodos	32
6. CNI (Container Network Interface)	33
7. Lab: Ajustes en VirtualBox (Clúster Kubernetes OnPremise)	35
7.1. Creando la Red Nat	35
7.2. Asignando la red a las máquinas virtuales	36
8. Motor de contenedores CRI-O	39
8.1. ¿Qué es CRI-O?	39
8.2. ¿Quién está contribuyendo al proyecto?	39
9. Lab: Instalación del motor de contenedores CRI-O (Centos7)	40
9.1. Instalación del runtime CRI-O	40
10. Lab: Instalación del motor de contenedores CRI-O (Fedora)	42
10.1. Instalación del runtime CRI-O	42
11. Lab: Instalación Kubernetes (1 master + 2 minions) - Centos7	44
11.1. Instalación manual con kubeadm	44
11.2. Configuración de master y nodos	45
11.3. Inicio del cluster	46
11.4. Configurando el acceso al clúster	47
11.5. Añadiendo nodos	47
11.6. Comprobando el inventario de nodos	48
12. Lab: Instalación Kubernetes (1 master + 2 minions) - Fedora	49
12.1. Instalación manual con kubeadm	49
12.2. Configuración de master y nodos	50
12.3. Configurando el IP Forwarding	50
12.4. Deshabilitando la memoria SWAP	50
12.5. Instalando los paquetes de Kubernetes	51
12.6. Desactivando systemd-resolved	51
12.7. Estableciendo registry para imágenes no FQDN	52
12.8. Inicio del cluster	52
12.9. Configurando el acceso al clúster en la máquina master	53
12.10. Añadiendo nodos de computación	53
12.11. Comprobando el inventario de nodos	54
13. Lab: Instalación Kubernetes HA (Stacked) - Centos 7	55
13.1. Clonando las máquinas maestras con VirtualBox	55
13.2. Clonando el balanceador de carga (haproxy)	57
13.3. Asignación de recursos a las máquinas virtuales	57
13.4. Ajustando el direccionamiento IP a las máquinas virtuales	58
13.5. Instalando el balanceador de carga (haproxy)	59

13.6. Instalando paquetes de Kubernetes	60
13.7. Creando el archivo de configuración para la red CNI de kubernetes	62
13.8. Realizando limpieza de Iptables	63
13.9. Inicio del cluster	63
13.10. Añadiendo a las máquinas master	65
13.11. Añadiendo las máquinas minions	66
13.12. Comprobando el estado del clúster	66
14. Lab: Instalación Kubernetes HA (Stacked) - Fedora	68
14.1. Clonando las máquinas maestras con VirtualBox	68
14.2. Clonando el balanceador de carga (haproxy)	70
14.3. Asignación de recursos a las máquinas virtuales	70
14.4. Ajustando el direccionamiento IP a las máquinas virtuales	71
14.5. Desactivando systemd-resolved	72
14.6. Estableciendo registry para imágenes no FQDN	73
14.7. Instalando el balanceador de carga (haproxy)	73
14.8. Configurando el IP Forwarding (en nodos de kubernetes)	74
14.9. Deshabilitando la memoria SWAP (en nodos de kubernetes)	75
14.10. Instalando los paquetes de Kubernetes	75
14.11. Creando el archivo de configuración para la red CNI de kubernetes	76
14.12. Realizando limpieza de Iptables	77
14.13. Inicio del cluster	77
14.14. Añadiendo a las máquinas master	79
14.15. Añadiendo las máquinas minions	80
14.16. Comprobando el estado del clúster	80
15. Driver de red: Weave	82
15.1. ¿Cómo funciona?	82
15.2. Configuración Simple	82
15.3. Descubrimiento DNS	83
16. Lab: Instalación del Driver de red Weave en Kubernetes	84
16.1. Aplicando el manifiesto	84
16.2. Comprobando el inventario de los nodos	84
17. Federación con Kops	86
17.1. Requisitos para su uso	86
17.2. RoadMap de liberación de versiones	87
18. Lab: Federación con Kops	89
18.1. Instalación del cliente de Amazon (awscli)	89
18.2. Configuración del cliente de Amazon (awscli)	89
18.3. Configurando recursos (Gestión de identidades y accessos IAM)	90
18.4. Configurando el usuario de Amazon	91
18.5. Configurando par de claves SSH	93
18.6. Instalando Kops	94

19. kubectl describe node	98
19.1. Listando los nodos	98
19.2. Etiquetando los roles de los nodos	98
19.3. Describiendo el estado de un nodo	99
19.4. Obteniendo información del clúster	103
20. Kubectl taint	105
20.1. Efectos del taint en el nodo	105
20.2. Creando el taint en el nodo kubeminion1 (NoSchedule)	106
20.3. Creando el taint en el nodo kubeminion2 (NoExecute)	106
20.4. Comprobando que el taint está presente en el nodo kubeminion1	107
20.5. Comprobando que el taint está presente en el nodo kubeminion2	108
20.6. Operadores del taint en el Pod	110
20.7. Creando un Pod con regla del taint (NoSchedule)	111
20.8. Eliminar el taint de un nodo	112
21. Mantenimiento de los nodos	113
21.1. Operación de Acordonado (cordón)	114
21.2. Operación de Desacordonado (uncordón)	114
21.3. Operación de Dreando (Drain)	115
21.4. Operación Backup & Restauración	116
22. Lab: Mantenimiento de los nodos	117
22.1. Operación acordonamiento del nodo	117
22.2. Operación drenado del nodo	117
22.3. Eliminación de nodo + generación de nuevo token de acceso	119
22.4. Backup del componente etcd	122
22.5. Restaurando backup del componente etcd	124
23. Actualización de clúster de Kubernetes	125
23.1. Comprobando la versión del agente kubelet	125
23.2. Comprobando la versión de kubectl + kube-api-server	125
23.3. Comprobando la versión del kube-proxy	125
23.4. Comprobando la versión del kube-controller-manager	126
23.5. Actualizando paquetes del sistema, sin tocar kubernetes	127
23.6. Actualizando kubernetes (Nodos maestros)	127
23.7. Instalando una versión específica de kubeadm	127
23.8. Actualizando mediante kubeadm	129
23.9. Actualizando la versión de kubectl	130
23.10. Actualizando el agente kubelet	130
23.11. Comprobando la versión de kubernetes en los nodos	130
23.12. Actualizando kubernetes (Nodos Minions)	131
24. Pods	132
24.1. Implicaciones de múltiples contenedores en un mismo Pod	133
24.2. Comunicaciones	134

24.3. Comunicación interpods (Mismo Nodo)	134
24.4. Comunicación interpods (Distintos Nodos)	135
24.5. Ciclo de vida	135
24.6. Acotamiento de recursos	136
25. Lab: Pods	138
25.1. Creando 1 Pod con 1 container	138
25.2. Listando los Pods	139
25.3. Actualizando el Pod que tiene desplegado 1 container	140
25.4. Creando 1 Pod con 2 containers	140
25.5. Añadiendo variables de entorno al Pod	141
25.6. Describiendo el Pod	142
25.7. Obteniendo manifiesto YAML a partir de un objeto desplegado en kubernetes	143
25.8. Probando conexiones HTTP con el Pod	143
25.9. Conectándonos dentro del Pod (Origen Mode :D)	144
25.10. Obteniendo logs del Pod	146
25.11. Attach a un Pod	147
25.12. Forwarding de puerto a un Pod	148
25.13. Selección del nodo para el Pod	148
25.14. Afinidad del Pod (Node affinity)	150
25.15. Health Check: livenessProbe	152
25.16. Health Check: readinessProbe	154
25.17. Health Check: startupProbe + sideCar	155
25.18. Límites de recursos	157
25.19. Eliminando los Pods	158
26. Autoescalado	159
26.1. Horizontal Pod Autoscaler	159
26.1.1. ¿Cómo funciona?	160
27. Lab: Autoescalado Horizontal Pod Autoscaler	161
27.1. Creando manifiesto de autoescalado	161
27.2. Obteniendo la IP del servicio	163
27.3. Realizando peticiones con JMeter	163
27.4. Observando el comportamiento del auto-escalado	163
28. Jobs	165
28.1. Tipos de Jobs	165
28.2. Tratando errores en contenedores fallidos	166
28.3. Política backoffLimit	167
28.4. Limpieza de Jobs finalizados	168
28.5. Terminación de Jobs	168
28.6. Limpieza automática de Jobs finalizados	169
29. Lab Jobs	170
29.1. Creando un Job que calcula 2000 dígitos del número Pi	170

29.2. Creando un Job que calcula 2000 dígitos del número Pi (Número de reintentos + fallo provocado)	171
29.3. Creando Jobs en paralelo (Cantidad de ejecuciones)	173
29.4. Creando Jobs en paralelo (Cantidad de ejecuciones + Cantidad de paralelismo)	174
29.5. Creando Jobs (Modo de operación, cola de procesos)	176
29.6. Creando Jobs (Autolimpieza de Job finalizado)	177
30. Lab: Kube-Scheduler	179
30.1. Creando nuestro scheduler personalizado	179
30.2. Creando el ClusterRole	180
30.3. Creando el ClusterRoleBinding	181
30.4. Creando el Rol	182
30.5. Creando el RoleBinding	182
30.6. Editando el kube-scheduler, activando nuestro scheduler	183
30.7. Comprobando el scheduler	184
30.8. Asignando pods a scheduler	185
30.9. Visualizando eventos (A nivel de Pod del scheduler)	187
30.10. Visualizando eventos (A nivel de event log de kubernetes)	187
30.11. Visualizando eventos (A nivel de log del pod del scheduler)	188
31. Namespace	189
31.1. ¿Cuándo utilizar multiples espacios de nombre?	189
31.2. Espacios de nombre por defecto	190
31.3. ¿Qué elementos no operan bajo un espacio de nombres?	190
32. Lab: Namespaces	191
32.1. Listando los namespaces	191
32.2. Estados de un namespace	193
32.3. Creando espacios de nombres	193
32.4. Desplegando un pod en un espacio de nombres	194
32.5. Estableciendo preferencia del espacio de nombres	195
32.6. Eliminando espacio de nombres	196
32.7. Comprobando qué elementos están bajo un espacio de nombres	196
32.8. Espacios de nombres y DNS	199
33. Labels	200
33.1. Restricciones de nombre	200
34. Lab: Labels	202
34.1. Creando pods con etiquetas	202
34.2. Consultando las etiquetas de los pods	202
34.3. Añadiendo etiquetas al pod	202
34.4. Filtrando pods mediante etiqueta	203
34.5. Filtrando pods por estado de ejecución	204
34.6. Filtrando pods por propiedad de metadata	204
34.7. Obteniendo logs del Pod, filtrando por etiqueta	205

34.8. Eliminando etiquetas del pod	206
34.9. Eliminando pod mediante etiquetas	207
35. DaemonSet	208
35.1. Operativa del DaemonSet	208
36. Lab: DaemonSet	210
36.1. Observando los DaemonSets del namespace kube-system	210
36.2. Probando regeneración DaemonSets de sistema	210
36.3. Creando un DaemonSet	211
36.4. Eliminación del DaemonSet	213
37. ReplicationController	215
37.1. Desventajas de uso de Pods simples	215
37.2. Ventajas de uso	215
37.3. ¿Cuándo debemos de usarlo?	216
37.4. Etiquetas en ReplicationController	216
38. Lab: ReplicationController	217
38.1. Creación del primer ReplicationController	217
38.2. Análisis de estructura mínima	217
38.3. Listando los Pods creados por el ReplicationController	218
38.4. Listando las unidades de replicación	218
38.5. Describiendo la unidad de replicación	218
38.6. Escalando el número de réplicas	220
38.7. Simulando caída de servidor	221
38.8. Obteniendo información detallada de las unidades rc	223
38.9. Eliminando las unidades rc	223
39. ReplicaSet	225
39.1. ¿Cómo funciona la nueva unidad?	225
39.2. Ventajas de uso	225
39.3. ¿Cuándo debemos de usarlo?	225
39.4. Etiquetas en ReplicationController	226
40. Lab: ReplicaSet	227
40.1. Creación del primer ReplicaSet	227
40.2. Análisis de estructura mínima	227
40.3. Listando los Pods creados por el ReplicaSet	228
40.4. Listando las unidades de replicación	228
40.5. Describiendo la unidad de replicación	229
40.6. Escalando el número de réplicas	230
40.7. Simulando caída de servidor	232
40.8. Añadiendo manualmente un Pod con nexo al ReplicaSet	235
40.9. Obteniendo información detallada de las unidades rs	236
40.10. Obteniendo metadata en formato .yml de un pod desplegado por el rs	236
40.11. Eliminando las unidades rs con Pods	237

40.12. Eliminando únicamente la unidad rs sin eliminar los Pods	238
41. Secrets	240
41.1. ¿Para qué valen?	240
41.2. ¿Quién puede utilizar secrets?	241
42. Lab: Secrets	242
42.1. Creando nuestros propios secretos con kubectl (inline)	242
42.2. Listando secretos	242
42.3. Describiendo los secretos	243
42.4. Creando secretos mediante archivo de manifiesto (.yml)	243
42.5. Obteniendo descripción detallada del secreto	244
42.6. Creando secretos que no estén en base64	245
42.7. Prioridad a la hora de definir secretos	246
42.8. Decodificando un secreto que esté en base64	247
42.9. Editando un secreto	247
42.10. Utilizando un secreto dentro de un Pod	248
42.11. Utilizando un secreto dentro de un Pod (especificando secretos y rutas de montado de claves)	250
42.12. Especificando permisos de los archivos de montaje	251
42.13. Actualización de los secretos	253
42.14. Utilizando secretos como variables de entorno	253
42.15. Limitaciones de los secretos/seguridad	254
43. Services	256
43.1. Arquitectura del servicio	258
43.2. Tipos de Servicios	259
43.3. Ingress	261
43.4. Labels	261
43.5. Service Discovery DNS	262
44. Lab: Services	263
44.1. Creando el deployment para operaciones	263
44.2. Creando un servicio de tipo ClusterIP (DHCP)	263
44.3. Creando un servicio de tipo NodePort (DHCP)	265
44.4. Creando un servicio de tipo ClusterIP (IP personalizada + binding a interfaz externa)	267
44.5. Listando los endpoints	269
44.6. Listando las reglas de iptables	269
44.7. Creando servicio NodePort mediante exposición en línea (Sin YAML)	270
44.8. Eliminando los servicios	272
44.9. Creando un servicio de tipo LoadBalancer	272
44.10. Política de enrutamiento del balanceador de carga	275
44.11. Comprobando el funcionamiento DNS	276
44.12. Ajustando DNS específica	278
44.13. Creando servicio para acceder a pod con DNS específica de servicio (ClusterIP)	280

44.14. Uso avanzado	282
45. Ingress Controller: Nginx	283
45.1. Lab: Ingress Controller (Nginx)	284
45.1.1. Creando el espacio de nombres (Namespace)	284
45.1.2. Creando la cuenta de servicio (ServiceAccount)	284
45.1.3. Creando el cluster role y realizando el binding (ClusterRole y ClusterRoleBinding)	285
45.1.4. Creando el cluster role y realizando el binding para aplicaciones protegidas (SSL)	287
45.1.5. Creando el secreto con el certificado TLS y la clave privada para el nginx (Secret)	288
45.1.6. Creando el objeto de configuración para nginx (ConfigMap)	289
45.1.7. Creando la IngressClass	290
45.1.8. Creando el VirtualServer	291
45.1.9. Creando el VirtualServerRoute	301
45.1.10. Creando el TransportServer	311
45.1.11. Creando las Policy	313
45.1.12. Creando el recurso personalizado APLogConf	315
45.1.13. Creando el recurso personalizado APPolicy	316
45.1.14. Desplegando el Ingress Controller nginx	338
45.1.15. Verificando que el controlador ingress nginx está online	339
45.1.16. Creando el servicio LoadBalancer de acceso	340
45.1.17. Verificando que todos los componentes del controlador ingress están operativos	341
45.2. Lab: Ingress App utilizando Ingress Controller (Nginx)	342
45.2.1. Creando el namespace	342
45.2.2. Creando el deployment (Aplicación Versión 1.0)	342
45.2.3. Creando el deployment (Aplicación Versión 2.0)	343
45.2.4. Creando los servicios	344
45.2.5. Creando las reglas de ingress	345
45.2.6. Añadiendo entradas DNS	347
45.2.7. Probando la ejecución de las reglas de ingress	347
46. Nexus Integration	348
46.1. Lab: Nexus Integration	349
46.1.1. Creando la unidad de persistencia (PV)	349
46.1.2. Creando el namespace para nexus	350
46.1.3. Creando el StatefulSet	350
46.1.4. Creando el servicio para las reglas del Ingress Controller	353
46.1.5. Añadiendo entradas DNS	354
46.1.6. Creando las reglas de Ingress	354
46.1.7. Obteniendo la contraseña inicial de nuestro sistema nexus	356
46.1.8. Accediendo al portal web nexus	356
46.1.9. Configurando Nexus	357
46.1.10. Probando una petición contra el repositorio	359
46.1.11. Configuración del daemon docker	359

46.1.12. Login en el nexus docker registry	359
46.1.13. Reetiquetando imagen docker	360
46.2. Lab: Desplegando aplicación con imagen de Nexus	361
46.2.1. Creando el namespace	361
46.2.2. Creando el secreto (Credenciales de acceso a nexus)	361
46.2.3. Creando el deployment de la aplicación	361
47. Deployments	364
47.1. Características del uso de Deployments	365
47.2. Implicaciones	365
48. Lab: Deployments	366
48.1. Creando 1 Deployment con 1 container	366
48.2. Análisis de estructura mínima	366
48.3. Listando los Pods creados por el Deployment	367
48.4. Listado las unidades Deployment	367
48.5. Describiendo la unidad de Deployment	368
48.6. Escalando el número de réplicas	369
48.7. Describiendo la unidad de replicación (ReplicaSet)	370
48.8. Comprobando que responde algún Pod con IP interna	371
48.9. Creando un Deployment con directivas de Update	373
48.10. Desplegando un Deployment con directivas de Update (rollout status)	374
48.11. Llevando a cabo una operación de Rollback (rollout history)	376
48.12. Pausando una operación de actualización (rollout pause)	377
48.13. Resumiendo una operación de actualización pausada (rollout resume)	379
48.14. Reiniciando de nuevo la operación de despliegue (rollout restart)	380
48.15. Eliminando los deployments	381
48.16. Despliegue ReadinessProbe	382
49. Volumen	385
49.1. Volúmenes Docker vs Kubernetes	385
49.1.1. Volúmenes en Docker	385
49.1.2. Volúmenes en Kubernetes	386
49.2. ¿Cómo usa el volumen un Pod?	386
49.3. Tipos de volúmenes soportados por Kubernetes	386
49.3.1. Ofertados por proveedores de nube privada	386
49.3.2. Con características de sistema de archivo distribuido	387
49.3.3. Para poder utilizarlo en nube propia (OpenStack)	387
49.3.4. Locales al nodo Kubernetes	387
49.3.5. Compartición de archivos	387
49.3.6. Otros tipos	387
49.4. Volúmen Local (PersistentVolume - PV)	388
49.5. Volúmen Local (PersistentVolumeClaim - PVC)	388
50. Lab: Volumen hostPath	390

50.1. Creando 1 Pod con 1 container que usa un volumen	390
50.2. Creando 1 Pod con 2 container que usan un volumen compartido.....	392
51. Lab: Volumen emptyDir	394
51.1. Posibles usos de este tipo de volumen	394
51.2. Creando un 1 Pod con 2 containers que usan un volumen compartido temporal	394
52. PersistentVolume	396
52.1. Aprovisionando estático vs dinámico	396
52.1.1. Estático	396
52.1.2. Dinámico	396
52.2. Capacity	397
52.3. VolumeMode	397
52.4. Access Modes	397
52.5. StorageClassName	398
52.6. PersistentVolumeReclaimPolicy	399
52.7. MountOptions	399
52.8. NodeAffinity	400
53. PersistentVolumeClaim	401
54. Lab: PersistentVolume	403
54.1. Creando una unidad de persistencia de 5Gb y acceso R/W	403
54.2. Listando las unidades de persistencia	404
55. Lab: PersistentVolumeClaim	405
55.1. Creando PersistentVolumeClaim que usa PersistentVolume	405
55.2. Listando las claims (PVC)	405
55.3. Añadiendo un volumen a un Pod a partir de un claim	406
56. Volúmenes mediante NFS	407
57. ServiceAccount	411
58. Lab: Cuentas de Servicio	412
58.1. Consultando las cuentas de servicio por defecto en Kubernetes	412
58.2. Creando nuestra propia cuenta de servicio	412
58.3. Obteniendo la cuenta de servicio en formato YAML	413
58.4. Especificando la cuenta de servicio de nuestro Pod	413
58.5. Comprobación de credenciales kubectl	414
58.6. Creando otro usuario para acceder al clúster	416
58.7. Configuraciones en el nodo kubeminion2	417
59. Procedimiento en las comunicaciones seguras	419
59.1. Fase de Autenticación	420
59.2. Fase de Autorización	420
59.3. Fase de Admisión	420
59.4. Certificado de autorización	421
60. RBAC (Role Based Access Control)	422
60.1. ClusterRole	423

60.2. Role	423
60.3. RoleBinding	424
60.4. ClusterRoleBinding	424
60.5. ServiceAccount	424
61. Lab: RBAC (Consulta de unidades PV desde un namespace distinto al default)	426
61.1. Creando el archivo del laboratorio	426
61.2. Creando un namespace propio	426
61.3. Creando ClusterRole	426
61.4. Creando ClusterRoleBinding	427
61.5. Creando nuestro Rol personalizado	427
61.6. Creando el RoleBinding	428
61.7. Creando un Pod que use el namespace	428
61.8. Test: Levantando proxy en el host para comunicaciones con la API	429
61.9. Test: Utilizando el proxy del pod para comunicaciones con la API	430
62. Lab: RBAC (Consulta de unidades PV desde el namespace default)	432
62.1. Creando el archivo del laboratorio	432
62.2. Creando ClusterRole	432
62.3. Creando ClusterRoleBinding	433
62.4. Creando nuestro Rol personalizado	433
62.5. Creando el RoleBinding	433
62.6. Creando un Pod que use el namespace	434
62.7. Test: Utilizando el proxy del pod para comunicaciones con la API	434
63. NetworkPolicy	437
64. Lab: NetworkPolicy	438
64.1. Creando política de red, denegando comunicaciones	438
64.2. Creando Pods afectados por la política	438
64.3. Comprobando comunicaciones	439
64.4. Estableciendo regla de comunicaciones (Política Ingress a nivel de Pod)	439
64.5. Estableciendo regla de comunicaciones (Política Ingress a nivel de NameSpace)	442
64.6. Estableciendo regla de comunicaciones (Política Ingress a nivel de IP)	447
64.7. Estableciendo regla de comunicaciones (Política Egress a nivel de Pod)	450
65. Certificados TLS	453
66. Lab: Certificados TLS	454
66.1. Mostrando la CA de un pod del clúster	454
66.2. Instalando la herramienta CFSSL	454
66.3. Generando la solicitud de certificado CSR	455
66.4. Aprobando el certificado	458
66.5. Extrayendo el certificado	458
67. Contexto de seguridad de ejecución	460
68. Lab: Contexto de seguridad de ejecución	461
68.1. Comprobando contexto de seguridad	461

68.2. Arrancando Pods con otro contexto de usuario	461
68.3. Arrancando Pod con contexto de no-root	462
68.4. Arrancado Pod con contexto Privileged	465
68.5. Arrancando un Pod añadiéndole capabilities	467
68.6. Arrancando un Pod eliminándole capabilities	468
68.7. Arrancando un Pod con contexto de seguridad de volumen, sólo lectura	469
68.8. Arrancando un Pod con contexto de seguridad a nivel de Pod	470
69. ConfigMap	473
70. Lab: ConfigMap	474
70.1. Generando la clave privada	474
70.2. Creando el secreto	474
70.3. Creando el ConfigMap	475
70.4. Creando el Pod que usa el ConfigMap y el Secret	476
71. Service Mesh	480
71.1. ¿Por qué es conveniente la malla de servicios para la arquitectura de microservicios?	480
71.2. ¿Qué es una malla de servicio - Service Mesh?	480
71.3. ¿Cómo funciona una malla de servicio - Service Mesh?	482
71.4. Principales ventajas de un Service Mesh	482
71.5. Principales desventajas de un Service Mesh	482
72. Linkerd	484
72.1. ¿Cómo funciona?	485
73. Lab: Linkerd	486
73.1. Verificar la versión de kubernetes	486
73.2. Instalación del cliente de consola	486
73.3. Añadir binario de ejecución al path	487
73.4. Comprobando versión instalada	487
73.5. Validando el clúster de kubernetes	487
73.6. Ajustando la zona horaria en el sistema	488
73.7. Instalando Linkerd	489
73.8. Comprobando la instalación de Linkerd	489
73.9. Activando el dashboard de control	490
74. Lab: Desplegando una aplicación con Linkerd	492
74.1. Instalación de la aplicación	492
74.2. Ejecutando un proxy de acceso	492
74.3. Aplicando la malla de servicios Linkerd a la aplicación	493
74.4. Comprobando el funcionamiento	495
75. Monitorización	497
75.1. describe pod	497
75.2. get events	497
75.3. get pod	498
75.4. status	498

76. Lab: Monitorización	499
76.1. Instalando el Servidor de métricas	499
76.2. Comprobando el servidor de métricas	500
76.3. kubectl top	501
76.4. Cluster Logs	503
77. Dashboard	508
78. Lab: Dashboard Kubernetes	509
78.1. Instalación mediante manifiesto	509
78.2. Activando el proxy de conexión	509
78.3. Creando la cuenta de servicio	510
78.4. Configurando el ClusterRoleBinding	510
78.5. Obteniendo el Token de acceso	511
79. Glosario de términos	513
80. Helm	514
80.1. Helm Charts	514
80.2. Arquitectura de Helm Charts	514
80.3. Arquitectura de Helm Charts (Tiller)	515
80.4. Arquitectura de Helm Charts (Helm)	515
80.5. Beneficios de utilizar Helm	515
80.5.1. Estado actual del proyecto	515

Capítulo 1. Kubernetes (k8s)

1.1. Origen

- Creado por Google
- Donado al *Cloud Native Computing Foundation* en 2014 (Plataforma OpenSource)
- Escrito en Go/Golang
- Repositorio en Github
 - <https://github.com/kubernetes/kubernetes>
- Google lo utilizó para gran cantidad de servicios internos como búsquedas, gmail, etc.
- Con ciertas herencias de los frameworks Borg y Omega de Google, tecnologías propietarias.
 - **Borg:** *Large-scale cluster management* para gestión de miles de nodos.



Figure 1. Borgle - Imagen del artículo [Search Engine Land](#)

- **Omega:** Planificador de tareas para clusters a gran escala

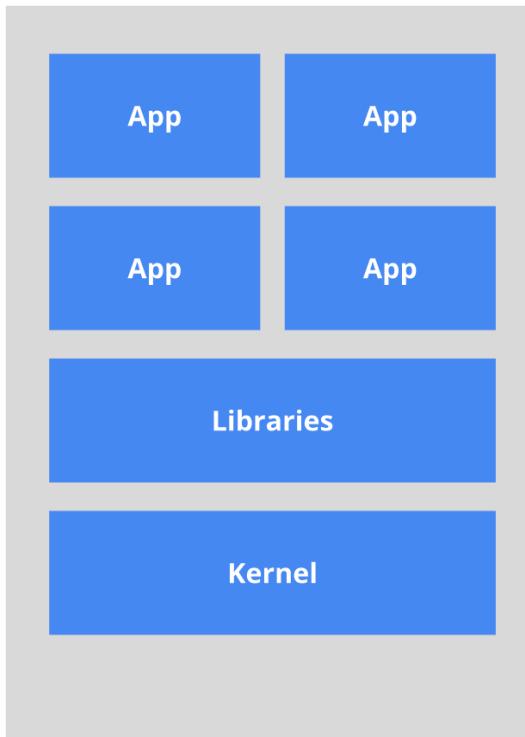


Figure 2. Imagen del artículo [Medium](#)

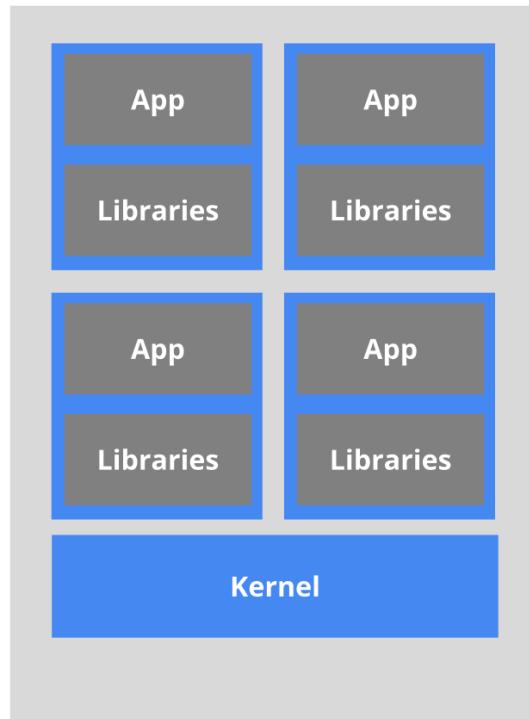
- Borg fue el primer sistema, evolucionó a Omega, y con esa base nació Kubernetes.
- Acortado a **k8s**

1.2. Introducción

- Helmsman, o pilot. Navegante
- Visión de un datacenter como una sola computadora.
- Permite la automatización de despliegues, escalado y gestión de aplicaciones en contenedores.
- Entre las características principales se encuentran:
 - Despliegue de contenedores por requerimientos y otras variables.
 - Reinicio de contenedores en caso de fallo.
 - Recolocación de contenedores en caso de caída de nodos.
 - Escalado horizontal
 - Descubrimiento de servicios y balanceo de carga
 - Actualizaciones y rollbacks automáticos
 - Orquestación de almacenamiento.
 - Gestión de ejecuciones Batch.
- El despliegue de aplicaciones permite usar contenedores que contienen las librerías necesarias para su funcionamiento aislando de las demás
- Orientado a gestionar un sistema de empaquetamiento estándar, asociado a un manifiesto.
- Fuertemente posicionado en el mercado
- Agnóstico de la plataforma



*Heavyweight, non-portable
Relies on OS package manager*



*Small and fast, portable
Uses OS-level virtualization*

- Los contenedores son pequeños y rápidos.
- Una aplicación se empaqueta en una imagen de contenedor.
- Son imágenes inmutables

1.3. Características

- Kubernetes permite programar y ejecutar aplicaciones de contenedores en clusters físicos o virtuales
- Permite la abstracción de esta característica y pasar a una infraestructura de contenedores como la que kubernetes ofrece.
- Kubernetes permite
 - Monitorización del sistema de almacenamiento
 - Distribución de claves
 - Monitorización de salud de aplicaciones
 - Replicación de instancias de aplicación
 - Autoescalado de Pods horizontal
 - Nombrado y descubrimiento
 - Balanceo de carga
 - Actualizaciones en cascada
 - Monitorización de recursos
 - Logs

- Debug de aplicaciones
- Autenticación y autorización

1.4. Restricciones

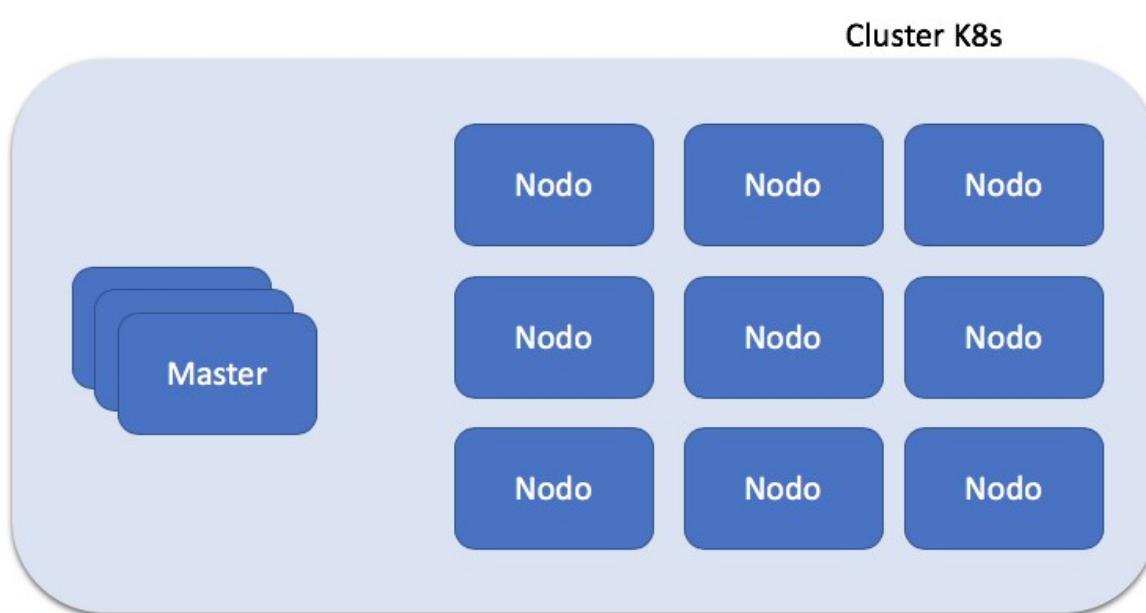
- No limita las aplicaciones soportadas basadas en lenguajes
- No distingue entre aplicaciones y servicios
- No provee de ningún tipo de middleware, frameworks de procesamiento de datos, base de datos, ni sistema de almacenamiento de disco en cluster, pero permite la ejecución de dichos recursos en kubernetes
- No existe un marketplace para instalación sencilla
- No realiza CI ni compila.
- Permite que los usuarios definan el log de aplicación, monitorización y alerta.
- No provee de una configuración, gestión o sistema de auto-healing



Capítulo 2. Arquitectura

2.1. Visión general

- Orquestador de aplicaciones basadas en microservicios
- Una aplicación está gestionada por microservicios que deben ser orquestados para su correcto funcionamiento.
- Kubernetes permite definir el comportamiento y despliegue de componentes (pods) dentro de un cluster de 1 o más servidores.
- Un cluster **k8s** está formado por uno o más masters, y uno o más nodos.



- Los nodos se les llama minions.
- El master toma las decisiones de orquestación frente a los minions.
- Los minions realizan el trabajo real, despliegan las aplicaciones y obedecen a las órdenes del master.
- La metodología estándar de operación es empaquetar los componentes en un despliegue y su manifiesto. Este se ejecuta en el master y éste da las órdenes a los minions para realizar las operaciones pertinentes para realizar el despliegue.



Figure 3. Cluster en funcionamiento, varios nodos maestros en comunicación coordinada

2.2. Master

- Se trata de un conjunto de pods que se ejecutan en un solo servidor.
- Se permite generar una estructura de Masters en HA para evitar la caída de servicio.
- No se deben cargar con trabajo en el master ya que es posible.
- Provee del panel de control de Kubernetes.
- Permite la toma de decisiones global sobre el cluster
- Se pueden ejecutar en cualquier nodo del cluster, aunque suelen ejecutarse en la misma VM, y prohíbe la ejecución de contenedores de usuario dentro del nodo.
- Posee una serie de pods que permiten dar el servicio de Master.



Figure 4. Nodos maestros en conjunto, su misión es controlar el despliegue

2.2.1. Kube-apiserver

- Expone el API (Rest) de comunicación con el cluster
- Es el panel de control principal, designado para escalado horizontal
- Consumo json vía los ficheros de manifiesto



Figure 5. Encargado de las comunicaciones del clúster



Figure 6. Panel de control principal del despliegue

2.2.2. Cluster store (etcd)

- Almacenamiento persistente
- Permite almacenar el estado del cluster y su configuración
- Se trata de un servicio k/v (clave/valor)
- Distribuido, consistente y monitorizable
- Se deben realizar backups continuos de los datos de este contenedor, ya que si lo perdemos, no tendremos cluster.



Figure 7. Encargado de la persistencia del estado del clúster

2.2.3. Kube-controller-manager

- Se trata de un controlador de controladores.
- Ejecuta hilos que gestionan tareas en el cluster. Cada controlador es un proceso separado. Está compilado como un binario y ejecutado en un solo proceso.
- Permite gestionar:
 - Control de nodos: Responsable de indicar cuando un nodo se cae.
 - Control de replicación: Mantiene el numero correcto de PODS para cada objeto de control de replicación en el sistema
 - Enpoint de Control: Publica Endpoints, uniones entre servicios y pods
 - Servicio de cuentas y controladores de token. Crea las cuentas por defecto y los tokens de acceso a las apis para nuevos namespaces

- Control de Namespace
- etc.
- Monitoriza los cambios y realiza los cambios necesarios para alcanzar el estado deseado.



Figure 8. El contramaestre es responsable de la marinería y de dirigir las maniobras, se ocupa de la estiba (colocar carga a bordo del buque) y del mantenimiento de la nave

2.2.4. Resumen

- El nodo Master es el corazón de Kubernetes, posee una serie de pods de servicio que le permiten realizar todas las operaciones contra los minions y almacenar la información relevante en un almacenamiento persistente.
- Solo se permite su comunicación con el endpoint del apiserver, cuyas comunicaciones están protegidas a través del puerto 443
- En ese punto se realiza la comunicación y la autenticación y autorización.
- Para hablar con el servicio usamos una herramienta cliente llamada kubectl que permite hablar con el API de forma sencilla, formando el json de comunicación.
- El apiserver es monitorizado por el planificador, que realiza las tareas para alcanzar el estado deseado del cluster.

2.3. Nodos

- Minions o workers de kubernetes
- Formados por tres componentes
 - Kubelet

- El motor de contenedores
- Kube-proxy



2.3.1. Kubelet

- Agente principal de kubernetes (Es el minion)
- Registra el nodo en el cluster
- Monitoriza el apiserver del master
- Instancia los pods
- Informa al master de los resultados de las operaciones.
 - En caso de fallo, es el master el que decide que hacer con el pod que haya fallado, no el agente.
- Posee un endpoint en el puerto 10255 con accesos como por ejemplo
 - /spec → Información del nodo
 - /healtz → Salud del nodo
 - /pods → Los pods desplegados
 - etc.
- **Operaciones:**

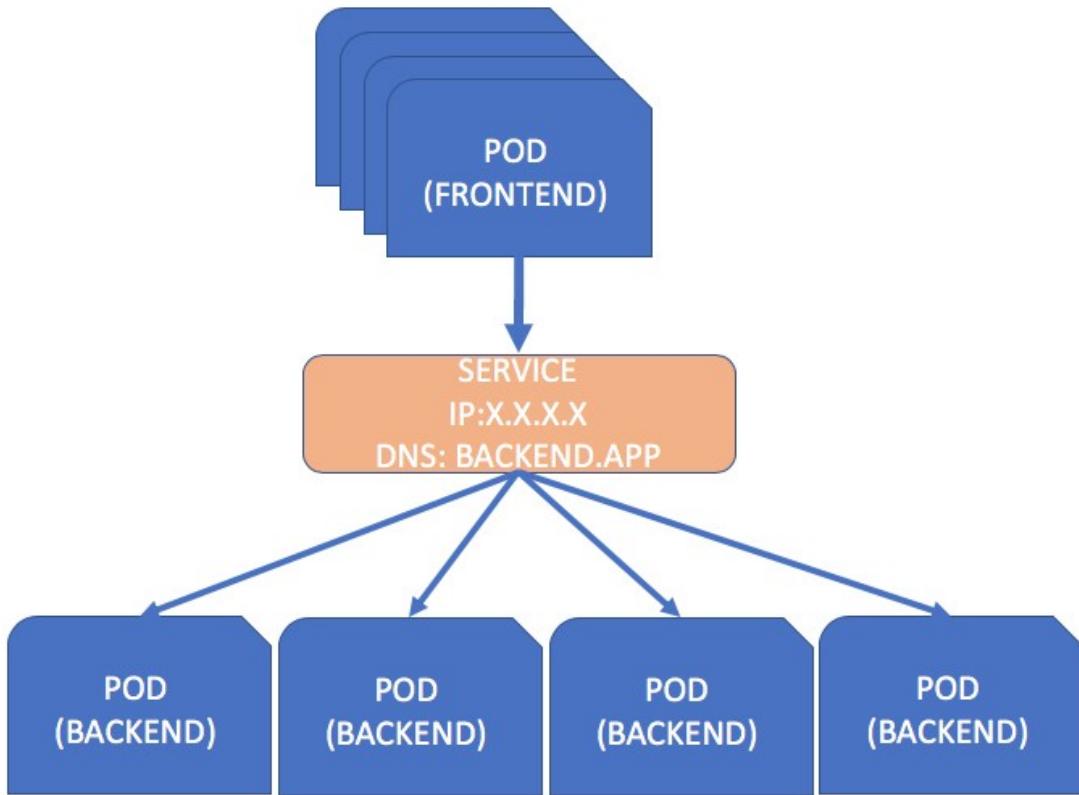
- Busca los pods asignados al nodo
- Monta los volumenes requeridos del POD
- Descarga los *secrets* del POD
- Periódicamente ejecuta un test de vida al contenedor indicado
- Informa del estado del POD, y crea un POD espejo si es necesario
- Informa del estado del nodo al sistema

2.3.2. Motor de contenedores

- Permite gestionar contenedores
 - Descarga de imagenes
 - Inicio y parada de contenedores
 - etc.
- Es modular
 - Normalmente usado con Docker
 - Permite rkt (Rocket) como sustituto a Docker

2.3.3. kube-proxy

- Red del nodo
- Definel la dirección de red de un pod
 - Todos los contenedores desplegados en un pod comparten la misma ip
- Permite gestionar balanceo de carga en todos los pods en un servicio



2.4. Modelo declarativo y estado del cluster

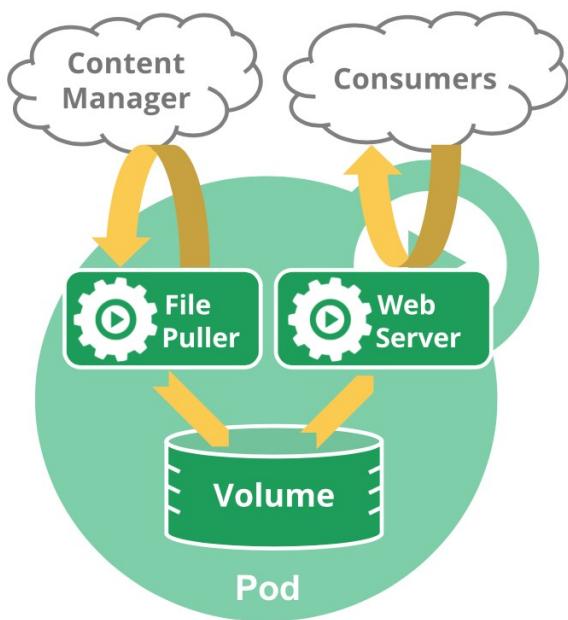
- Un fichero de manifiesto en yaml o json describe cual es el estado deseado del cluster.
- Este manifiesto se envia al apiserver en el master para que realice los cambios
- El master calcula desde el estado actual cuales son los cambios a realizar en los servidores.
- En el caso de que perdamos un nodo, el cluster recalcula para que vuelva a estar el cluster en el estado deseado.

2.5. PODS

- Se trata de un bloque básico de construcción de kubernetes
- Es la unidad mas simple
- Encapsula un contenedor de aplicación, o en algunos casos, múltiples contenedores
- Encapsula almacenamiento, una ip de red única, y opciones para orquestar los contenedores
- Representa una instancia con contenedores acoplados
- Objetivo
 - Un contenedor por POD. Es el caso de uso más común.
 - Muchos contenedores por POD. Contenedores que deben estar juntos para compartir recursos (Uso avanzado).

2.5.1. Características

- Posee un entorno completo para la supervivencia de los contenedores
 - Pila de red
 - Punto de montaje
 - Espacio de nombres del kernel
 - Memoria asignada
 - ...
- Todo compartido dentro del mismo pod, luego todos los contenedores comparten el entorno
- Los contenedores altamente acoplados se crean dentro de un solo pod
- Los contenedores poco acoplados los creamos en distintos pods
- Los pods son las unidades básicas de escalado. No podemos agregar contenedores dentro de un pod, no son escalables.
- Dentro de un pod puede haber un servidor web y un extractor de logs, considerando el contenido como atómico.



- Si un pod no ha levantado todos los contenedores, el pod no está iniciado.

2.5.2. Ciclo de vida:

- **Pending** : Pendiente de iniciar todos sus componentes
- **Running** : En ejecución
- **Succeeded/failed** : Destruido, ya sea correcto o por un fallo, no se intenta reactivar, se crea un pod nuevo.

2.5.3. Despliegues

- Podemos desplegar un POD directamente por medio de un manifiesto.
- El servidor valida el manifiesto y aplica el estado deseado.
- Despliega un solo pod.
- Sin embargo, usando una estructura mas compleja como un Deployment, podemos desplegar Pods desde un controlador de replicación, lo que permite desplegar múltiples pods iguales dentro de k8s

2.6. Services

- Permite gestionar las comunicaciones entre los pods.
- El acceso a los pods permite gestionar su nombre para acceder a distintos pods, aunque algun pod sea sustituido por una nueva versión, o se caiga y se recrea con otra ip.
- Se gestiona a partir de Labels o etiquetas.
- Un pod puede tener cuantas etiquetas como queramos.
- Si las etiquetas que posee un servicio las poseen pods, el servicio se encarga de balancear entre todas las etiquetas.
- Las etiquetas son dinámicas, luego si usamos una etiqueta de versión, podemos cambiarla por la versión superior para balancear peticiones solo a los que posean exactamente las etiquetas que posee
- Solo envia información a pods saludables
- Se puede definir una afinidad de sesión
- Podemos dirigir el tráfico fuera del cluster
- Balanceo de carga aleatorio.
- Por defecto usa TCP pero puede usar UDP

2.7. ReplicationControllers

- Permite escalar PODs
- Definición de estado deseado
- Sencillo pero poco potente

2.8. Deployments

- Contiene las funcionalidades de los ReplicationControllers
- Está autodocumentado
- Permite desplegar múltiples componentes
- Versionable
- Permite realizar actualizaciones

- Despliegues via apiserver
- Manifiestos Yaml o JSON
- Permite múltiples versiones concurrentes!
- Rollbacks simples

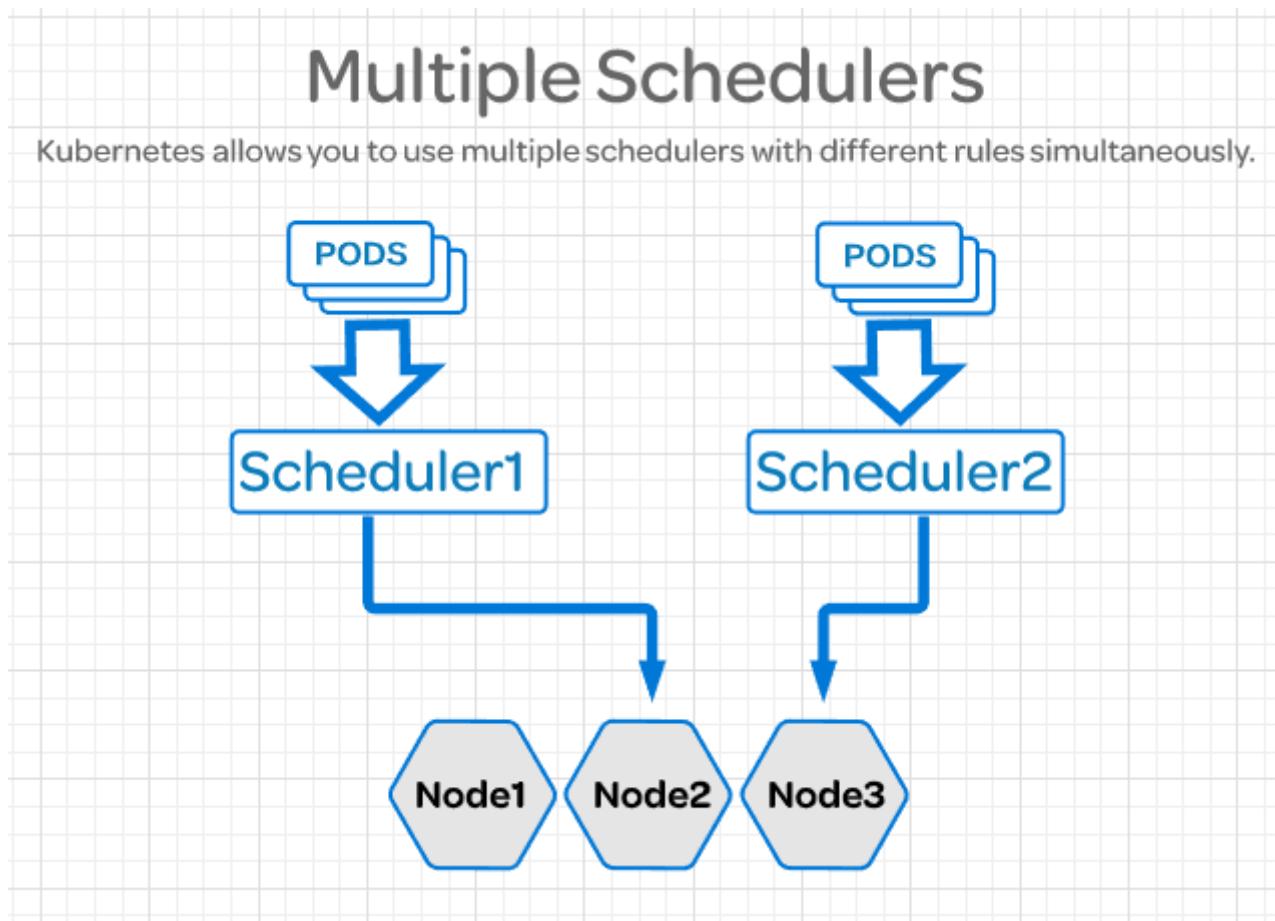


Capítulo 3. Arquitectura KubeScheduler

En el conglomerado de componentes que la arquitectura de kubernetes posee, el kube-scheduler, es el encargado entre otras tareas de:

- Monitorizar el kube-apiserver ante los nuevos trabajos que vayan entrando
- Asignar Pods a cada nodo
- Calcular las reglas de Afinidad, para determinar si un Pod debe de caer en un nodo o en otro
- Tener constancia de las métricas de los recursos de los nodos, para determinar si es posible alojar un Pod en un nodo en cuestión
- Etc.

Por defecto, se utiliza un scheduler por defecto que trae de serie el propio kubernetes, pero tenemos opción de utilizar varios schedulers con diferentes reglas de aplicación y puesta en marcha de manera simultánea



En la siguiente web de referencia, podemos encontrar información detallada al respecto:

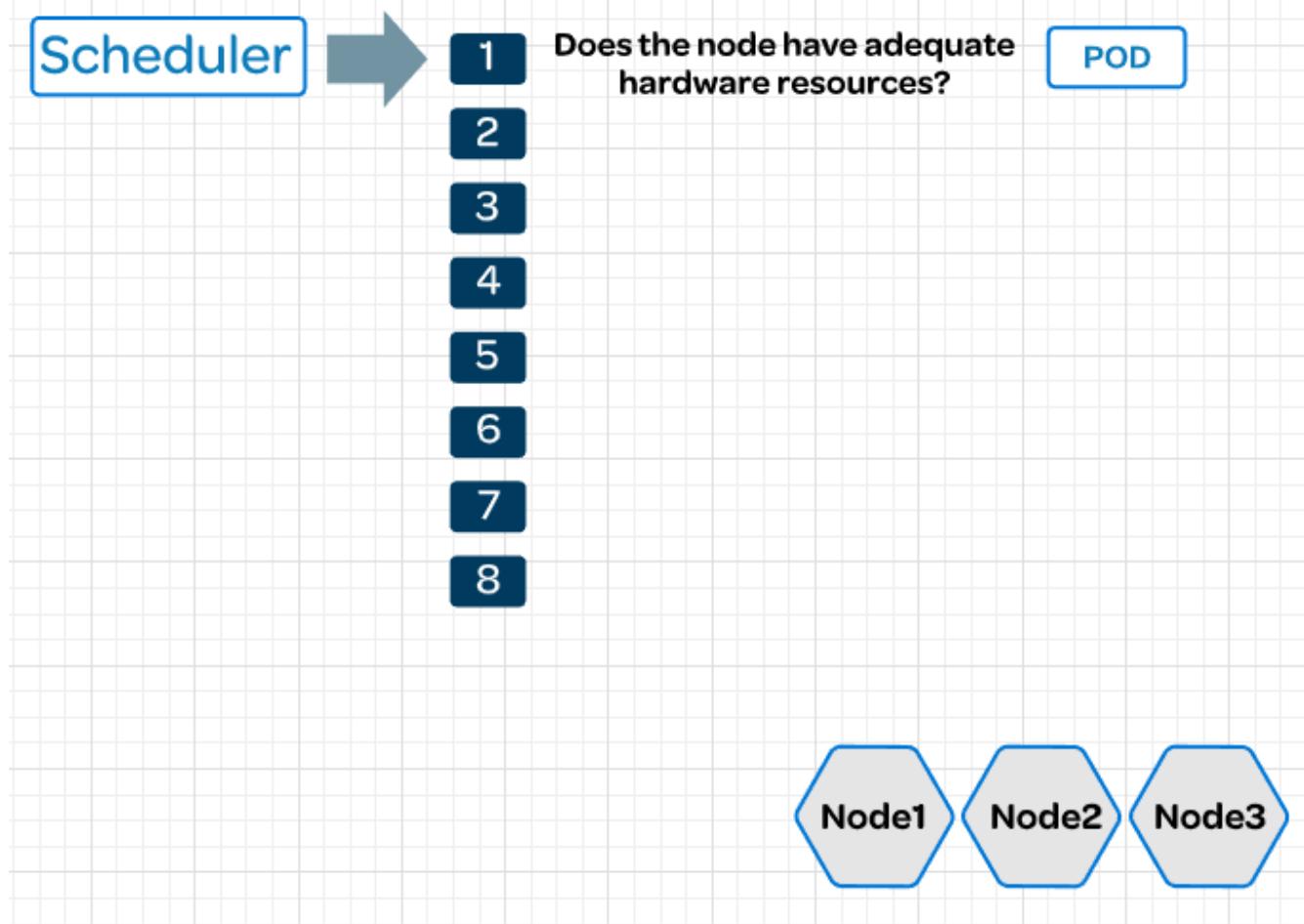
<https://kubernetes.io/docs/tasks/administer-cluster/configure-multiple-schedulers/>

Vamos a desglosar los pasos que sigue o etapas que tiene en cuenta el kube-scheduler por defecto:

3.1. Paso 1: Análisis de recursos hardware

The Default Scheduler

The default scheduler goes through a series of steps to determine the right node for the pod.



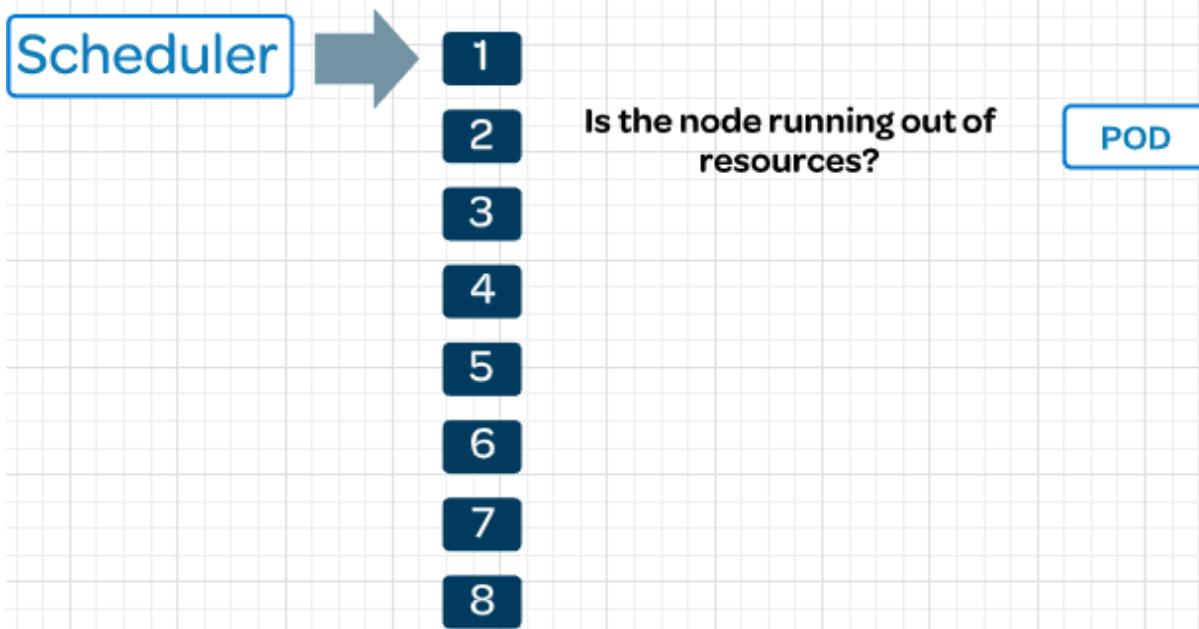
El primer paso que lleva a cabo el kube-scheduler, es determinar si el nodo dispone de recursos hardware suficientes para poder alojar el Pod

Es decir, si el nodo a nivel de características hardware, cumple lo necesario

3.2. Paso 2: Análisis de recursos libres

The Default Scheduler

The default scheduler goes through a series of steps to determine the right node for the pod.

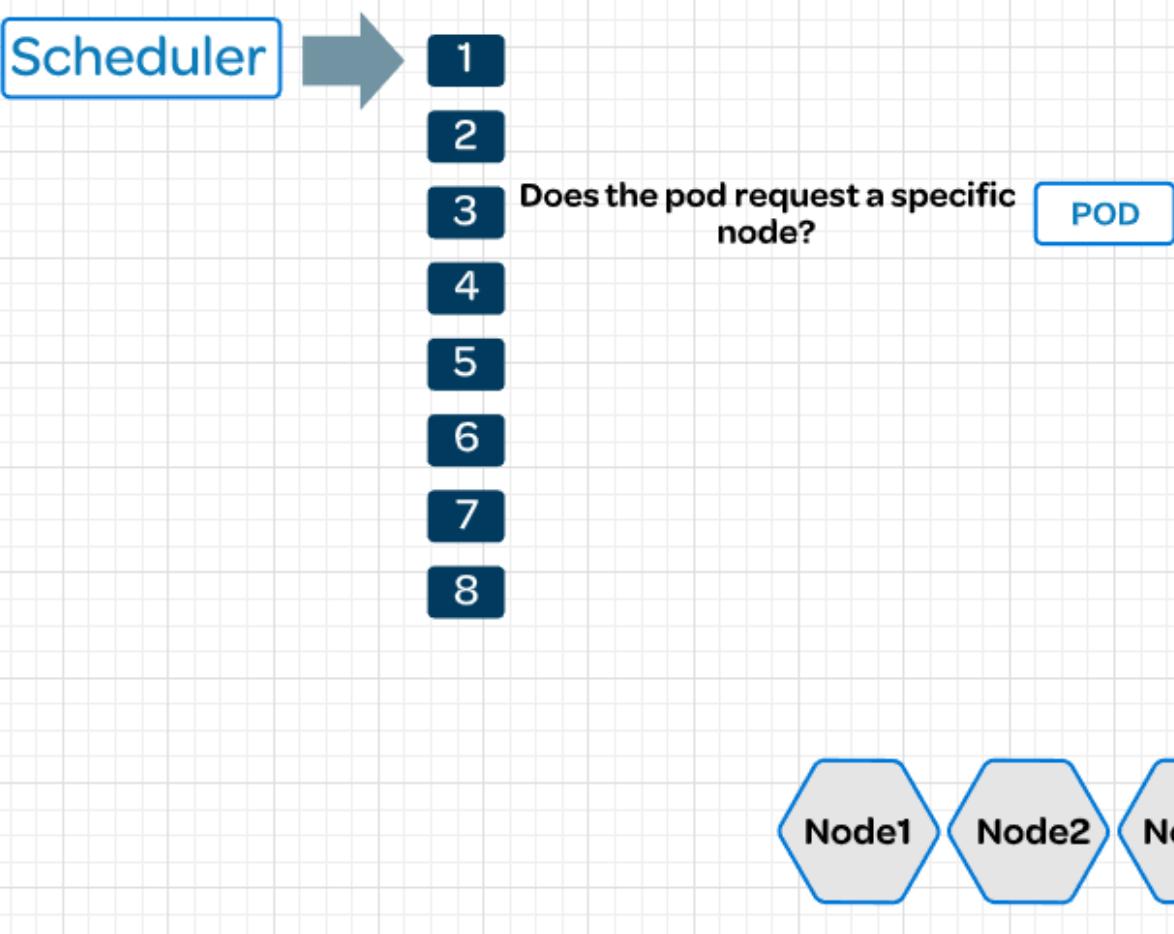


Una vez los recursos hardware han sido analizados, el siguiente paso, es llevar a cabo un análisis de los recursos libres disponibles en el nodo en cuestión, para determinar si caben los Pods con lo que ahora mismo hay libre en el nodo

3.3. Paso 3: Análisis de nodo específico

The Default Scheduler

The default scheduler goes through a series of steps to determine the right node for the pod.

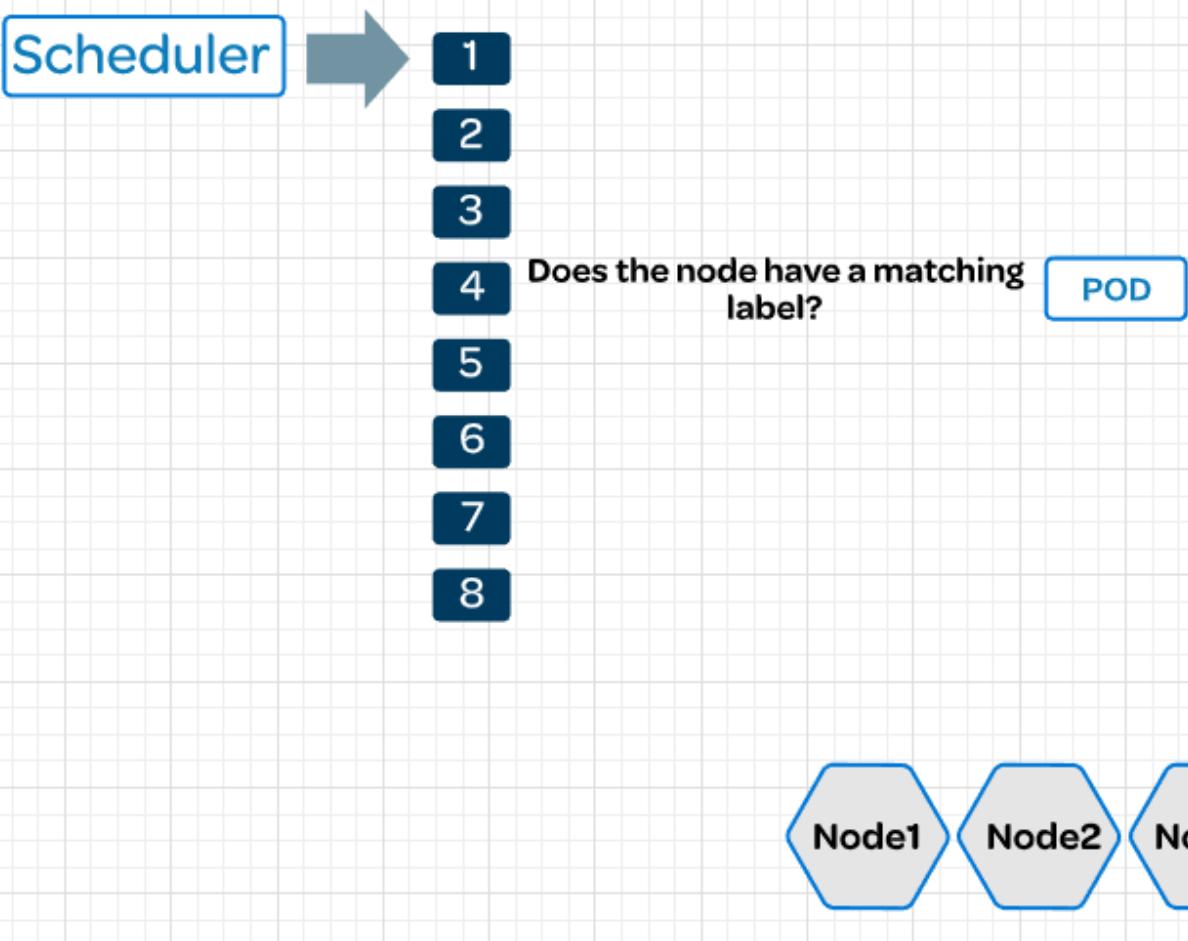


A continuación, se realiza un análisis de especificidad del nodo, para comprobar aspectos como el nombre del nodo, la misión aquí, es determinar si el nodo cumple reglas de especificaciones de nombre si las hubiera, para permitir su puesta en ejecución

3.4. Paso 4: Análisis de etiquetas

The Default Scheduler

The default scheduler goes through a series of steps to determine the right node for the pod.

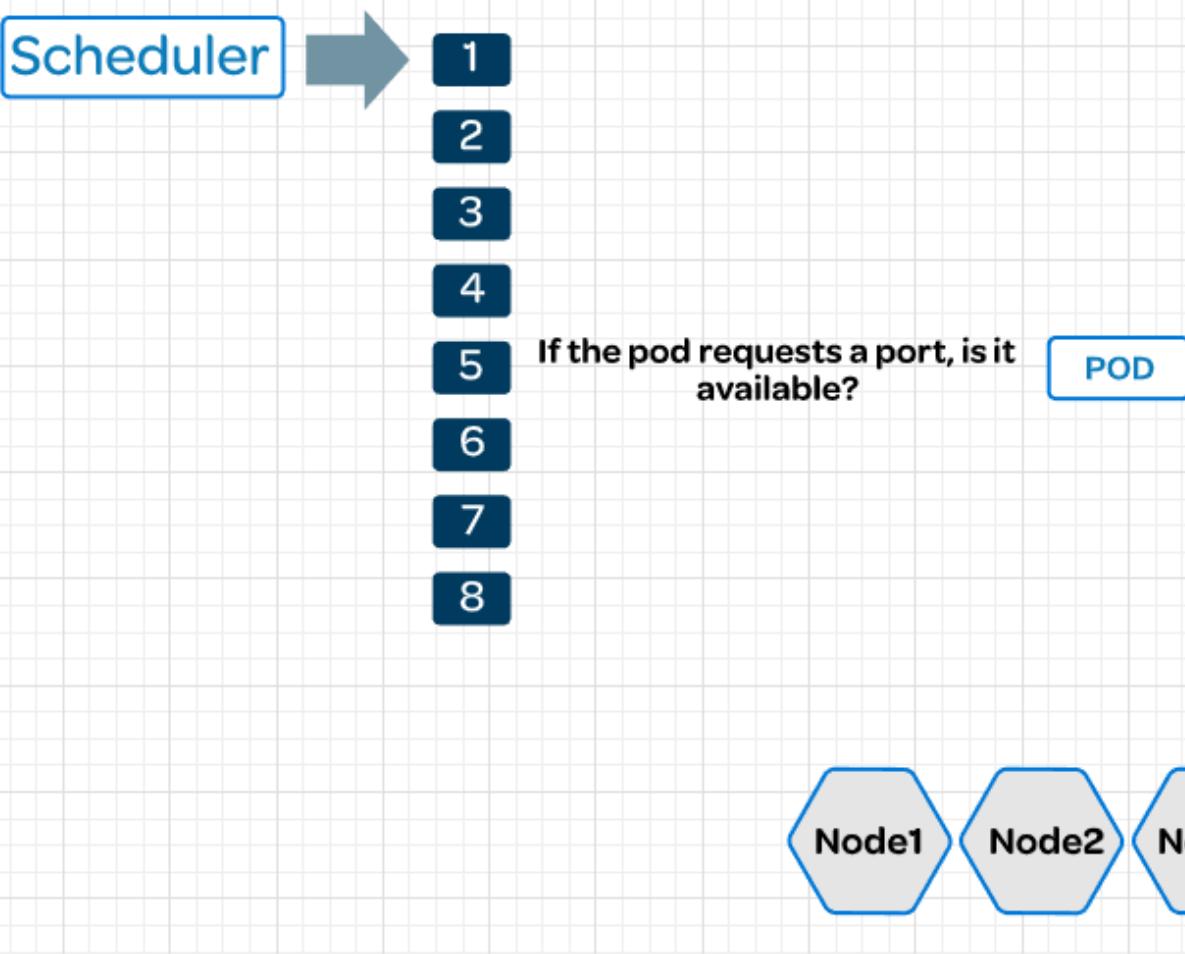


Seguidamente, se lleva a cabo el análisis de las etiquetas del nodo, de forma que se cumpla si especificamos, la regla de etiquetado para que sea alojado

3.5. Paso 5: Análisis de red

The Default Scheduler

The default scheduler goes through a series of steps to determine the right node for the pod.

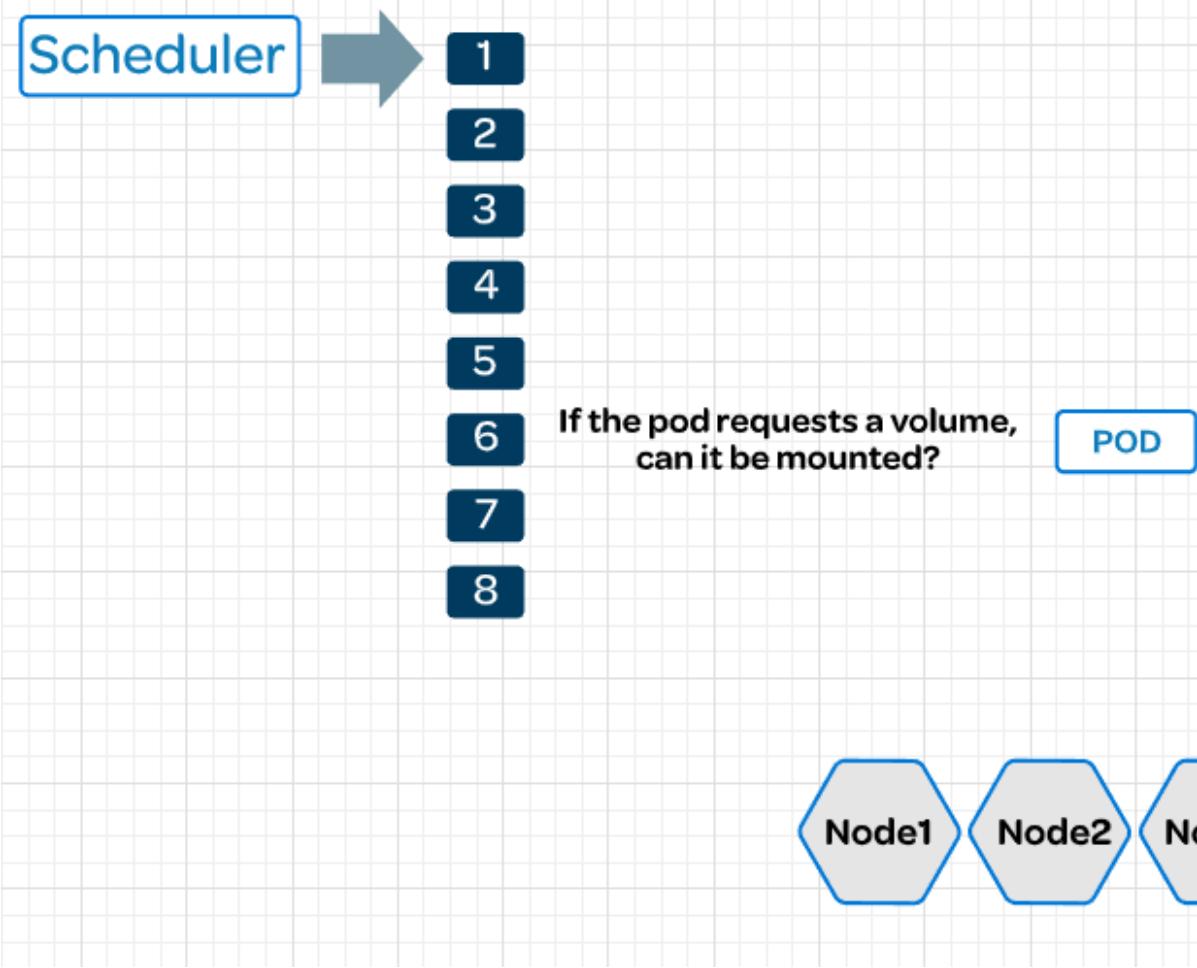


En este paso, se analizan recursos de red, como por ejemplo, que haya puertos disponibles para usar en el host, cuando estamos indicando binding de puertos hacia el anfitrión

3.6. Paso 6: Análisis de volúmenes

The Default Scheduler

The default scheduler goes through a series of steps to determine the right node for the pod.

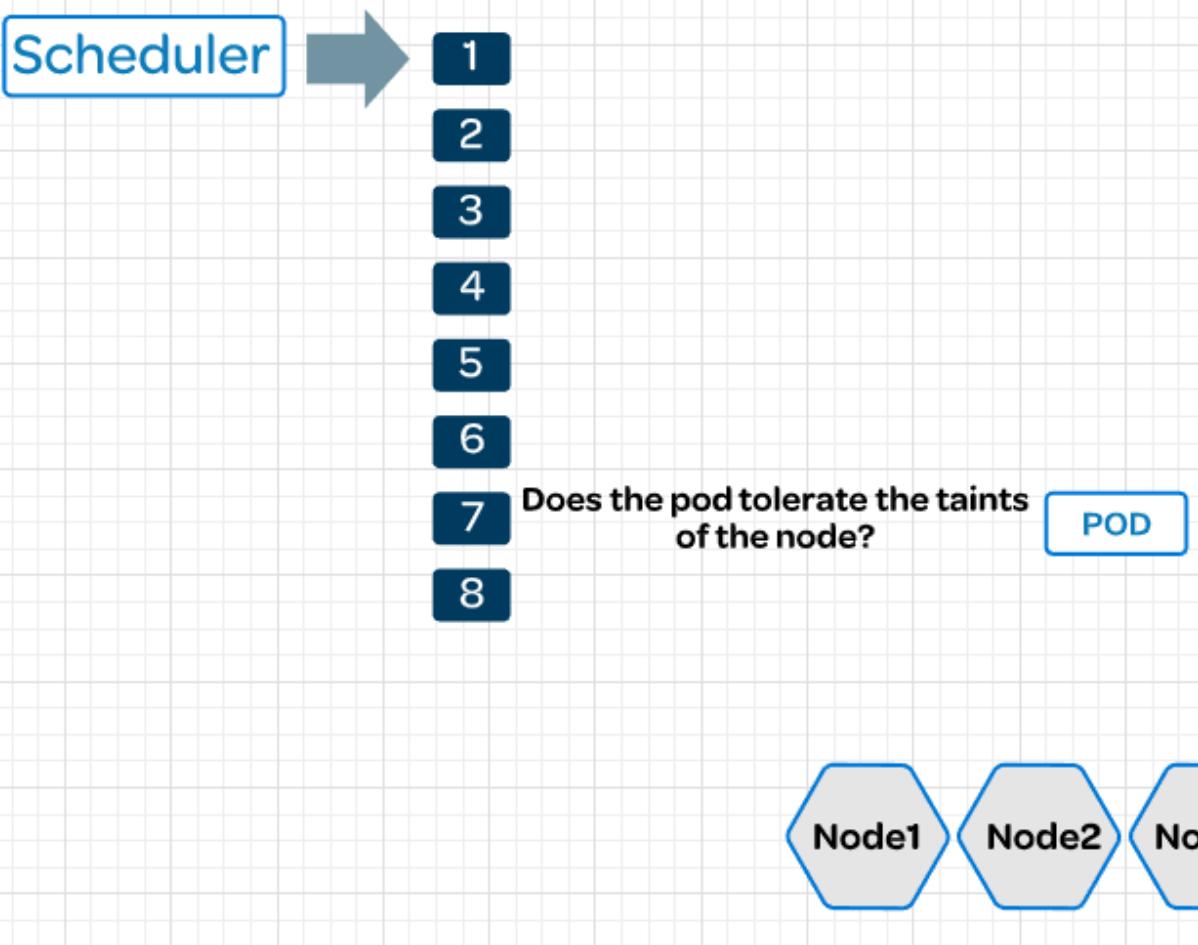


En este paso, se analiza si el Pod requiere de volúmenes, si pueden ser montados o no en el nodo, si pueden ser bindeados al Pod, etc.

3.7. Paso 7: Análisis de la tolerancia

The Default Scheduler

The default scheduler goes through a series of steps to determine the right node for the pod.

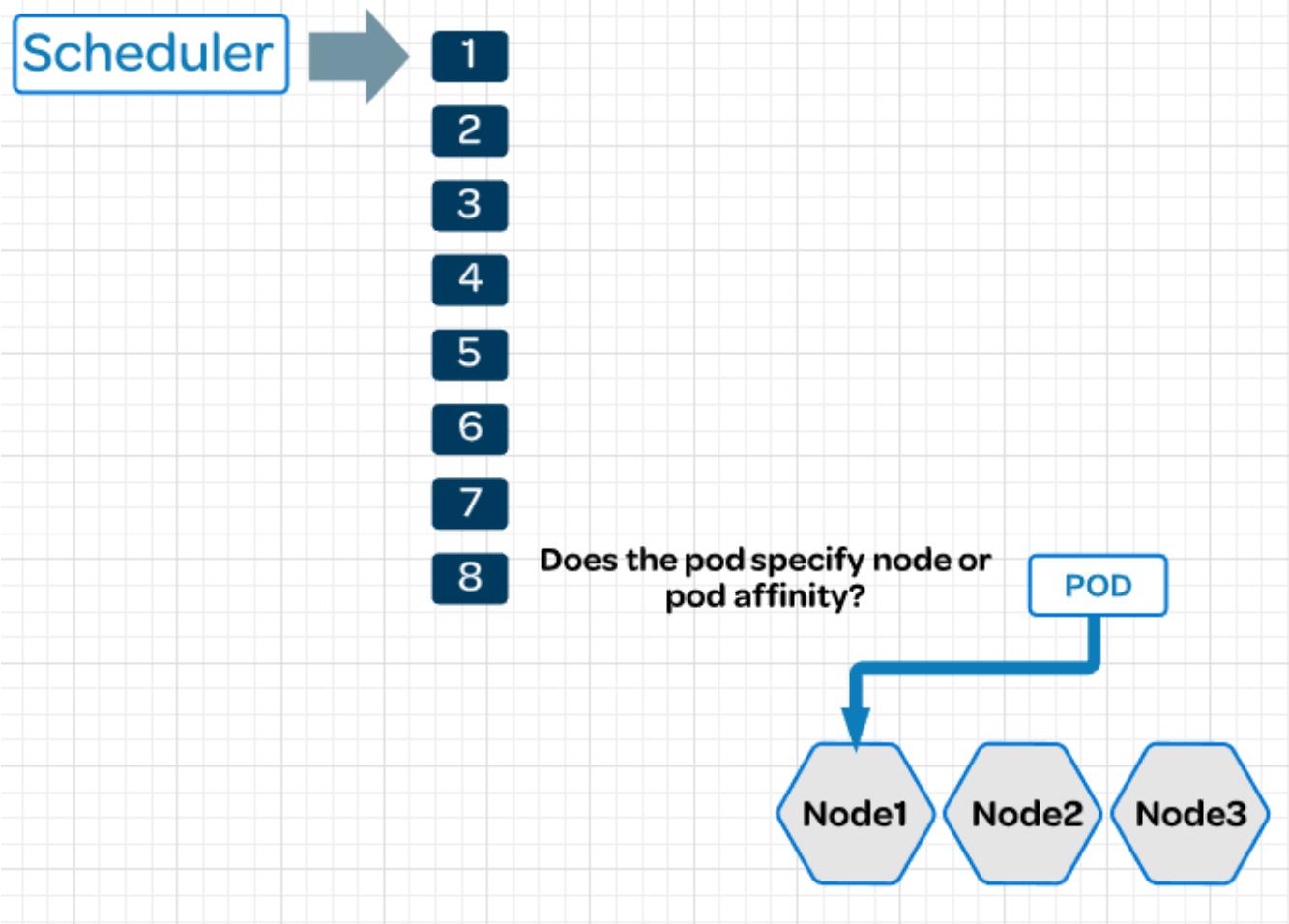


Ahora, se analiza el tema de la tolerancia, de forma que se analiza si el Pod tolera las "manchas" (taints) que el nodo tenga en ese momento

3.8. Paso 8: Análisis de la afinidad

The Default Scheduler

The default scheduler goes through a series of steps to determine the right node for the pod.



El último paso, consiste en analizar lo que se denomina la afinidad del nodo, de forma que cuando especificamos afinidad de nodos, pudiera darse que hubiera como candidatos más de uno, por lo que el kube-scheduler realizará un cálculo de medida ponderada para determinar el mejor nodo que cumple la especificación

Ante un empate técnico en los cálculos, elegirá uno de los candidatos de forma aleatoria

El kube-scheduler para determinar esta decisión sigue el principio del algoritmo Round-Robin

Capítulo 4. API Primitives

El Kube API Server es el único que se comunica con el etcd data store

Todos los componentes de Kubernetes se comunican con el etcd mediante el API Server, para modificar el estado final de cada operación que se realiza

La herramienta kubectl nos permite comunicarnos con el API Server, de manera que podemos crear, modificar o eliminar objetos desplegados en el clúster de kubernetes

El API Server expone los recursos de la API, también llamados componentes de estados

El siguiente comando mostrará todos los componentes del "Control Plane" y el estado de cada uno de ellos:

```
$ kubectl get componentstatus
```

NAME	STATUS	MESSAGE	ERROR
scheduler	Healthy	ok	
controller-manager	Healthy	ok	
etcd-0	Healthy	{"health":"true"}	

Observamos los siguientes componentes:

- El estado del kube-scheduler
- El estado del kube-controller-manager
- El estado del data store etcd

4.1. Manifiestos YAML

Kubernetes está continuamente monitoreando el total del sistema, para que el estado deseado en nuestros manifiestos, sea el que le hemos indicado

Los manifiestos, los escribimos en formato YAML, mediante una sintaxis declarativa, indicando la extensión .yml a los archivos en cuestión

El cliente kubectl lleva a cabo una conversión del formato YAML a formato JSON

La identación del archivo, utiliza 2 espacios, no tabulador :D

Cada línea del archivo contiene un valor

Los valores después de los 2 puntos significan que las instrucciones de abajo, tendrán a su vez una doble identación, para asociar el valor, en formato pares "key: value"

El siguiente ejemplo, muestra una unidad de despliegue de tipo **Deployment** que ilustra los puntos que hemos comentado:

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.9
          ports:
            - containerPort: 80

```



4.2. Estructura básica de un manifiesto YAML

Cualquier manifiesto YAML debe de tener una cabecera común, donde le informamos a kubernetes ciertos aspectos de comportamiento y estructura del objeto

- **apiVersion:** Indicamos la versión del API de kubernetes que deseamos implementar
 - Dependiendo de la versión de kubernetes en nuestro cluster, podremos utilizar una versión u otra del API
- **kind:** Indicamos el tipo de objeto que queremos poner en marcha en Kubernetes
- **metadata:** Aquí indicamos los elementos necesarios que de forma única, identificarán al Objeto, en nuestro caso, indicamos como nombre del objeto **nginx-deployment**
 - La cadena de caracteres que indiquemos en la propiedad **name**, no puede superar los 253 caracteres de longitud
 - Paralelamente al nombre del Objeto, kubernetes generará un campo con nombre **uid**, que estará también bajo el ámbito del bloque metadata, será visible, si una vez desplegado el objeto, indicamos un comando de obtención de YAML a partir del objeto desplegado
 - Los campos uid, así como los valores correspondientes, le compete generarlos a kubernetes, como operadores del clúster, no es nuestro cometido encargarnos de esa tarea
 - Por consiguiente, un objeto en kubernetes, siempre tiene asignado de forma única 2 elementos **name** y **uid**



En un clúster, no puede haber 2 objetos que se llamen exactamente igual, lo que si podremos tener es varios objetos que se llamen igual, pero cada uno en lo que se conoce como un **namespace** diferente



4.3. Ventajas del uso de manifiestos YAML

- Dejamos constancia de las maniobras de despliegue que estamos llevando a cabo en el clúster
- Podemos versionar los despliegues, de forma que podemos guardar nuestros manifiestos en un sistema de control de versiones, tipo GIT
- Con la herramienta kubectl podemos llegar a crear objetos en kubernetes, pero ni dejamos constancia de los mismos, y por otra parte, el conjunto de instrucciones que podemos indicar al kube-api-server es más reducido que vía manifiestos

4.4. Obtención de YAML de objetos desplegados en kubernetes

Otra opción interesante que podemos realizar con kubernetes, relacionado con el API Server, es la obtención de estructura YAML a partir de cualquier objeto que se encuentre desplegado en el clúster de kubernetes

La siguiente instrucción a modo genérico nos permitiría llevarlo a cabo:

```
$ kubectl get <KUBERNETES_TYPE_OBJECT> <KUBERNETES_OBJECT> -o yaml
```

- **KUBERNETES_TYPE_OBJECT**: El tipo de Objeto que queremos consultar con la API Server
 - pod, secret, rs, deployment, etc.
- **KUBERNETES_OBJECT**: El nombre en sí del elemento
 - pod-frontend-1
 - deployment-frontend
 - Etc.

Capítulo 5. Node

En el mundo de kubernetes, un nodo puede ser una máquina física o virtual, en definitiva una instancia de Docker Engine que da soporte a la ejecución de los contenedores de orquestación de kubernetes

Los programadores si están inmersos en los circuitos de integración continua, o bien disponen de acceso a algún entorno de despliegue con kubernetes, normalmente el tema del nodo no es algo que les preocupa o que se encarguen de administrar

Por otro lado, el personal de administración, debe de estar perfectamente familiarizado con la gestión de los nodos, ya que la gestión de los mismos se hace fundamental para asegurar el buen funcionamiento del clúster

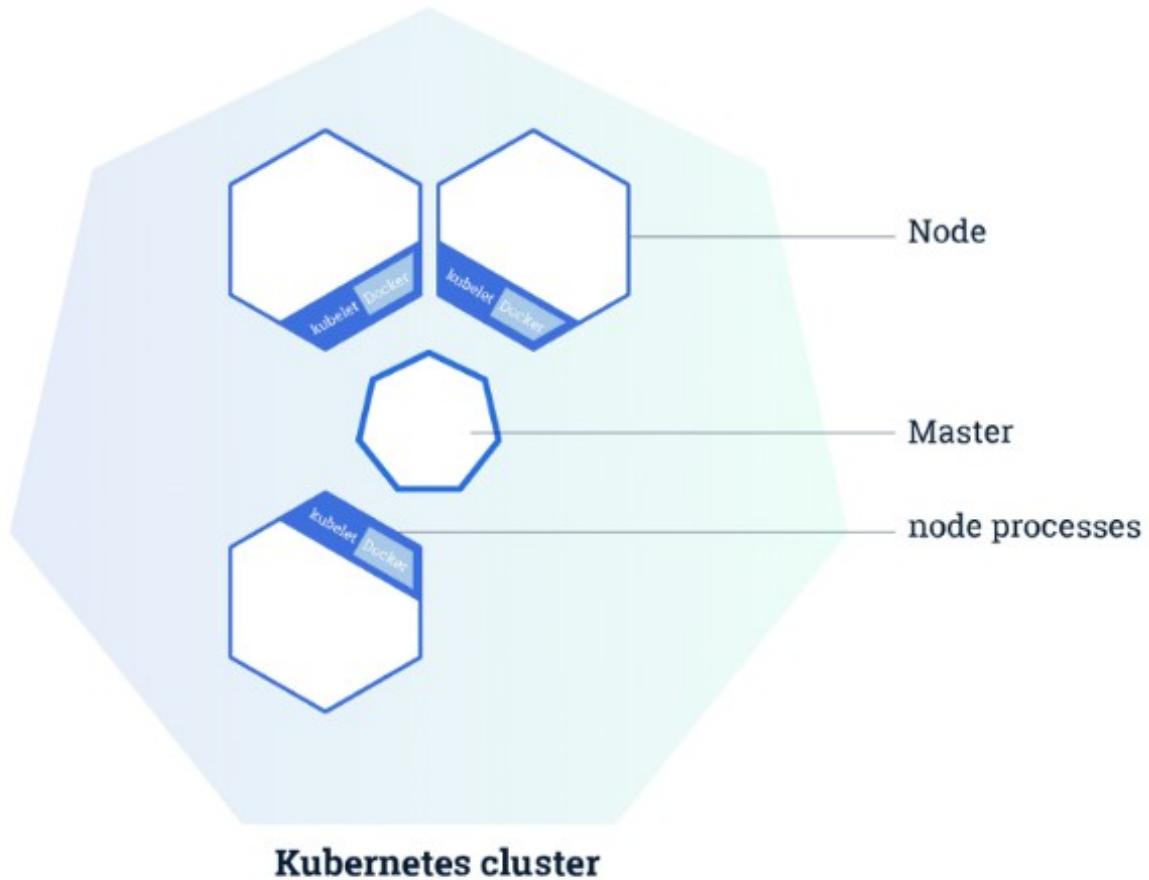
Dependiendo de la topología que decidamos, podemos llegar a disponer de varios tipos:

- **Single-node cluster**

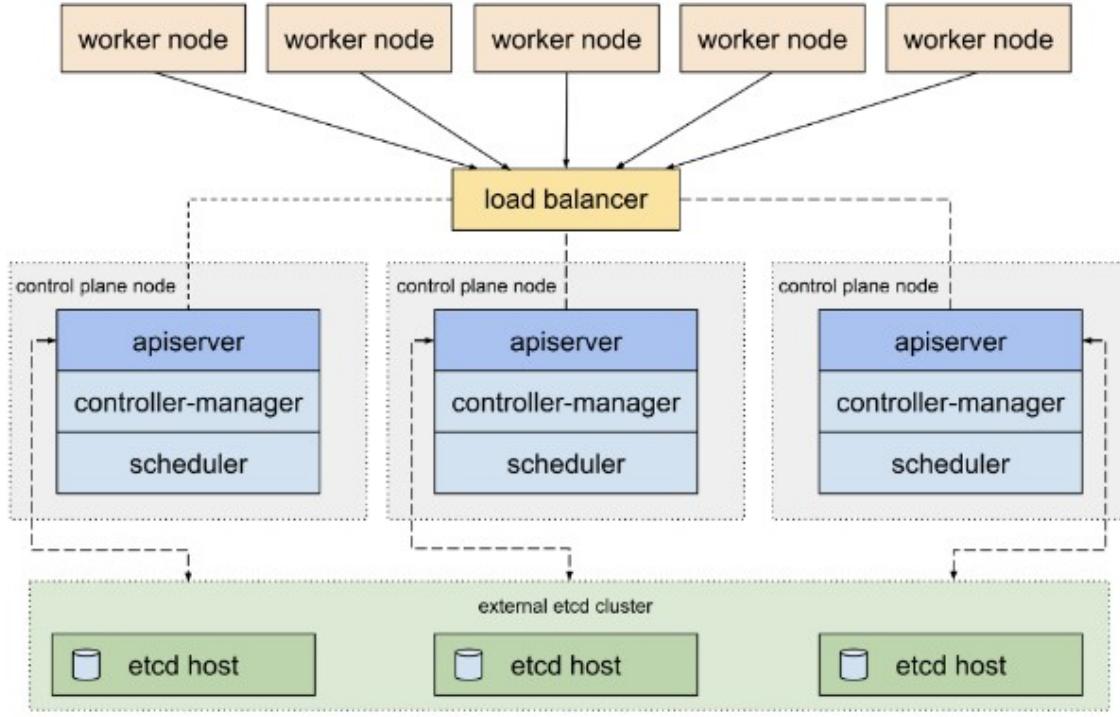
- Utilizando la implementación minikube
- Es ideal para entornos de aprendizaje básico, y familiarizarnos con todos los conceptos que forma parte de kubernetes
- Disponemos de un único nodo donde ahí se encuentra todo, tanto los elementos necesarios que kubernetes necesita para su fucionamiento como los sistemas que vayamos a desplegar
- No está recomendado su uso en producción
- URL: <https://github.com/kubernetes/minikube>

- **Single-master, multi-node cluster**

- Este entorno ya si es viable de utilizar en producción
- Consta de un nodo maestro y múltiples nodos separados denominados "minions"



- **HA (High-Availability) Cluster**
 - Este es un entorno complejo basado en la alta disponibilidad
 - Para que sea efectivo en operatividad, necesitaríamos (mínimo)
 - 3 Nodos maestros
 - 3 nodos a modo de minions
 - 3 Nodos con la custodia de la información del clúster etc



URL: <https://kubernetes.io/docs/setup/independent/ha-topology/>

5.1. Requisitos Hardware

Basándonos en la documentación oficial de kubernetes

URL: <https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/#before-you-begin>

Cada nodo del clúster debería de tener al menos 2 CPUs y 2 GB de RAM como mínimo

Dependiendo de las necesidades y características Software que queramos poner en marcha, estos requisitos mínimos para nuestros servidores podrán variar considerablemente

5.2. Requisitos Software

Como distribuciones base para nuestros servidores, serían muy buenas opciones:

- Alguna distribución basada en la familia Ubuntu Server, la última LTS que haya disponible
 - Mínimo de la 16.04 LTS en adelante
- Alguna distribución basada en la familia CentOS
 - Mínimo una CentOS 7.5+



Es preferible utilizar una distribución cuyo Kernel esté actualizado lo máximo posible, como sabemos CentOS tendría el soporte de IBM ya que RedHat fue absorbida por esta, pero... Estamos hablando que tendríamos un Kernel estancado en una versión 3.10... Con todo lo que ello implica en la seguridad con todas las nuevas actualizaciones y parches que han ido saliendo

Independientemente de la distribución a utilizar, el funcionamiento y operatividad del clúster Kubernetes es exactamente el mismo

5.3. Requisitos de Red

Una vez tengamos clara la topología deseada, así como mínimos requisitos software y hardware necesarios, llega el momento de pensar en la red

Un clúster de kubernetes cuando se instala está sin driver DNS, debemos de elegir instalar un CNI (Container Networking Interface)

Si no instalamos ningún driver de red, el clúster sencillamente no funcionará :)

Nosotros utilizaremos para nuestro clúster el driver **WeaveNet** pero hay múltiples en el mercado, con diferentes características



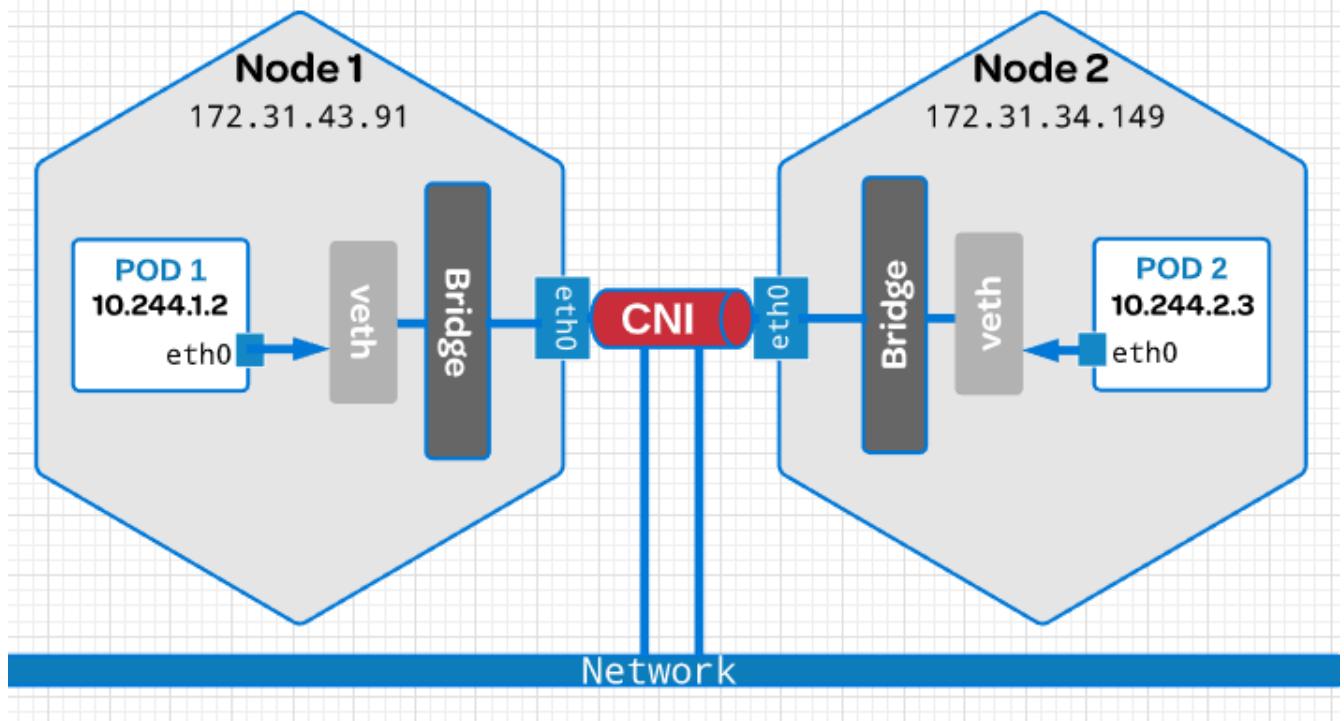
5.4. Tipos de Nodos

En una arquitectura de clúster siempre vamos a tener 2 tipos de nodos, nodos maestros y nodos minions

Capítulo 6. CNI (Container Network Interface)

Network Overlay

A CNI goes on top of your existing network, allowing us to essentially build a tunnel between nodes.



El soporte CNI, se sitúa encima de la red existente del propio servidor, de manera que nos permite construir esencialmente un túnel entre nodos, o también llamado, una red distribuída entre diferentes servidores.

Cuando hablamos del componente CNI, estamos hablando de las comunicaciones que se producen de nodo a nodo

Observamos en el diagrama los puntos de enlace de los diferentes servidores mediante el mecanismo de red distribuidía CNI

Cuando se conectan contenedores a la red, en realidad lo que está sucediendo es un encapsulamiento de los paquetes de red

En dichos paquetes, se introduce un encabezado encima del paquete, que cambia tanto la fuente como el destino del paquete que va al pod 1 desde el pod2 y desde el pod 2 hacia el pod 1,

Esto es así para poder determinar a nivel interno del clúster, el pod del que originalmente viene el paquete y hacia el que va dirigido, estando en una red de ámbito distribuído

El soporte CNI, establece un mapeo asociado en el espacio de nombres de usuario, donde se programan todas las direcciones IP de los Pods, hacia el nodo/pod al que van dirigidos los paquetes

Cuando el paquete alcanza el nodo de destino, el driver de red, desencapsula el paquete y lo entrega al puente de red "Bridge" del nodo en cuestión

Los drivers de red, no son soluciones nativas de kubernetes, debiendo de tirar de terceros, existen drivers como waveworks, flannel, calico, romana, etc.

La instalación de dichos drivers, también llamados plugins, se lleva a cabo vía manifiesto, con la herramienta de consola kubectl

Por ejemplo, si quisiéramos instalar el driver de flannel, ejecutaríamos en la consola la siguiente instrucción:

```
$ kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/bc79dd1505b0c8681ece4de4c0d86c5cd2643275/Documentation/kube-flannel.yml
```

Cuando se instala el driver de red, se instala un agente (pod) en cada nodo, el cual establece enlaces de red con el componente de interfaz CNI de kubernetes

Dependiendo del driver de red que queramos utilizar, tenemos que llevar a cabo cierta configuración.

En la siguiente página de la documentación oficial de kubernetes podemos echarle un vistazo a qué ajustes necesitamos utilizar dependiendo de qué driver queramos utilizar

```
https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/#pod-network
```

También podemos consultar la documentación oficial de kubernetes sobre la extensión de la funcionalidad del mismo, mediante la instalación de plugins:

```
https://kubernetes.io/docs/concepts/cluster-administration/addons/
```

El plugin que instalaremos, será el responsable de la asignación de IP interna a cada Pod que se crea en el clúster, qué IP ha dejado de usarse, cual se puede reasignar a otro Pod, etc.

Capítulo 7. Lab: Ajustes en VirtualBox (Clúster Kubernetes OnPremise)

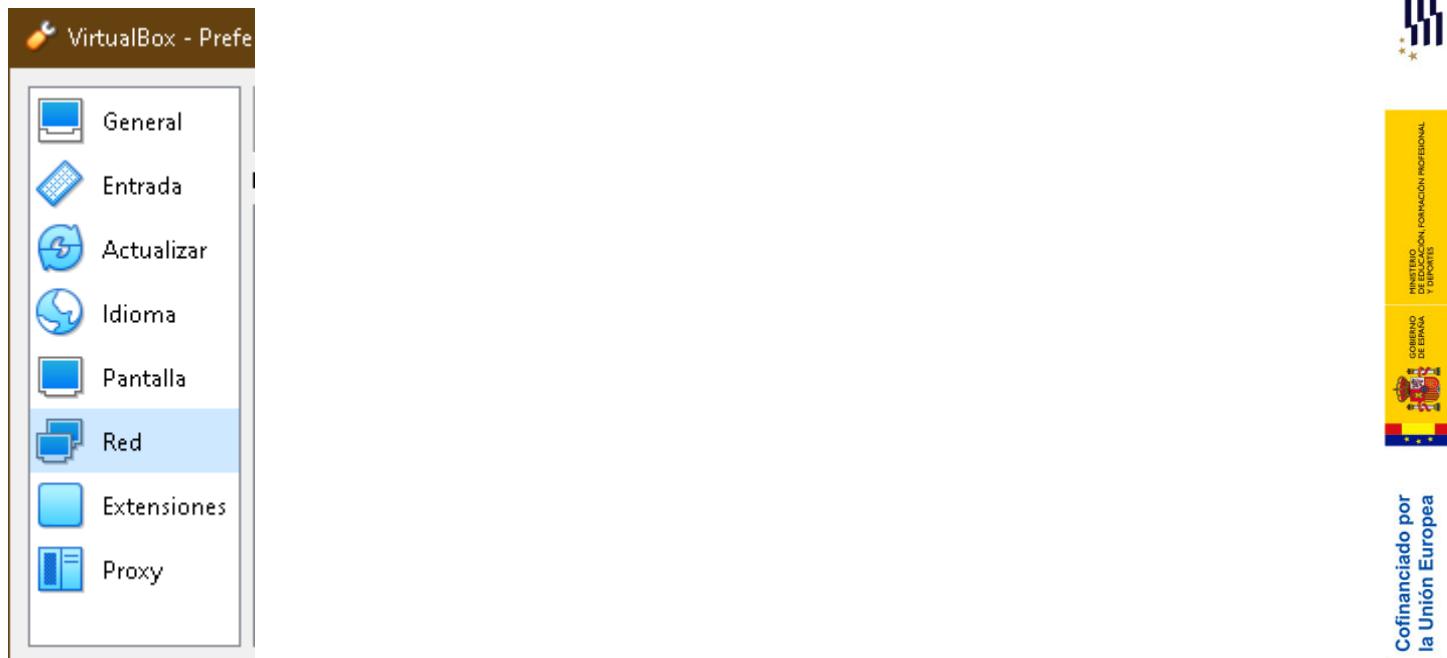
Mediante este laboratorio, vamos a llevar a cabo los ajustes de red necesarios para poder tener operativo un clúster de kubernetes de forma local en nuestra infraestructura mediante el hypervisor VirtualBox.

7.1. Creando la Red Nat

Desde el propio VirtualBox, accedemos al menú Archivo > Preferencias



Seguidamente, accedemos a la opción de Red



Situar el ratón encima del ícono y aparecerá el texto emergente **Agregar nueva red NAT**, pulsamos sobre la opción.

Indicamos como nombre de red **kubenetwerk**



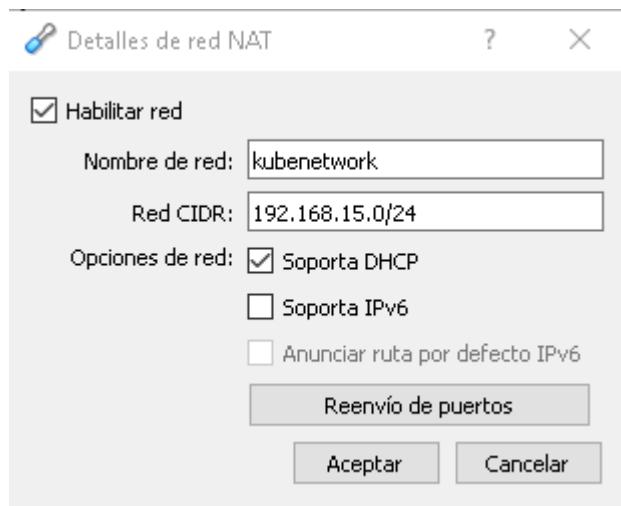
Pulsamos sobre el ícono del lateral derecho, donde aparece una rueda dentada de configuración.

Indicamos la siguiente configuración:

- Habilitar red

- Marcamos el check
- Nombre de red
 - kubenetwerk
- Red CIDR
 - 192.168.15.0/24
- Soporta DHCP
 - Marcamos el check

El resto de opciones las dejamos como están y pulsamos **Aceptar**



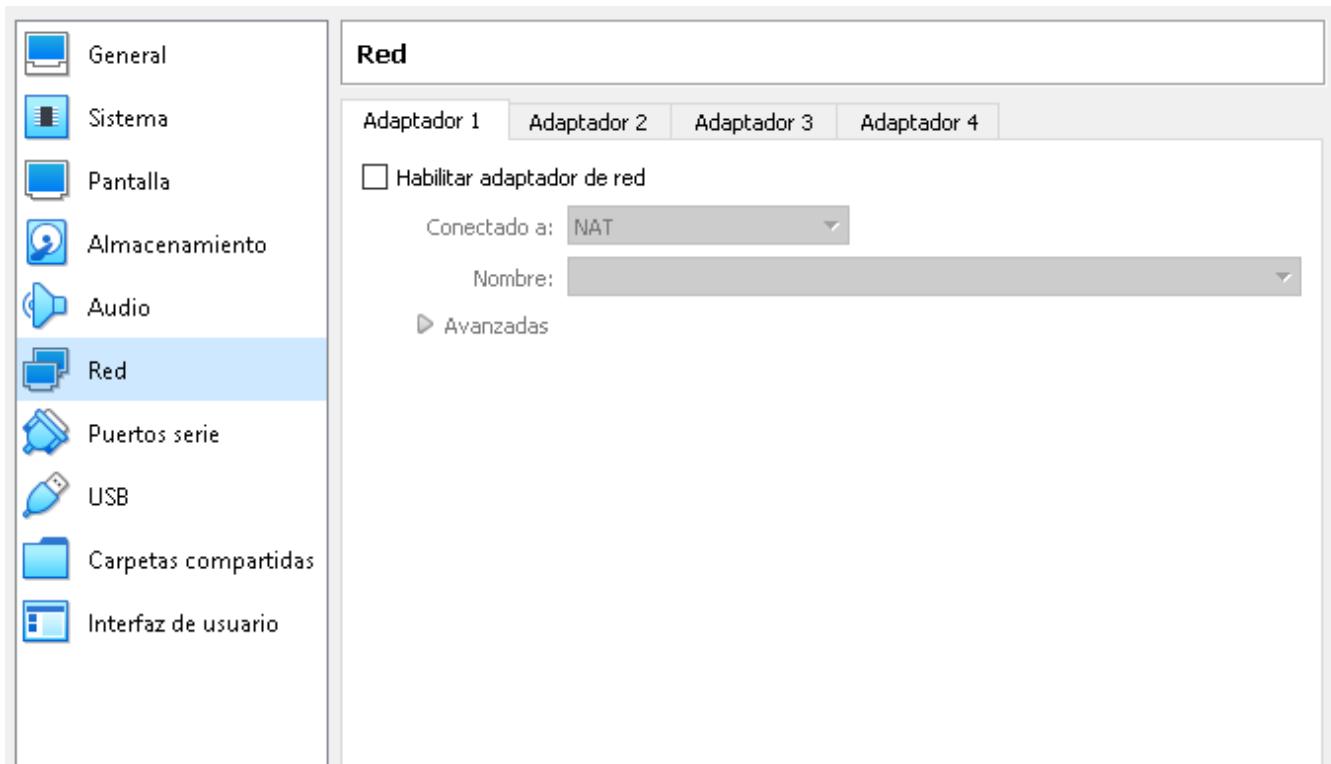
7.2. Asignando la red a las máquinas virtuales

Por cada máquina virtual que forme parte del clúster, debemos de realizar los pasos en el ajuste de red que vamos a indicar.

Una vez la máquina virtual esté cargada en virtualbox, mediante la opción de **Máquina > añadir**, pulsamos sobre cada máquina y luego pulsamos sobre el botón de **Configuración**.

Vamos al apartado de **Red**

El **Adaptador 1** lo desactivamos:



Pulsamos sobre el **Adaptador 2** y ajustamos las siguientes opciones:

- Habilitar adaptador de red
 - Lo marcamos
- Conectado a
 - Red NAT
- Nombre
 - kubenetwor
 - Seleccionamos del combo la red que previamente hemos creado en VirtualBox
- Tipo de adaptador
 - Red paravirtualizada (vitrionet)
- Modo promiscuo
 - Permitir todo
- Dirección MAC
 - Pulsamos varias veces sobre el icono de regeneración, para regenerar dicha MAC
- Cable conectado
 - Marcamos el check

General
Sistema
Pantalla
Almacenamiento
Audio
Red
Puertos serie
USB
Carpetas compartidas
Interfaz de usuario

Red

Adaptador 1 Adaptador 2 Adaptador 3 Adaptador 4

Habilitar adaptador de red

Conectado a: Red NAT

Nombre: kubenetwerk

Avanzadas

Tipo de adaptador: Red paravirtualizada (virtio-net)

Modo promiscuo: Permitir todo

Dirección MAC: 080027FD710E

Cable conectado

Reenvío de puertos

Aceptar Cancelar

Pulsamos **Aceptar**

Capítulo 8. Motor de contenedores CRI-O



8.1. ¿Qué es CRI-O?

CRI-O es una implementación de Kubernetes CRI (Container Runtime Interface) para permitir el uso de tiempos de ejecución compatibles con OCI (Open Container Initiative).

Es una alternativa ligera al uso de Docker como runtime de ejecución para kubernetes.

CRI-O admite imágenes de contenedor OCI y puede extraerlas de cualquier registry de imágenes.

Es una alternativa ligera al uso de Docker, Moby o rkt como runtime de ejecución de contenedores para Kubernetes.

8.2. ¿Quién está contribuyendo al proyecto?

Detrás del proyecto hay contribuidores como:

- RedHat
- Intel
- SUSE
- Hyper
- IBM

CRI-O es desarrollado por colaboradores de estas y otras empresas.

Es un proyecto de código abierto impulsado por la comunidad.

Los comentarios, los usuarios y, por supuesto, los contribuyentes, siempre son bienvenidos a través del proyecto en GitHub: <https://github.com/cri-o/cri-o>

Podemos consultar la siguiente URL para más información: <https://cri-o.io/>

Capítulo 9. Lab: Instalación del motor de contenedores CRI-O (Centos7)

Mediante este laboratorio procederemos a llevar a cabo la instalación del motor de contenedores CRI-O en los nodos del clúster de kubernetes.

9.1. Instalación del runtime CRI-O



Este procedimiento se realizará tanto en el master como en los minions

Creamos en primer lugar el archivo de carga de módulos del propio runtime, para que se cargen al inicio del sistema.

```
$ cat <<EOF | sudo tee /etc/modules-load.d/crio.conf
overlay
br_netfilter
EOF
```

Seguidamente agregamos los siguientes módulos al kernel:

```
$ sudo modprobe overlay
$ sudo modprobe br_netfilter
```

Ahora, configuraremos las reglas de ip tables para que sean persistentes ante reinicios del servidor:

```
# cat <<EOF | sudo tee /etc/sysctl.d/99-kubernetes-cri.conf
net.bridge.bridge-nf-call-iptables = 1
net.ipv4.ip_forward = 1
net.bridge.bridge-nf-call-ip6tables = 1
EOF
```

Recargamos la configuración de iptables en caliente sin reiniciar:

```
$ sudo sysctl --system
```

Declaramos las siguientes variables de entorno para indicar, por un lado, la versión del runtime de CRI-O que deseamos utilizar, así como la versión del sistema operativo que disponemos:

```
export VERSION=1.21
export OS=CentOS_7
```

Añadimos los repositorios:

```
$ sudo curl -L -o /etc/yum.repos.d-devel:kubic:libcontainers:stable.repo  
https://download.opensuse.org/repositories/devel:/kubic:/libcontainers:/stable/$OS-devel:kubic:libcontainers:stable.repo  
  
$ sudo curl -L -o /etc/yum.repos.d-devel:kubic:libcontainers:stable:cri-o:$VERSION.repo  
https://download.opensuse.org/repositories/devel:kubic:libcontainers:stable:cri-  
o:$VERSION/$OS-devel:kubic:libcontainers:stable:cri-o:$VERSION.repo
```

Instalamos el siguiente paquete:

```
$ sudo yum install -y yum-utils
```

Instalamos el paquete del runtime:

```
$ sudo yum install cri-o
```

Recargamos los servicios del sistema y habilitamos el runtime para que se inice automáticamente ante próximos reinicios del servidor:

```
$ sudo systemctl daemon-reload  
$ sudo systemctl start crio  
$ sudo systemctl enable crio --now
```



Fondos Europeos



GOBIERNO
DE ESPAÑA
MINISTERIO
DE EDUCACIÓN, FORMACIÓN PROFESIONAL
Y DEPORTE

Cofinanciado por
la Unión Europea



Capítulo 10. Lab: Instalación del motor de contenedores CRI-O (Fedora)

Mediante este laboratorio procederemos a llevar a cabo la instalación del motor de contenedores CRI-O en los nodos del clúster de kubernetes.

10.1. Instalación del runtime CRI-O



Este procedimiento se realizará tanto en el master como en los minions

Creamos en primer lugar el archivo de carga de módulos del propio runtime, para que se cargen al inicio del sistema.

```
$ cat <<EOF | sudo tee /etc/modules-load.d/crio.conf
overlay
br_netfilter
iptable_raw
EOF
```

Seguidamente agregamos los siguientes módulos al kernel:

```
$ sudo modprobe overlay
$ sudo modprobe br_netfilter
$ sudo modprobe iptable_raw
```

Ahora, configuraremos las reglas de ip tables para que sean persistentes ante reinicios del servidor:

```
# cat <<EOF | sudo tee /etc/sysctl.d/99-kubernetes-cri.conf
net.bridge.bridge-nf-call-iptables = 1
net.ipv4.ip_forward = 1
net.bridge.bridge-nf-call-ip6tables = 1
EOF
```

Recargamos la configuración de iptables en caliente sin reiniciar:

```
$ sudo sysctl --system
```

Declararemos las siguientes variables de entorno para indicar, por un lado, la versión del runtime de CRI-O que deseamos utilizar, así como la versión del sistema operativo que disponemos:

```
VERSION=1.22
```

Habilitamos el módulo en el kernel:

```
$ sudo dnf module enable cri-o:$VERSION
```

Instalamos el siguiente paquete:

```
$ sudo dnf install cri-o
```

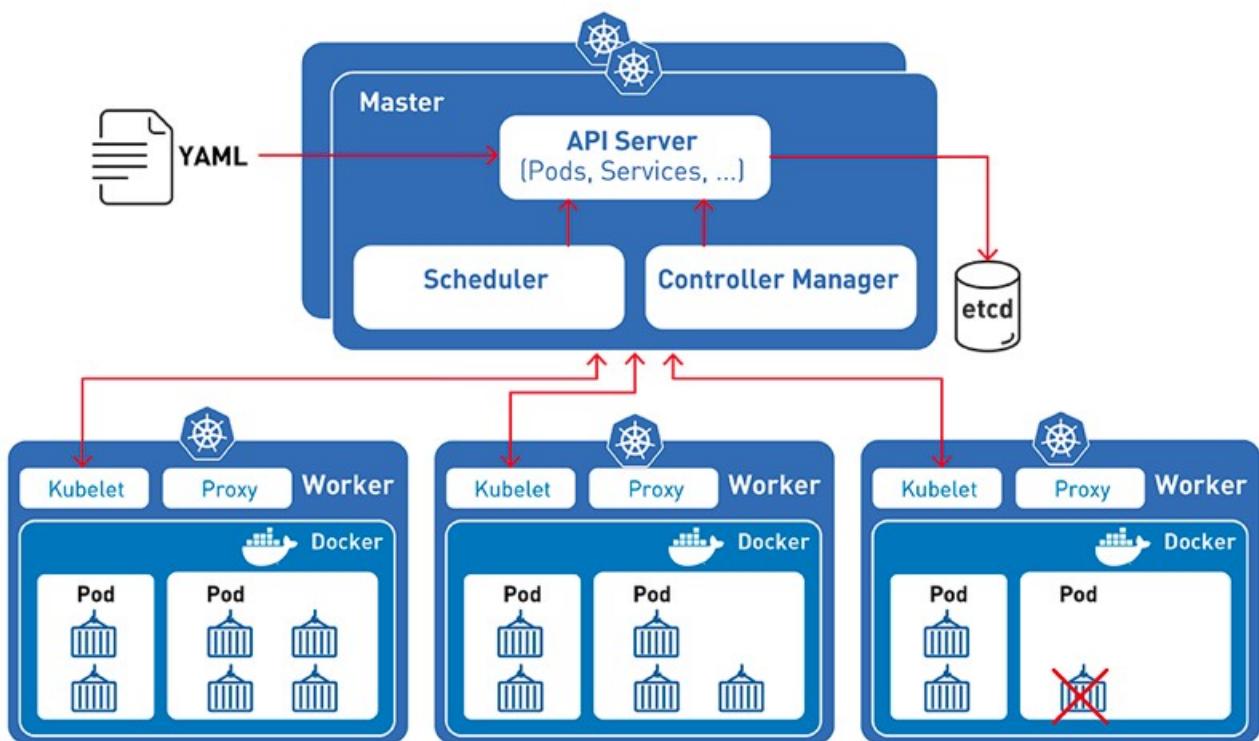
Recargamos los servicios del sistema y habilitamos el runtime para que se inice automáticamente ante próximos reinicios del servidor:

```
$ sudo systemctl daemon-reload  
$ sudo systemctl start crio  
$ sudo systemctl enable crio --now
```



Capítulo 11. Lab: Instalación Kubernetes (1 master + 2 minions) - Centos7

Mediante este laboratorio, pondremos en marcha un clúster de kubernetes compuesto por una máquina maestra y dos máquinas de computación minions.



Para la correcta instalación del cluster de kubernetes en modo manual, los nodos deben poseer una interfaz de red con internet y deben poder ver al resto de nodos.

11.1. Instalación manual con kubeadm

El proceso de instalación lo realizaremos con la propia herramienta **kubeadm** que nos proporcionan los binarios de kubernetes.

Llevaremos a cabo el proceso de instalación en los tres nodos:

- Nodo máster
- Nodo minion1
- Nodo minion2

El correcto aprovisionamiento de kubernetes en cada nodo del clúster requiere del siguiente software en cada uno de los nodos:

- Runtime de contenedores (Gestor de contenedores)
 - Docker

- CRI-O
- Etc.
- Kubelet (Agente de kubernetes)
- Kubeadm (Comando de construcción del cluster local)
- kubectl (Cliente de Kubernetes)
- CNI (Soporte para redes CNI)

11.2. Configuración de master y nodos



Este procedimiento se realizará tanto en el master como en los minions

```
[kubernetes@kubemaster ~]$ sudo su -
Last login: Tue May 22 21:20:44 CEST 2018 on pts/0
```

- Instalamos el repositorio en los tres nodos:
- Ejecutamos el conjunto de instrucciones con privilegios de root

```
#: cat <<EOF > /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-el7-x86_64
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg
exclude=kube*
EOF
```

- Configuramos el bridge de iptables y el forwarding de ipv4

```
#: cat <<EOF > /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
net.ipv4.ip_forward=1
EOF
```

- Activamos la configuración

```
#: sysctl --system
```

- Es necesario deshabilitar la swap para kubernetes.
- Para ello paramos la swap

```
#: swapoff -a
```

- Para eliminarla de forma permanente, se deben borrar las entradas de swap del fichero /etc/fstab
- Luego instalaremos el software necesario para master y nodes

```
[root@kubemaster ~]# yum -y install kubeadm kubectl kubelet kubernetes-cni --disableexcludes=kubernetes
```

Consideramos que está ya instalado el motor de contenedores, sino habría que instalarlo

Con el modificador --disableexclude, estamos indicando que cuando el sistema operativo, en nuestro caso, el gestor de paquetes yum, actualice el sistema, tenga en cuenta que también hay que actualizar el repositorio que atiende al nombre de kubernetes

Para bloquear la actualización de paquetes del repositorio de kubernetes ante una actualización de mantenimiento de paquetes el sistema, cambiariamos el modificador por --disableincludes=kubernetes

- Automatizamos el arranque del servicio kubelet en los tres nodos

```
# systemctl enable kubelet
```

11.3. Inicio del cluster

Una de las máquinas será la que iniciará el clúster, para ello, utilizaremos la herramienta **kubeadm** y ejecutaremos este comando una única vez en la máquina maestra que hayamos designado.

Si quisiéramos indicar al crear el clúster, el rango de direcciones IP para la red de los Pods, podemos indicarlo de la siguiente forma, de forma que el control-plane automáticamente alojará los segmentos CIDRs para cada nodo.

Por ejemplo, podríamos indicar el siguiente segmento de red

```
# kubeadm init --pod-network-cidr=10.244.0.0/16
```

```
[root@kubemaster ~]# kubeadm init --apiserver-advertise-address 192.168.15.100 --control-plane-endpoint 192.168.15.100
[init] Using Kubernetes version: v1.21.1
[preflight] Running pre-flight checks
  [WARNING SystemVerification]: this Docker version is not on the list of validated versions: 18.09.3. Latest validated
version: 18.06
  [preflight] Pulling images required for setting up a Kubernetes cluster
  [preflight] This might take a minute or two, depending on the speed of your internet connection
  [preflight] You can also perform this action in beforehand using 'kubeadm config images pull'
  [kubelet-start] Writing kubelet environment file with flags to file "/var/lib/kubelet/kubeadm-flags.env"
  [kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"
...
...
...
Your Kubernetes master 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/>

You can now join any number of machines by running the following on each node as root:

```
kubeadm join 192.168.15.100:6443 --token s61afs.6xwwwi0nthlu6t96 --discovery-token-ca-cert-hash
sha256:8d99d3a480ddf80b0305adb4312d86999bfb51d571b4e0d573b79438df48363
```

11.4. Configurando el acceso al clúster

Para poder comunicarnos con el clúster, necesitamos ejecutar ciertas líneas que la traza del comando de inicio del clúster nos muestra por pantalla.

De esta forma, le decimos al binario cliente kubectl donde está el clúster y como debe de comunicarse con el.

```
[kubernetes@kubemaster ~]$ mkdir -p $HOME/.kube
[kubernetes@kubemaster ~]$ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
[kubernetes@kubemaster ~]$ sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

11.5. Añadiendo nodos

Seguidamente, vamos a añadir los nodos minions de computación al clúster.



Debemos de acordarnos de usar la URL que se nos otorga al iniciar el master, no la que viene a continuación :D

Ejecutamos la instrucción en el nodo minion1:

```
[root@minion1 ~]# kubeadm join 192.168.15.100:6443 --token s61afs.6xwwwi0nthlu6t96 --discovery-token-ca-cert-hash sha256:8d99d3a480ddf80b0305adb4312d86999bfbb51d571b4e0d573b79438df48363
```

Posteriormente, ejecutamos también la instrucción en el minion 2:

```
[root@minion2 ~]# kubeadm join 192.168.15.100:6443 --token s61afs.6xwwwi0nthlu6t96 --discovery-token-ca-cert-hash sha256:8d99d3a480ddf80b0305adb4312d86999bfbb51d571b4e0d573b79438df48363
```

11.6. Comprobando el inventario de nodos

Finalmente, procedemos a consultar el inventario de nodos, para comprobar que estos han sido añadidos correctamente al clúster.

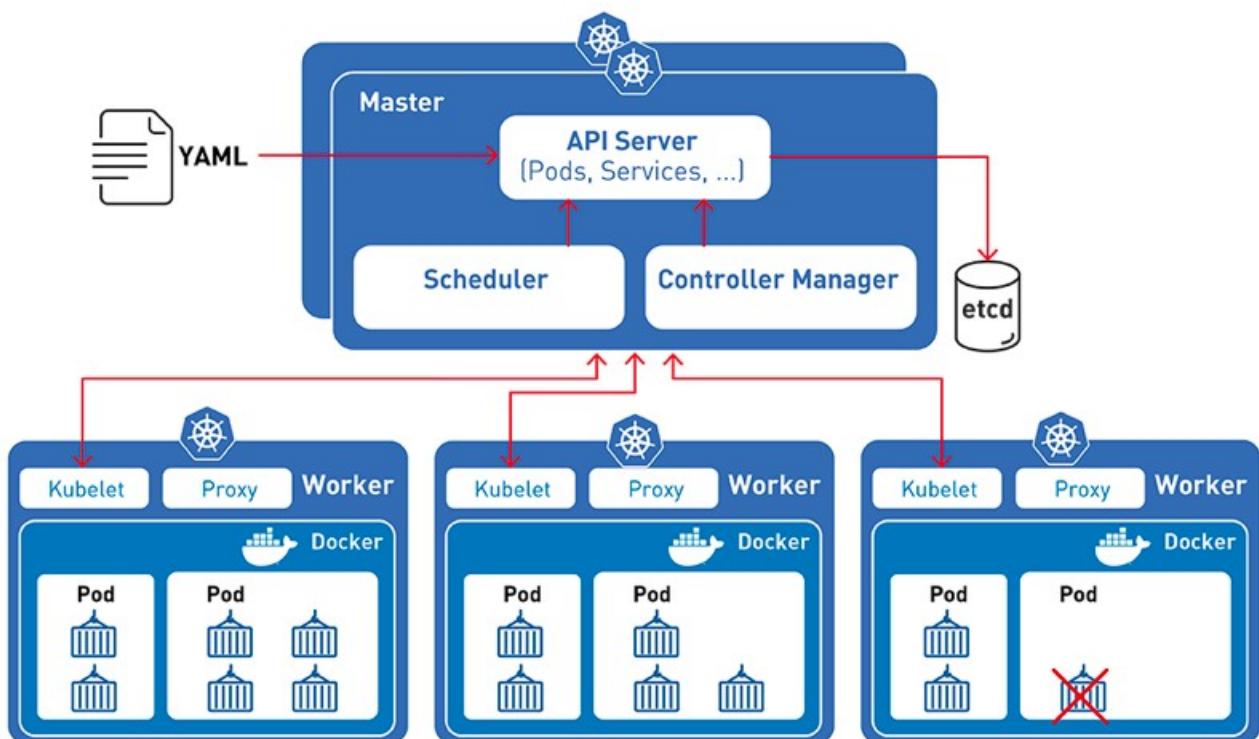
```
[kubernetes@kubemaster ~]$ kubectl get nodes
NAME      STATUS   ROLES   AGE    VERSION
kubemaster.local   Ready    master  14m   v1.21.1
minion1.local     Ready    <none> 103s  v1.21.1
minion2.local     Ready    <none> 96s   v1.21.1
```

Puede suceder que una vez añadidos los nodos y sin haber instalado aún un driver de red, que los nodos aparezcan en el inventario en estado **NotReady**, no debemos de preocuparnos por eso.

Una vez el driver de red esté correctamente instalado, todos los nodos deberían de aparecer en el estado que se muestra, **Ready**

Capítulo 12. Lab: Instalación Kubernetes (1 master + 2 minions) - Fedora

Mediante este laboratorio, pondremos en marcha un clúster de kubernetes compuesto por una máquina maestra y dos máquinas de computación minions.



Para la correcta instalación del cluster de kubernetes en modo manual, los nodos deben poseer una interfaz de red con internet y deben poder ver al resto de nodos.

12.1. Instalación manual con kubeadm

El proceso de instalación lo realizaremos con la propia herramienta **kubeadm** que nos proporcionan los binarios de kubernetes.

Llevaremos a cabo el proceso de instalación en los tres nodos:

- Nodo máster
- Nodo minion1
- Nodo minion2

El correcto aprovisionamiento de kubernetes en cada nodo del clúster requiere del siguiente software en cada uno de los nodos:

- Runtime de contenedores (Gestor de contenedores)
 - Docker

- CRI-O
- Etc.
- Kubelet (Agente de kubernetes)
- Kubeadm (Comando de construcción del cluster local)
- kubectl (Cliente de Kubernetes)
- CNI (Soporte para redes CNI)

12.2. Configuración de master y nodos



Este procedimiento se realizará tanto en el master como en los minions

```
$ sudo su -
Last login: Tue May 22 21:20:44 CEST 2018 on pts/0
```

12.3. Configurando el IP Forwarding

Ahora, vamos a proceder a configurar el bridge de iptables y el forwarding de ipv4.

Ejecutamos la siguiente instrucción:

```
#: cat <<EOF > /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
net.ipv4.ip_forward=1
EOF
```

Recargamos seguidamente la configuración para no tener que reiniciar el servidor:

```
#: sysctl --system
```

12.4. Deshabilitando la memoria SWAP

Procedemos a continuación con la desactivación de la memoria swap del servidor.

Ejecutamos el siguiente comando y a continuación reiniciamos el servidor:

```
#: dnf remove zram-generator-defaults
```

Una vez reiniciado el servidor, ejecutamos el siguiente comando para asegurarnos de que la swap no ha sido cargada al inicio del sistema:

```
# zramctl
```

Si no indica nada, es que la swap está correctamente desactivada.

12.5. Instalando los paquetes de Kubernetes

```
# dnf install kubernetes-kubeadm cri-tools iproute-tc -y
```



Consideramos que está ya instalado el motor de contenedores, sino habría que instalarlo

Seguidamente, vamos a indicar al sistema operativo que si se reinicia, debe de levantar de forma automática el servicio de kubelet, esta operación hay que realizarla en los 3 nodos:

```
# systemctl enable kubelet
```

12.6. Desactivando systemd-resolved

systemd-resolved es un servicio de systemd que proporciona resolución de nombres de red a aplicaciones locales a través de una interfaz D-Bus , el servicio NSS de resolve (nss-resolve(8)), y un receptor de escucha de DNS local en 127.0.0.53.

Para que a los Pods CoreDNS no tengan problemas a la hora de seleccionar el sistema de resolución DNS, y que utilicen el interno del propio kubernetes, debemos de desactivar dicha funcionalidad en el sistema operativo.

Podemos encontrar más información en la siguiente URL: <https://coredns.io/plugins/loop/#troubleshooting>

Ejecutamos la siguiente instrucción:

```
# systemctl disable --now systemd-resolved
```

Adicionalmente, editamos el archivo **/etc/NetworkManager/NetworkManager.conf** y ajustamos en la sección **[main]** lo siguiente:

```
# nano /etc/NetworkManager/NetworkManager.conf
```

```
[main]
dns=default
```

Ahora, eliminamos el archivo **/etc/resolv.conf** y lo creamos de nuevo limpio:

```
# unlink /etc/resolv.conf  
# touch /etc/resolv.conf
```

12.7. Estableciendo registry para imágenes no FQDN

En los sistemas operativos modernos, como es el caso de la distribución fedora que utilizamos para el curso, cuando tenemos imágenes que se consideran "no correctamente cualificadas", esto quiere decir, imágenes docker que no llevan en el nombre el prefijo del registry del cual deben de obtenerse, sucede que el sistema operativo puede tener un orden de preferencia por defecto para dirigirse a un registry concreto y puede que ese registry no sea el que nosotros queremos, provocando en la mayoría de los casos, errores del tipo "acceso no autorizado", o bien, del tipo "imagen no encontrada".

Editamos el archivo que se encuentra en la ruta: **/etc/containers/registries.conf** y comentamos la siguiente línea añadiendo un carácter almohadilla delante (#):

```
unqualified-search-registries = ["registry.fedoraproject.org", "registry.access.redhat.com", "docker.io", "quay.io"]
```

Seguidamente, añadimos la siguiente línea justo debajo:

```
unqualified-search-registries = ["docker.io"]
```

Con esto, estamos indicándole al sistema operativo, y más concretamente, al motor de contenedores, que cuando se intente descargar imágenes que no se consideren full FQDN (que no tengan la dns del registry como prefijo en el nombre de la imagen docker), que se vaya por defecto al registry docker.io para proceder a su descarga.

Llegado este punto reiniciamos el servidor ejecutando en consola el siguiente comando:

```
# shutdown -r now
```

12.8. Inicio del cluster

Una de las máquinas será la que iniciará el clúster, para ello, utilizaremos la herramienta **kubeadm** y ejecutaremos este comando una única vez en la máquina maestra que hayamos designado.

```
# kubeadm init --apiserver-advertise-address 192.168.15.100 --control-plane-endpoint 192.168.15.100 --pod-network-cidr=10.244.0.0/16

[init] Using Kubernetes version: v1.21.1
[preflight] Running pre-flight checks
  [WARNING SystemVerification]: this Docker version is not on the list of validated versions: 18.09.3. Latest validated version: 18.06
  [preflight] Pulling images required for setting up a Kubernetes cluster
  [preflight] This might take a minute or two, depending on the speed of your internet connection
  [preflight] You can also perform this action in beforehand using 'kubeadm config images pull'
  [kubelet-start] Writing kubelet environment file with flags to file "/var/lib/kubelet/kubeadm-flags.env"
  [kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"

...
...
...

Your Kubernetes master 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/>

You can now join any number of machines by running the following on each node as root:

```
kubeadm join 192.168.15.100:6443 --token s61afs.6xwwwi0nthlu6t96 --discovery-token-ca-cert-hash sha256:8d99d3a480ddf80b0305adb4312d86999bfbb51d571b4e0d573b79438df48363
```

12.9. Configurando el acceso al clúster en la máquina master

Para poder comunicarnos con el clúster, necesitamos ejecutar ciertas líneas que la traza del comando de inicio del clúster nos muestra por pantalla.

De esta forma, le decimos al binario cliente kubectl donde está el clúster y como debe de comunicarse con el.



Es importante recordar que debemos ejecutar las líneas en una consola de la máquina maestra siendo un usuario normal, no el root.

```
$ mkdir -p $HOME/.kube
$ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
$ sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

12.10. Añadiendo nodos de computación

Seguidamente, vamos a añadir los nodos minions de computación al clúster :D



Debemos de acordarnos de usar la URL que se nos otorga al iniciar el master, no la que viene a continuación :D

Ejecutamos la instrucción en el nodo minion1:

```
[root@minion1 ~]# kubeadm join 192.168.15.100:6443 --token s61afs.6xwwwi0nthlu6t96 --discovery-token-ca-cert-hash sha256:8d99d3a480ddf80b0305adb4312d86999bfbb51d571b4e0d573b79438df48363
```

Posteriormente, ejecutamos también la instrucción en el minion 2:

```
[root@minion2 ~]# kubeadm join 192.168.15.100:6443 --token s61afs.6xwwwi0nthlu6t96 --discovery-token-ca-cert-hash sha256:8d99d3a480ddf80b0305adb4312d86999bfbb51d571b4e0d573b79438df48363
```



12.11. Comprobando el inventario de nodos

Finalmente, procedemos a consultar el inventario de nodos, para comprobar que estos han sido añadidos correctamente al clúster.

```
[kubernetes@kubemaster ~]$ kubectl get nodes
NAME      STATUS   ROLES   AGE    VERSION
kubemaster.local   Ready    master  14m   v1.21.1
minion1.local     Ready    <none>  103s  v1.21.1
minion2.local     Ready    <none>  96s   v1.21.1
```



Puede suceder que una vez añadidos los nodos y sin haber instalado aún un driver de red, que los nodos aparezcan en el inventario en estado **NotReady**, no debemos de preocuparnos por eso.

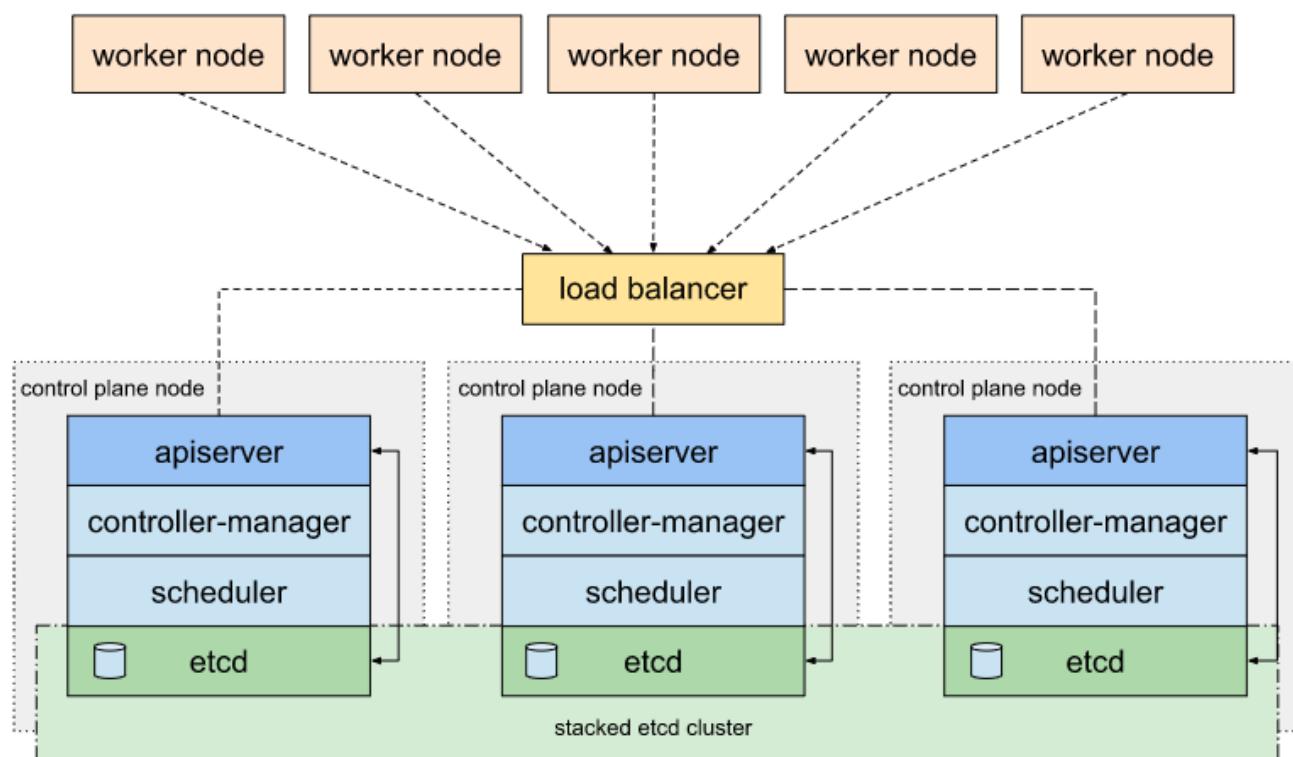
Una vez el driver de red esté correctamente instalado, todos los nodos deberían de aparecer en el estado que se muestra, **Ready**

Capítulo 13. Lab: Instalación Kubernetes HA (Stacked) - Centos 7

Mediante este laboratorio, instalaremos un clúster de kubernetes en modo de alta disponibilidad, de forma que tendremos varias máquinas máster dando cobertura al clúster.

Adicionalmente, dispondremos de una máquina que actuará de balanceador de carga, así como de 2 máquinas de computación minions.

Mediante este laboratorio, el servicio de almacenamiento etcd estará replicado entre los servidores maestros, ya que se encontrará dentro de los mismos.



13.1. Clonando las máquinas maestras con VirtualBox

Lo primero que vamos a llevar a cabo es la clonación de nuestra máquina maestra **kubernetes-master.local-clone** con VirtualBox.

El primer clon lo vamos a llamar **kubernetes-master2.local-clone**

El segundo clon lo vamos a llamar **kubernetes-master3.local-clone**

De forma que tendremos 3 máquinas maestras cargadas en VirtualBox con los nombres:

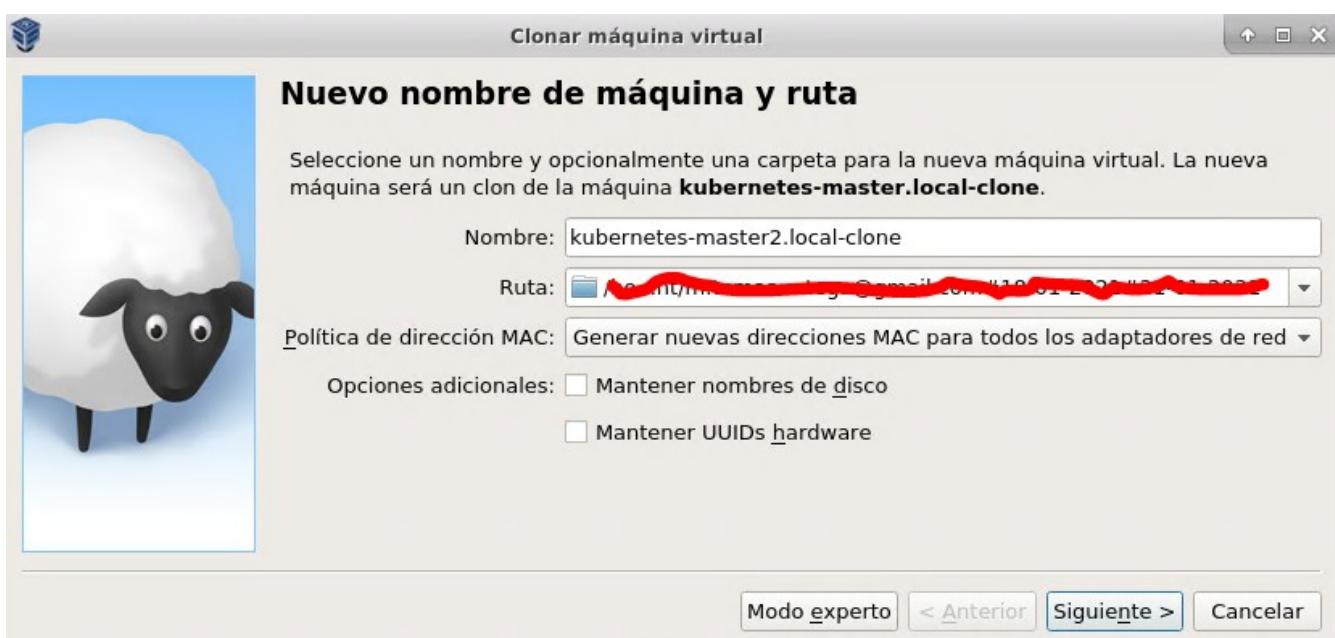
- **kubernetes-master.local-clone**
- **kubernetes-master2.local-clone**
- **kubernetes-master3.local-clone**



La máquina sobre la que posteriormente iniciaremos el clúster será la **kubernetes-master.local-clone**

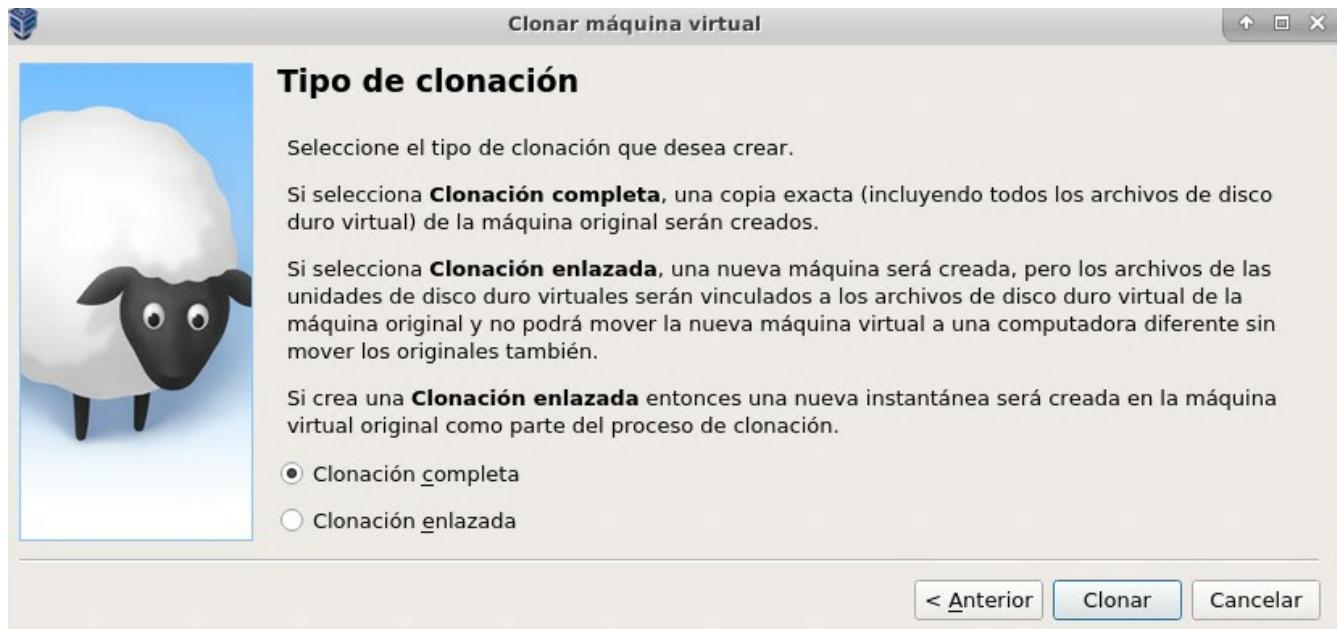
Seleccionamos la máquina maestra desde la propia interfaz de VirtualBox y seguidamente pulsamos sobre el menú de **Máquina > Clonar**, indicando las siguientes opciones:

- Nombre
 - kubernetes-master2.local-clone
- Ruta
 - Indicamos la ruta donde tengamos nuestras máquinas virtuales
- Política de dirección MAC
 - Generar nuevas direcciones MAC para todos los adaptadores de red



Pulsamos sobre el botón **Siguiente**

Indicamos que queremos **Clonación completa** y pulsamos sobre el botón **Clonar**



Repetimos los pasos de un nuevo clon a partir de la máquina **kubernetes-master.local-clone** para obtener la máquina **kubernetes-master3.local-clone**

13.2. Clonando el balanceador de carga (haproxy)

Para instalar y poner en marcha nuestro balanceador de carga haproxy vamos a alojarlo en una máquina separada que haga dicha función.

Clonamos una máquina realizando el mismo procedimiento que para las máquinas maestras, en este caso, vamos a partir de la máquina **kubernetes-minion1.local-clone** y le pondremos de nombre al clon **kubernetes-loadbalancer.local-clone**

13.3. Asignación de recursos a las máquinas virtuales

Para que el clúster vaya bien y kubernetes no se queje de que las máquinas tienen pocos recursos para que operen con normalidad, vamos a asignar las siguientes cuotas de CPU y RAM a las máquinas:

- **kubernetes-master.local-clone**
 - CPU: 2
 - RAM: 3072 MB
- **kubernetes-master2.local-clone**
 - CPU: 2
 - RAM: 3072 MB
- **kubernetes-master3.local-clone**
 - CPU: 2
 - RAM: 3072 MB
- **kubernetes-minion1.local-clone**

- CPU: 2
- RAM: 4096 MB

- **kubernetes-minion2.local-clone**

- CPU: 2
- RAM: 2048 MB

- **kubernetes-loadbalancer.local-clone**

- CPU: 2
- RAM: 2048 MB

13.4. Ajustando el direccionamiento IP a las máquinas virtuales

El siguiente paso va a ser llevar a cabo el pertinente ajuste en el direccionamiento IP de nuestros servidores.

Asignaremos IP fija y nombre de host en todos los servidores, de forma que todas las máquinas puedan actuar como resolvidores DNS y no haya ningún problema en el tráfico de red entre las mismas.

Editamos en todas las máquinas el archivo **/etc/hosts** y le asignamos el siguiente contenido:

```
127.0.0.1 localhost localhost.localdomain localhost4 localhost4.localdomain4
::1      localhost localhost.localdomain localhost6 localhost6.localdomain6

192.168.15.100 kubernetes-master.local-clone
192.168.15.101 kubernetes-master2.local-clone
192.168.15.102 kubernetes-master3.local-clone
192.168.15.103 kubernetes-loadbalancer.local-clone
192.168.15.104 kubernetes-minion1.local-clone
192.168.15.105 kubernetes-minion2.local-clone
```

Arrancamos la primera máquina **kubernetes-master.local-clone**:

Cambiamos el contenido del archivo **/etc/hosts** por el que hemos comentado.

```
$ sudo nano /etc/hosts
```

Seguidamente, cambiamos el nombre de host de la máquina **kubernetes-master.local-clone**, editamos el archivo **/etc/hostname** e indicamos el nombre que tendrá la propia máquina **kubernetes-master.local-clone**

```
$ sudo nano /etc/hostname
```

A continuación, vamos a ajustar la dirección IP estática correcta para la máquina virtual, ejecutamos el siguiente comando para editar el archivo `/etc/sysconfig/network-scripts/ifcfg-eth0`:

```
$ sudo nano /etc/sysconfig/network-scripts/ifcfg-eth0
```

Indicamos la siguiente información para la primera máquina maestra:

```
DEVICE=eth0
NM_CONTROLLED="no"
TYPE=Ethernet
IPADDR=192.168.15.100
NETMASK=255.255.255.0
GATEWAY=192.168.15.1
DNS1=8.8.8.8
BOOTPROTO=none
ONBOOT=yes
DELAY=0
```



Cambiamos la IP que aparece en el elemento IPADDR, el resto de opciones no se tocan.

Tendremos que ajustar en cada máquina su ip fija correspondiente que hemos previamente establecido como criterio de asignación a cada una.

El último paso sería reiniciar primero el servicio de Red y a continuación reiniciar la propia máquina para verificar que toda la configuración que hemos indicado es correcta, ejecutamos:

```
$ sudo systemctl restart network && sudo shutdown -r now
```

Si la máquina ha arrancado, la apagamos, encendemos la siguiente y repetimos el procedimiento, ajustando los datos específicos para la misma.

13.5. Instalando el balanceador de carga (haproxy)

Vamos a instalar y configurar el balanceador de carga haproxy.

Nos conectamos a la máquina `kubernetes-loadbalancer.local-clone` y ejecutamos el siguiente comando:

```
$ sudo yum install haproxy -y
```

Seguidamente, vamos a editar el archivo de configuración del balanceador `/etc/haproxy/haproxy.cfg`

```
$ sudo nano /etc/haproxy/haproxy.cfg
```

Y sustituímos todo el contenido por este:

```
global
    user haproxy
    group haproxy

defaults
    mode http
    log global
    retries 2
    timeout connect 3000ms
    timeout server 5000ms
    timeout client 5000ms

frontend kubernetes
    bind *:6443
    option tcplog
    mode tcp
    default_backend kubernetes-master-nodes

backend kubernetes-master-nodes
    mode tcp
    balance roundrobin
    option tcp-check
    server kubernetes-master1.local 192.168.15.100:6443 check fall 3 rise 2
    server kubernetes-master2.local 192.168.15.101:6443 check fall 3 rise 2
    server kubernetes-master3.local 192.168.15.102:6443 check fall 3 rise 2
```

Recargamos los daemons del sistema y reiniciamos el propio servicio haproxy:

```
$ sudo systemctl daemon-reload && sudo systemctl restart haproxy
```

13.6. Instalando paquetes de Kubernetes



Este procedimiento se realizará tanto en las máquinas maestras como en los minions.

En primer lugar, pasamos a modo root en la consola:

```
$ sudo su -
```

- Instalamos el repositorio tanto en las máquinas maestras como en las máquinas de computación minions

- Ejecutamos el conjunto de instrucciones con privilegios de root

```
#: cat <<EOF > /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-el7-x86_64
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg
exclude=kube*
EOF
```

- Configuramos el bridge de iptables y el forwarding de ipv4

```
#: cat <<EOF > /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
net.ipv4.ip_forward=1
EOF
```

- Activamos la configuración

```
#: sysctl --system
```

- Es necesario deshabilitar la swap para kubernetes.
- Para ello paramos la swap

```
#: swapoff -a
```

- 
- Para eliminarla de forma permanente, se deben borrar las entradas de swap del fichero /etc/fstab

- Luego instalaremos el software necesario para master y minions

```
# yum -y install kubeadm kubectl kubelet kubernetes-cni --disableexcludes=kubernetes
```



Consideramos que está ya instalado Docker, sino habría que instalarlo

Con el modificador --disableexclude, estamos indicando que cuando el sistema operativo, en nuestro caso, el gestor de paquetes yum, actualice el sistema, tenga en cuenta que también hay que actualizar el repositorio que atiende al nombre de kubernetes



Para bloquear la actualización de paquetes del repositorio de kubernetes ante una actualización de mantenimiento de paquetes el sistema, cambiariamos el modificador por --disableincludes=kubernetes

- Automatizamos el arranque del servicio kubelet en los nodos master y los nodos minions

```
# systemctl enable kubelet
```

13.7. Creando el archivo de configuración para la red CNI de kubernetes

Tanto en las máquinas maestras como en las minions, vamos a crear la siguiente estructura de directorio:

```
# mkdir -p /etc/cni/net.d/
```

Tanto en las máquinas maestras como en las minions, vamos a crear el archivo **/etc/cni/net.d/10-weave.conflist**

```
$ sudo nano /etc/cni/net.d/10-weave.conflist
```

Le añadimos el siguiente contenido:

```
{
  "cniVersion": "0.3.0",
  "name": "weave",
  "plugins": [
    {
      "name": "weave",
      "type": "weave-net",
      "hairpinMode": true
    },
    {
      "type": "portmap",
      "capabilities": {"portMappings": true},
      "snat": true
    }
  ]
}
```

Actualmente se produce un error en la puesta en marcha de un clúster stacked de kubernetes en modo HA con varios máster.



Al incorporar al segundo máster, parece que hay un problema de acceso de permisos al directorio de configuración de la red CNI que acabamos de crear.

Por el momento, la comunidad lo está solvendando indicando a dicha carpeta los permisos 777 hasta que se corrija el problema.

Ejecutamos en las máquinas maestras y los minions:

```
$ sudo chmod 777 /etc/cni/net.d
```

13.8. Realizando limpieza de Iptables

Partimos de una instalación limpia, pero no viene mal ejecutar el siguiente comando en todos los servidores master y minions que van a formar parte del clúster, para dejar las Iptables por defecto y ya posteriormente que kubernetes o nosotros, vayamos ajustando lo que se vaya necesitando.

```
$ sudo iptables -F
```

13.9. Inicio del cluster

El inicio del clúster vamos a llevarlo a cabo desde la máquina maestra **kubernetes-master.local-clone** que fue la primera máquina a partir de la cual luego hicimos los clones de las otras.

La que tiene asignada la IP 192.168.15.100.

Ejecutamos el siguiente comando:

```
# kubeadm init --pod-network-cidr=10.95.10.0/16 --apiserver-advertise-address "192.168.15.100" --control-plane-endpoint "192.168.15.103:6443" --upload-certs
```

- **--pod-network-cidr**
 - Indicamos el segmento de red para los pods de usuario que el clúster creará
- **--apiserver-advertise-address**
 - Indicamos como referencia de inicio del clúster la ip de la máquina donde lanzamos el comando
- **--control-plane-endpoint**
 - Indicamos la IP y el puerto por el que vamos a conectar contra el balanceador de carga
- **--upload-certs**
 - Indicamos que queremos crear un objeto de tipo secret en el clúster, donde estarán los

certificados que cifran las comunicaciones del clúster de kubernetes, de manera que cuando se vayan incorporando más máquinas maestras o minions, estas los puedan utilizar para establecer comunicaciones de forma distribuida y cifrada

Si todo ha ido bien, debería de aparecer una traza en la consola parecida a esta:

```
...
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
```

Alternatively, if you are the root user, you can run:

```
export KUBECONFIG=/etc/kubernetes/admin.conf
```

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/>

You can now join any number of the control-plane node running the following command on each as root:

```
kubeadm join 192.168.15.103:6443 --token g5k7nq.ay8g5jpwhy48j9ez \
--discovery-token-ca-cert-hash sha256:f77ef9030fc6394e281b403c4e2a833ff0027830849ff58dc91517c366e0275a \
--control-plane --certificate-key cfbf72d887ae45325eade51f4a0994c378f02b4ac16356563bc70f02291e2fe1
```

Please note that the certificate-key gives access to cluster sensitive data, keep it secret!

As a safeguard, uploaded-certs will be deleted in two hours; If necessary, you can use "kubeadm init phase upload-certs --upload-certs" to reload certs afterward.

Then you can join any number of worker nodes by running the following on each as root:

```
kubeadm join 192.168.15.103:6443 --token g5k7nq.ay8g5jpwhy48j9ez \
--discovery-token-ca-cert-hash sha256:f77ef9030fc6394e281b403c4e2a833ff0027830849ff58dc91517c366e0275a
```

El sistema nos indica dos comandos a ejecutar, el comando kubeadm join para las máquinas maestras que queramos incorporar así como el comando específico kubeadm join para las máquinas mininos que queramos incorporar.

Para comenzar a interactuar con el clúster desde la máquina maestra que ha iniciado el clúster vamos a ejecutar primero los comandos que nos indica la traza de la consola: **you need to run the following as a regular user**

Seguidamente, ejecutamos en la máquina maestra, mediante el cliente kubectl, la orden de instalación del driver weavenet para kubernetes, ejecutamos el siguiente comando:

```
$ kubectl apply -f "https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version | base64 | tr -d '\n')"

serviceaccount/weave-net created
clusterrole.rbac.authorization.k8s.io/weave-net created
clusterrolebinding.rbac.authorization.k8s.io/weave-net created
role.rbac.authorization.k8s.io/weave-net created
rolebinding.rbac.authorization.k8s.io/weave-net created
daemonset.apps/weave-net created
```

Monitorizamos el proceso de instalación y puesta en marcha de los componentes del driver, ejecutamos el comando:

```
$ watch kubectl get pods -n kube-system
```

Cuando observemos que todos los pods están operativos, entonces el procedimiento de instalación del driver habrá finalizado:

NAME	READY	STATUS	RESTARTS	AGE
coredns-74ff55c5b-m9dpz	1/1	Running	0	8m48s
coredns-74ff55c5b-q9p42	1/1	Running	0	8m48s
etcd-kubernetes-master.local-clone	1/1	Running	0	8m55s
kube-apiserver-kubernetes-master.local-clone	1/1	Running	0	8m55s
kube-controller-manager-kubernetes-master.local-clone	1/1	Running	0	8m55s
kube-proxy-qcn62	1/1	Running	0	8m48s
kube-scheduler-kubernetes-master.local-clone	1/1	Running	0	8m55s
weave-net-b9vsc	2/2	Running	0	78s

Posteriormente, también comprobamos que los nodos del clúster (en este momento sólo hay uno) están completamente operativos, ejecutamos el comando:

```
$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
kubernetes-master.local-clone	Ready	control-plane,master	9m57s	v1.20.1

13.10. Añadiendo a las máquinas master

En las máquinas máster, ejecutamos el siguiente comando:

```
kubeadm join 192.168.15.103:6443 --token g5k7nq.ay8g5jpwhy48j9ez \
--discovery-token-ca-cert-hash sha256:f77ef9030fc6394e281b403c4e2a833ff0027830849ff58dc91517c366e0275a \
--control-plane --certificate-key cfbf72d887ae45325eade51f4a0994c378f02b4ac16356563bc70f02291e2fe1
```

This node has joined the cluster and a new control plane instance was created:

- * Certificate signing request was sent to apiserver and approval was received.
- * The Kubelet was informed of the new secure connection details.
- * Control plane (master) label and taint were applied to the new node.
- * The Kubernetes control plane instances scaled up.
- * A new etcd member was added to the local/stacked etcd cluster.

To start administering your cluster from this node, 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
```

Run 'kubectl get nodes' to see this node join the cluster.



Cada alumno tendrá que lanzar su propio comando, ya que los token sha256 se generan de forma diferente por cada kubeadm init que hagamos.

13.11. Añadiendo las máquinas minions

En las máquinas máster, ejecutamos el siguiente comando:

```
kubeadm join 192.168.15.103:6443 --token g5k7nq.ay8g5jpwhy48j9ez \
--discovery-token-ca-cert-hash sha256:f77ef9030fc6394e281b403c4e2a833ff0027830849ff58dc91517c366e0275a
```

This node has joined the cluster:

- * Certificate signing request was sent to apiserver and a response was received.
- * The Kubelet was informed of the new secure connection details.

Run 'kubectl get nodes' on the control-plane to see this node join the cluster.



Cada alumno tendrá que lanzar su propio comando, ya que los token sha256 se generan de forma diferente por cada kubeadm init que hagamos.

13.12. Comprobando el estado del clúster

Finalmente, vamos a ejecutar el siguiente comando para verificar que todos los Pods necesarios para la maquinaria de kubernetes están operativos:

```
$ kubectl get pod -n kube-system
```

NAME	READY	STATUS	RESTARTS	AGE
coredns-74ff55c5b-m9dpz	1/1	Running	0	23m
coredns-74ff55c5b-q9p42	1/1	Running	0	23m
etcd-kubernetes-master.local-clone	1/1	Running	0	23m
etcd-kubernetes-master2.local-clone	1/1	Running	0	7m57s
etcd-kubernetes-master3.local-clone	1/1	Running	0	4m34s
kube-apiserver-kubernetes-master.local-clone	1/1	Running	0	23m
kube-apiserver-kubernetes-master2.local-clone	1/1	Running	0	7m58s
kube-apiserver-kubernetes-master3.local-clone	1/1	Running	0	4m35s
kube-controller-manager-kubernetes-master.local-clone	1/1	Running	1	23m
kube-controller-manager-kubernetes-master2.local-clone	1/1	Running	0	7m58s
kube-controller-manager-kubernetes-master3.local-clone	1/1	Running	0	4m35s
kube-proxy-5zfvl	1/1	Running	0	68s
kube-proxy-bg9sg	1/1	Running	0	2m29s
kube-proxy-c8tgc	1/1	Running	0	7m59s
kube-proxy-jsqvm	1/1	Running	0	4m35s
kube-proxy-qcn62	1/1	Running	0	23m
kube-scheduler-kubernetes-master.local-clone	1/1	Running	1	23m
kube-scheduler-kubernetes-master2.local-clone	1/1	Running	0	7m58s
kube-scheduler-kubernetes-master3.local-clone	1/1	Running	0	4m35s
weave-net-2g5vv	2/2	Running	0	2m29s
weave-net-48lpj	2/2	Running	0	7m59s
weave-net-b9vsc	2/2	Running	0	15m
weave-net-j9lsz	2/2	Running	0	4m35s
weave-net-zbpm8	2/2	Running	1	68s

También ejecutamos el siguiente comando, para observar el estado de los nodos:

```
$ kubectl get nodes
```

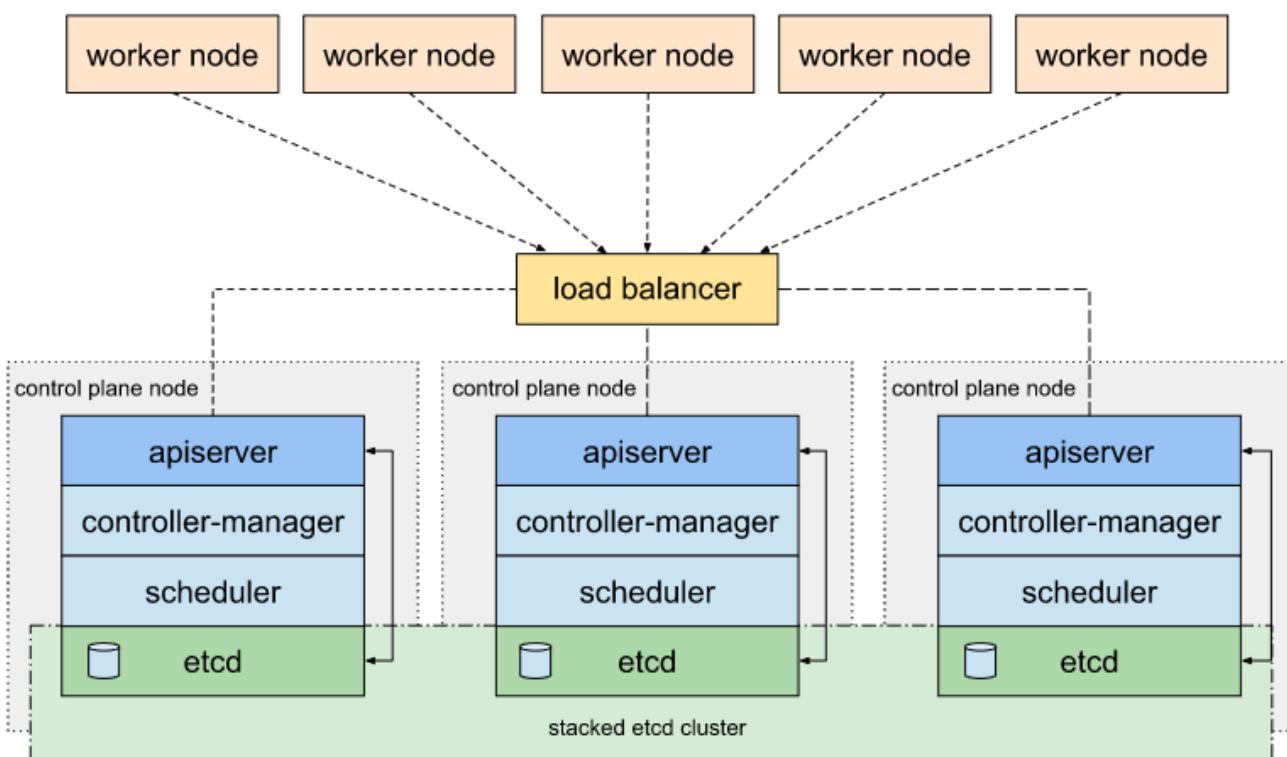
NAME	STATUS	ROLES	AGE	VERSION
kubernetes-master.local-clone	Ready	control-plane,master	25m	v1.20.1
kubernetes-master2.local-clone	Ready	control-plane,master	10m	v1.20.1
kubernetes-master3.local-clone	Ready	control-plane,master	6m49s	v1.20.1
kubernetes-minion1.local-clone	Ready	<none>	4m42s	v1.20.1
kubernetes-minion2.local-clone	Ready	<none>	3m21s	v1.20.1

Capítulo 14. Lab: Instalación Kubernetes HA (Stacked) - Fedora

Mediante este laboratorio, instalaremos un clúster de kubernetes en modo de alta disponibilidad, de forma que tendremos varias máquinas máster dando cobertura al clúster.

Adicionalmente, dispondremos de una máquina que actuará de balanceador de carga, así como de 2 máquinas de computación minions.

Mediante este laboratorio, el servicio de almacenamiento etcd estará replicado entre los servidores maestros, ya que se encontrará dentro de los mismos.



14.1. Clonando las máquinas maestras con VirtualBox

Lo primero que vamos a llevar a cabo es la clonación de nuestra máquina maestra **kubernetes-master-clone** con VirtualBox.

El primer clon lo vamos a llamar **kubernetes-master2-clone**

El segundo clon lo vamos a llamar **kubernetes-master3-clone**

De forma que tendremos 3 máquinas maestras cargadas en VirtualBox con los nombres:

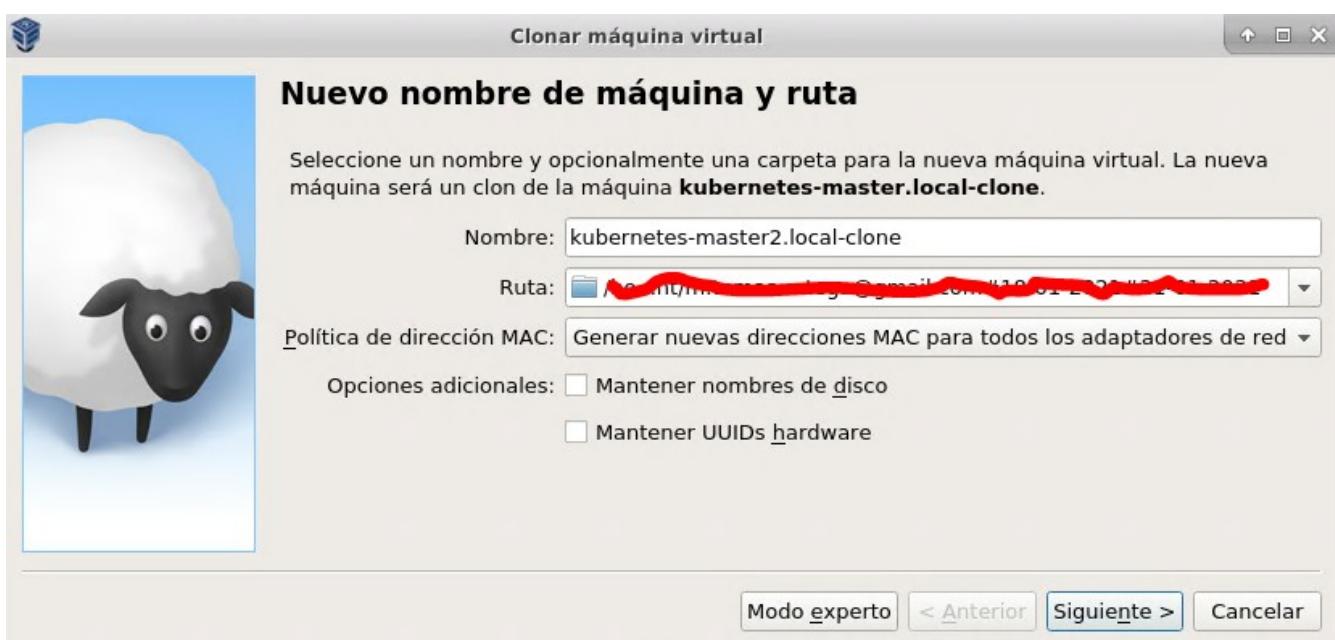
- **kubernetes-master-clone**
- **kubernetes-master2-clone**
- **kubernetes-master3-clone**



La máquina sobre la que posteriormente iniciaremos el clúster será la **kubernetes-master-clone**

Seleccionamos la máquina maestra desde la propia interfaz de VirtualBox y seguidamente pulsamos sobre el menú de **Máquina > Clonar**, indicando las siguientes opciones:

- Nombre
 - kubernetes-master2.local-clone
- Ruta
 - Indicamos la ruta donde tengamos nuestras máquinas virtuales
- Política de dirección MAC
 - Generar nuevas direcciones MAC para todos los adaptadores de red



Pulsamos sobre el botón **Siguiente**

Indicamos que queremos **Clonación completa** y pulsamos sobre el botón **Clonar**

Clonar máquina virtual



Tipo de clonación

Seleccione el tipo de clonación que desea crear.

Si selecciona **Clonación completa**, una copia exacta (incluyendo todos los archivos de disco duro virtual) de la máquina original serán creados.

Si selecciona **Clonación enlazada**, una nueva máquina será creada, pero los archivos de las unidades de disco duro virtuales serán vinculados a los archivos de disco duro virtual de la máquina original y no podrá mover la nueva máquina virtual a una computadora diferente sin mover los originales también.

Si crea una **Clonación enlazada** entonces una nueva instantánea será creada en la máquina virtual original como parte del proceso de clonación.

Clonación completa
 Clonación enlazada

[**< Anterior**](#) [**Clonar**](#) [**Cancelar**](#)



Repetimos los pasos de un nuevo clon a partir de la máquina **kubernetes-master-clone** para obtener la máquina **kubernetes-master3-clone**

14.2. Clonando el balanceador de carga (haproxy)

Para instalar y poner en marcha nuestro balanceador de carga haproxy vamos a alojarlo en una máquina separada que haga dicha función.

Clonamos una máquina realizando el mismo procedimiento que para las máquinas maestras, en este caso, vamos a partir de la máquina **kubernetes-minion1-clone** y le pondremos de nombre al clon **kubernetes-loadbalancer-clone**

14.3. Asignación de recursos a las máquinas virtuales

Para que el clúster vaya bien y kubernetes no se queje de que las máquinas tienen pocos recursos para que operen con normalidad, vamos a asignar las siguientes cuotas de CPU y RAM a las máquinas:

- **kubernetes-master-clone**

- CPU: 2
- RAM: 3072 MB

- **kubernetes-master2-clone**

- CPU: 2
- RAM: 3072 MB

- **kubernetes-master3-clone**

- CPU: 2
- RAM: 3072 MB

- **kubernetes-minion1-clone**

- CPU: 2
- RAM: 4096 MB

- **kubernetes-minion2-clone**

- CPU: 2
 - RAM: 2048 MB
- **kubernetes-loadbalancer-clone**
- CPU: 2
 - RAM: 2048 MB

14.4. Ajustando el direccionamiento IP a las máquinas virtuales

El siguiente paso va a ser llevar a cabo el pertinente ajuste en el direccionamiento IP de nuestros servidores.

Asignaremos IP fija y nombre de host en todos los servidores, de forma que todas las máquinas puedan actuar como resolvidores DNS y no haya ningún problema en el tráfico de red entre las mismas.

Editamos en todas las máquinas el archivo **/etc/hosts** y le asignamos el siguiente contenido:

```
127.0.0.1 localhost localhost.localdomain localhost4 localhost4.localdomain4
::1      localhost localhost.localdomain localhost6 localhost6.localdomain6

192.168.15.100 kubernetes-master1
192.168.15.101 kubernetes-master2
192.168.15.102 kubernetes-master3
192.168.15.103 kubernetes-loadbalancer
192.168.15.104 kubernetes-minion1
192.168.15.105 kubernetes-minion2
```

Arrancamos la primera máquina **kubernetes-master-clone**:

Cambiamos el contenido del archivo **/etc/hosts** por el que hemos comentado.

```
$ sudo nano /etc/hosts
```

Seguidamente, cambiamos el nombre de host de la máquina **kubernetes-master-clone**, editamos el archivo **/etc/hostname** e indicamos el nombre que tendrá la propia máquina **kubernetes-master**

```
$ sudo nano /etc/hostname
```

A continuación, vamos a ajustar la dirección IP estática correcta para cada máquina virtual.

Listamos en primer lugar las interfaces de red de la máquina y nos quedamos con el UUID de la interfaz con nombre **enp0s3**:

```
$ nmcli connection show -a  
NAME      UUID           TYPE      DEVICE  
enp0s3   44866789-3614-4231-a686-55f47245b0c4  ethernet  enp0s3
```

Seguidamente, indicamos la dirección IP en cuestión y el UUID al cual hacemos referencia:

```
$ sudo nmcli connection modify 44866789-3614-4231-a686-55f47245b0c4 IPv4.address 192.168.15.100/24
```



Tendremos que ajustar en cada máquina su ip fija correspondiente que hemos previamente establecido como criterio de asignación a cada una.

14.5. Desactivando systemd-resolved

systemd-resolved es un servicio de systemd que proporciona resolución de nombres de red a aplicaciones locales a través de una interfaz D-Bus , el servicio NSS de resolve (nss-resolve(8)), y un receptor de escucha de DNS local en 127.0.0.53.

Para que a los Pods CoreDNS no tengan problemas a la hora de seleccionar el sistema de resolución DNS, y que utilicen el interno del propio kubernetes, debemos de desactivar dicha funcionalidad en el sistema operativo.

Podemos encontrar más información en la siguiente URL: <https://coredns.io/plugins/loop/#troubleshooting>

Ejecutamos la siguiente instrucción:

```
# systemctl disable --now systemd-resolved
```

Adicionalmente, editamos el archivo **/etc/NetworkManager/NetworkManager.conf** y ajustamos en la sección **[main]** lo siguiente:

```
# nano /etc/NetworkManager/NetworkManager.conf  
  
[main]  
dns=default
```

Ahora, eliminamos el archivo **/etc/resolv.conf** y lo creamos de nuevo limpio:

```
# unlink /etc/resolv.conf  
# touch /etc/resolv.conf
```

14.6. Estableciendo registry para imágenes no FQDN

En los sistemas operativos modernos, como es el caso de la distribución fedora que utilizamos para el curso, cuando tenemos imágenes que se consideran "no correctamente cualificadas", esto quiere decir, imágenes docker que no llevan en el nombre el prefijo del registry del cual deben de obtenerse, sucede que el sistema operativo puede tener un orden de preferencia por defecto para dirigirse a un registry concreto y puede que ese registry no sea el que nosotros queremos, provocando en la mayoría de los casos, errores del tipo "acceso no autorizado", o bien, del tipo "imagen no encontrada".

Editamos el archivo que se encuentra en la ruta: `/etc/containers/registries.conf` y comentamos la siguiente línea añadiendo un carácter almohadilla delante (#):

```
unqualified-search-registries = ["registry.fedoraproject.org", "registry.access.redhat.com", "docker.io", "quay.io"]
```

Seguidamente, añadimos la siguiente línea justo debajo:

```
unqualified-search-registries = ["docker.io"]
```

Con esto, estamos indicándole al sistema operativo, y más concretamente, al motor de contenedores, que cuando se intente descargar imágenes que no se consideren full FQDN (que no tengan la dns del registry como prefijo en el nombre de la imagen docker), que se vaya por defecto al registry docker.io para proceder a su descarga.

Llegado este punto reiniciamos el servidor ejecutando en consola el siguiente comando:

```
# shutdown -r now
```

Reiniciamos la máquina, si esta ha arrancado, la apagamos, encendemos la siguiente y repetimos el procedimiento, ajustando los datos específicos para la misma.

14.7. Instalando el balanceador de carga (haproxy)

Vamos a instalar y configurar el balanceador de carga haproxy.

Nos conectamos a la máquina **kubernetes-loadbalancer-clone** y ejecutamos el siguiente comando:

```
$ sudo yum install haproxy -y
```

Seguidamente, vamos a editar el archivo de configuración del balanceador

/etc/haproxy/haproxy.cfg

```
$ sudo nano /etc/haproxy/haproxy.cfg
```

Y sustituímos todo el contenido por este:

```
global
    user haproxy
    group haproxy

defaults
    mode http
    log global
    retries 2
    timeout connect 3000ms
    timeout server 5000ms
    timeout client 5000ms

frontend kubernetes
    bind *:6443
    option tcplog
    mode tcp
    default_backend kubernetes-master-nodes

backend kubernetes-master-nodes
    mode tcp
    balance roundrobin
    option tcp-check
    server kubernetes-master1 192.168.15.100:6443 check fall 3 rise 2
    server kubernetes-master2 192.168.15.101:6443 check fall 3 rise 2
    server kubernetes-master3 192.168.15.102:6443 check fall 3 rise 2
```

Recargamos los daemons del sistema y reiniciamos el propio servicio haproxy:

```
$ sudo systemctl daemon-reload && sudo systemctl restart haproxy
```

14.8. Configurando el IP Forwarding (en nodos de kubernetes)

Ahora, vamos a proceder a configurar el bridge de iptables y el forwarding de ipv4.

Ejecutamos la siguiente instrucción:

```
#: cat <<EOF > /etc/sysctl.d/k8s.conf  
net.bridge.bridge-nf-call-ip6tables = 1  
net.bridge.bridge-nf-call-iptables = 1  
net.ipv4.ip_forward=1  
EOF
```

Recargamos seguidamente la configuración para no tener que reiniciar el servidor:

```
#: sysctl --system
```

14.9. Deshabilitando la memoria SWAP (en nodos de kubernetes)



Procedemos a continuación con la desactivación de la memoria swap del servidor.

Ejecutamos el siguiente comando y a continuación reiniciamos el servidor:

```
#: dnf remove zram-generator-defaults
```

Una vez reiniciado el servidor, ejecutamos el siguiente comando para asegurarnos de que la swap no ha sido cargada al inicio del sistema:

```
# zramctl
```

Si no indica nada, es que la swap está correctamente desactivada.

14.10. Instalando los paquetes de Kubernetes



Este procedimiento se realizará tanto en las máquinas maestras como en los minions.

En primer lugar, pasamos a modo root en la consola:

```
$ sudo su -
```

```
# dnf install kubernetes-kubeadm cri-tools iproute-tc -y
```



Consideramos que está ya instalado el motor de contenedores, sino habría que instalarlo

Seguidamente, vamos a indicar al sistema operativo que si se reinicia, debe de levantar de forma



automática el servicio de kubelet, esta operación hay que realizarla en todos los nodos:

```
# systemctl enable kubelet
```

14.11. Creando el archivo de configuración para la red CNI de kubernetes

Tanto en las máquinas maestras como en las minions, vamos a crear la siguiente estructura de directorio:

```
# mkdir -p /etc/cni/net.d/
```

Tanto en las máquinas maestras como en las minions, vamos a crear el archivo **/etc/cni/net.d/10-weave.conflist**

```
$ sudo nano /etc/cni/net.d/10-weave.conflist
```

Le añadimos el siguiente contenido:

```
{
  "cniVersion": "0.3.0",
  "name": "weave",
  "plugins": [
    {
      "name": "weave",
      "type": "weave-net",
      "hairpinMode": true
    },
    {
      "type": "portmap",
      "capabilities": {"portMappings": true},
      "snat": true
    }
  ]
}
```

Actualmente se produce un error en la puesta en marcha de un clúster stacked de kubernetes en modo HA con varios máster.

Al incorporar al segundo máster, parece que hay un problema de acceso de permisos al directorio de configuración de la red CNI que acabamos de crear.

Por el momento, la comunidad lo está solvendando indicando a dicha carpeta los permisos 777 hasta que se corrija el problema.

Ejecutamos en las máquinas maestras y los minions:

```
$ sudo chmod 777 /etc/cni/net.d
```

14.12. Realizando limpieza de Iptables

Partimos de una instalación limpia, pero no viene mal ejecutar el siguiente comando en todos los servidores master y minions que van a formar parte del clúster, para dejar las Iptables por defecto y ya posteriormente que kubernetes o nosotros, vayamos ajustando lo que se vaya necesitando.

```
$ sudo iptables -F
```

14.13. Inicio del cluster

El inicio del clúster vamos a llevarlo a cabo desde la máquina maestra **kubernetes-master-clone** que fue la primera máquina a partir de la cual luego hicimos los clones de las otras.

La que tiene asignada la IP 192.168.15.100.

Ejecutamos el siguiente comando:

```
# kubeadm init --pod-network-cidr=10.95.10.0/16 --apiserver-advertise-address "192.168.15.100" --control-plane-endpoint  
"192.168.15.103:6443" --upload-certs
```

- **--pod-network-cidr**

- Indicamos el segmento de red para los pods de usuario que el clúster creará

- **--apiserver-advertise-address**

- Indicamos como referencia de inicio del clúster la ip de la máquina donde lanzamos el comando

- **--control-plane-endpoint**

- Indicamos la IP y el puerto por el que vamos a conectar contra el balanceador de carga

- **--upload-certs**

- Indicamos que queremos crear un objeto de tipo secret en el clúster, donde estarán los certificados que cifran las comunicaciones del clúster de kubernetes, de manera que cuando se vayan incorporando más máquinas maestras o minions, estas los puedan utilizar para establecer comunicaciones de forma distribuida y cifrada

Si todo ha ido bien, debería de aparecer una traza en la consola parecida a esta:

...

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
```

Alternatively, if you are the root user, you can run:

```
export KUBECONFIG=/etc/kubernetes/admin.conf
```

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/>

You can now join any number of the control-plane node running the following command on each as root:

```
kubeadm join 192.168.15.103:6443 --token g5k7nq.ay8g5jpwhy48j9ez \  
--discovery-token-ca-cert-hash sha256:f77ef9030fc6394e281b403c4e2a833ff0027830849ff58dc91517c366e0275a \  
--control-plane --certificate-key cfbf72d887ae45325eade51f4a0994c378f02b4ac16356563bc70f02291e2fe1
```

Please note that the certificate-key gives access to cluster sensitive data, keep it secret!

As a safeguard, uploaded-certs will be deleted in two hours; If necessary, you can use
"kubeadm init phase upload-certs --upload-certs" to reload certs afterward.

Then you can join any number of worker nodes by running the following on each as root:

```
kubeadm join 192.168.15.103:6443 --token g5k7nq.ay8g5jpwhy48j9ez \  
--discovery-token-ca-cert-hash sha256:f77ef9030fc6394e281b403c4e2a833ff0027830849ff58dc91517c366e0275a
```

El sistema nos indica dos comandos a ejecutar, el comando kubeadm join para las máquinas maestras que queramos incorporar así como el comando específico kubeadm join para las máquinas mininos que queramos incorporar.

Para comenzar a interactuar con el clúster desde la máquina maestra que ha iniciado el clúster vamos a ejecutar primero los comandos que nos indica la traza de la consola: **you need to run the following as a regular user**

Seguidamente, ejecutamos en la máquina maestra, mediante el cliente kubectl, la orden de instalación del driver weavenet para kubernetes, ejecutamos el siguiente comando:

```
$ kubectl apply -f "https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version | base64 | tr -d '\n')"  
  
serviceaccount/weave-net created  
clusterrole.rbac.authorization.k8s.io/weave-net created  
clusterrolebinding.rbac.authorization.k8s.io/weave-net created  
role.rbac.authorization.k8s.io/weave-net created  
rolebinding.rbac.authorization.k8s.io/weave-net created  
daemonset.apps/weave-net created
```

Monitorizamos el proceso de instalación y puesta en marcha de los componentes del driver, ejecutamos el comando:

```
$ watch kubectl get pods -n kube-system
```

Cuando observemos que todos los pods están operativos, entonces el procedimiento de instalación del driver habrá finalizado:

NAME	READY	STATUS	RESTARTS	AGE
coredns-74ff55c5b-m9dpz	1/1	Running	0	8m48s
coredns-74ff55c5b-q9p42	1/1	Running	0	8m48s
etcd-kubernetes-master.local-clone	1/1	Running	0	8m55s
kube-apiserver-kubernetes-master.local-clone	1/1	Running	0	8m55s
kube-controller-manager-kubernetes-master.local-clone	1/1	Running	0	8m55s
kube-proxy-qcn62	1/1	Running	0	8m48s
kube-scheduler-kubernetes-master.local-clone	1/1	Running	0	8m55s
weave-net-b9vsc	2/2	Running	0	78s

Posteriormente, también comprobamos que los nodos del clúster (en este momento sólo hay uno) están completamente operativos, ejecutamos el comando:

```
$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
kubernetes-master.local-clone	Ready	control-plane,master	9m57s	v1.20.1

14.14. Añadiendo a las máquinas master

En las máquinas máster, ejecutamos el siguiente comando:

```
kubeadm join 192.168.15.103:6443 --token g5k7nq.ay8g5jpwhy48j9ez \
--discovery-token-ca-cert-hash sha256:f77ef9030fc6394e281b403c4e2a833ff0027830849ff58dc91517c366e0275a \
--control-plane --certificate-key cfbf72d887ae45325eade51f4a0994c378f02b4ac16356563bc70f02291e2fe1
```

This node has joined the cluster and a new control plane instance was created:

- * Certificate signing request was sent to apiserver and approval was received.
- * The Kubelet was informed of the new secure connection details.
- * Control plane (master) label and taint were applied to the new node.
- * The Kubernetes control plane instances scaled up.
- * A new etcd member was added to the local/stacked etcd cluster.

To start administering your cluster from this node, 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
```

Run 'kubectl get nodes' to see this node join the cluster.



Cada alumno tendrá que lanzar su propio comando, ya que los token sha256 se generan de forma diferente por cada kubeadm init que hagamos.

14.15. Añadiendo las máquinas minions

En las máquinas máster, ejecutamos el siguiente comando:

```
kubeadm join 192.168.15.103:6443 --token g5k7nq.ay8g5jpwhy48j9ez \
--discovery-token-ca-cert-hash sha256:f77ef9030fc6394e281b403c4e2a833ff0027830849ff58dc91517c366e0275a
```

This node has joined the cluster:

- * Certificate signing request was sent to apiserver and a response was received.
- * The Kubelet was informed of the new secure connection details.

Run 'kubectl get nodes' on the control-plane to see this node join the cluster.



Cada alumno tendrá que lanzar su propio comando, ya que los token sha256 se generan de forma diferente por cada kubeadm init que hagamos.

14.16. Comprobando el estado del clúster

Finalmente, vamos a ejecutar el siguiente comando para verificar que todos los Pods necesarios para la maquinaria de kubernetes están operativos:



Fondos Europeos



Cofinanciado por
la Unión Europea



```
$ kubectl get pod -n kube-system
```

NAME	READY	STATUS	RESTARTS	AGE
coredns-74ff55c5b-m9dpz	1/1	Running	0	23m
coredns-74ff55c5b-q9p42	1/1	Running	0	23m
etcd-kubernetes-master.local-clone	1/1	Running	0	23m
etcd-kubernetes-master2.local-clone	1/1	Running	0	7m57s
etcd-kubernetes-master3.local-clone	1/1	Running	0	4m34s
kube-apiserver-kubernetes-master.local-clone	1/1	Running	0	23m
kube-apiserver-kubernetes-master2.local-clone	1/1	Running	0	7m58s
kube-apiserver-kubernetes-master3.local-clone	1/1	Running	0	4m35s
kube-controller-manager-kubernetes-master.local-clone	1/1	Running	1	23m
kube-controller-manager-kubernetes-master2.local-clone	1/1	Running	0	7m58s
kube-controller-manager-kubernetes-master3.local-clone	1/1	Running	0	4m35s
kube-proxy-5zfvl	1/1	Running	0	68s
kube-proxy-bg9sg	1/1	Running	0	2m29s
kube-proxy-c8tgc	1/1	Running	0	7m59s
kube-proxy-jsqvm	1/1	Running	0	4m35s
kube-proxy-qcn62	1/1	Running	0	23m
kube-scheduler-kubernetes-master.local-clone	1/1	Running	1	23m
kube-scheduler-kubernetes-master2.local-clone	1/1	Running	0	7m58s
kube-scheduler-kubernetes-master3.local-clone	1/1	Running	0	4m35s
weave-net-2g5vv	2/2	Running	0	2m29s
weave-net-48lpj	2/2	Running	0	7m59s
weave-net-b9vsc	2/2	Running	0	15m
weave-net-j9lsz	2/2	Running	0	4m35s
weave-net-zbpm8	2/2	Running	1	68s

También ejecutamos el siguiente comando, para observar el estado de los nodos:

```
$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
kubernetes-master.local-clone	Ready	control-plane,master	25m	v1.20.1
kubernetes-master2.local-clone	Ready	control-plane,master	10m	v1.20.1
kubernetes-master3.local-clone	Ready	control-plane,master	6m49s	v1.20.1
kubernetes-minion1.local-clone	Ready	<none>	4m42s	v1.20.1
kubernetes-minion2.local-clone	Ready	<none>	3m21s	v1.20.1

Capítulo 15. Driver de red: Weave



URL: <https://www.weave.works/docs/net/latest/overview/>

Weave Net crea una red virtual que conecta contenedores Docker a través de múltiples hosts y permite su descubrimiento automático.

Con Weave Net, las aplicaciones portátiles basadas en microservicios que constan de múltiples contenedores pueden ejecutarse en cualquier lugar:

- En un host
- Múltiples hosts
- Proveedores de nube y centros de datos

Las aplicaciones utilizan la red como si todos los contenedores estuvieran conectados al mismo conmutador de red, sin tener que configurar asignaciones de puertos o enlaces.

Los servicios proporcionados por contenedores de aplicaciones en la red pueden exponerse al mundo exterior, independientemente de dónde se estén ejecutando.

De manera similar, los sistemas internos existentes se pueden abrir para aceptar conexiones desde contenedores de aplicaciones independientemente de su ubicación.

15.1. ¿Cómo funciona?

Weave Net crea una nueva red de Capa 2 utilizando las características del kernel de Linux; un demonio configura esa red y administra el enruteamiento entre máquinas y hay varias formas de conectarse a esa red (por ejemplo, un complemento CNI que se ejecuta como un proceso separado).

También un demonio más si está utilizando la política de red de Kubernetes.

15.2. Configuración Simple

Weave Net simplifica la configuración de una red de contenedores.

Debido a que los contenedores en una red Weave usan números de puerto estándar (por ejemplo, el puerto predeterminado de MySQL es 3306), administrar microservicios es sencillo.

Cada contenedor puede encontrar la IP de cualquier otro contenedor usando una simple consulta de DNS en el nombre del contenedor, y también puede comunicarse directamente sin NAT, sin usar mapeos de puertos o complicados enlaces de puertas de enlace.

Y lo mejor de todo, implementar una red de contenedores Weave no requiere cambios en el código de nuestra aplicación.

15.3. Descubrimiento DNS

Weave Net implementa el descubrimiento de servicios al proporcionar un servidor "micro DNS" rápido en cada nodo.

Simplemente nombra los contenedores y todo "simplemente funciona", incluido el balanceo de carga en varios contenedores con el mismo nombre.



Capítulo 16. Lab: Instalación del Driver de red Weave en Kubernetes

Mediante este laboratorio, vamos a llevar a cabo la instalación del driver de red Weave en nuestro clúster de kubernetes.

16.1. Aplicando el manifiesto

El driver de Weave se instalará en nuestro clúster mediante el uso de objetos DaemonSets, de manera que introducirá un Pod en cada uno de los servidores para orquestar la maquinaria de propio driver.

Ejecutamos el siguiente comando en la consola:

```
$ kubectl apply -f https://github.com/weaveworks/weave/releases/download/v2.8.1/weave-daemonset-k8s.yaml

serviceaccount/weave-net created
clusterrole.rbac.authorization.k8s.io/weave-net created
clusterrolebinding.rbac.authorization.k8s.io/weave-net created
role.rbac.authorization.k8s.io/weave-net created
rolebinding.rbac.authorization.k8s.io/weave-net created
daemonset.apps/weave-net created
```

16.2. Comprobando el inventario de los nodos

Seguidamente, vamos a obtener el inventario de los Pods del namespace de **kube-system**, donde los Pods que forman parte de la maquinaria de kubernetes deberían de aparecer en estado **Running**

```
$ kubectl get pods --all-namespaces
NAMESPACE     NAME                               READY   STATUS    RESTARTS   AGE
kube-system   coredns-86c58d9df4-9gcmd        1/1    Running   0          9m22s
kube-system   coredns-86c58d9df4-gqrm9        1/1    Running   0          9m22s
kube-system   etcd-kubemaster.local           1/1    Running   0          8m14s
kube-system   kube-apiserver-kubemaster.local  1/1    Running   0          8m22s
kube-system   kube-controller-manager-kubemaster.local  1/1    Running   0          8m32s
kube-system   kube-proxy-gpk8z                1/1    Running   0          9m22s
kube-system   kube-proxy-2cbbh                1/1    Running   0          9m15s
kube-system   kube-proxy-7z7rs                1/1    Running   0          9m10s
kube-system   kube-scheduler-kubemaster.local  1/1    Running   0          8m37s
kube-system   weave-net-8tjrg                 2/2    Running   0          9m20s
kube-system   weave-net-bgg8p                 2/2    Running   0          9m10s
kube-system   weave-net-k4cjd                 2/2    Running   0          9m5s
```

Posteriormente, vamos a obtener el inventario de los nodos, donde todos deberían de aparecer en estado **Ready**.

```
$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
kubemaster-master.local	Ready	master	14m	v1.21.1
kubernetes-minion1.local	Ready	<none>	103s	v1.21.1
kubernetes-minion2.local	Ready	<none>	96s	v1.21.1



Capítulo 17. Federación con Kops



Las siglas Kops, vienen de Kubernetes Operations

Se trata de una nueva herramienta para hacer más fácil la tarea de la puesta en funcionamiento de un clúster de Kubernetes en producción.

Kops nos va a ayudar a crear, destruir, actualizar y mantener clústeres de Kubernetes de alta disponibilidad mediante la línea de comandos.

Kops está comenzando a operar con los siguientes operadores de cloud:

- AWS (Amazon Web Services)
 - Actualmente es oficialmente soportado
- GCE (Google Compute Engine)
 - Actualmente en fase beta
- OpenStack
 - Actualmente en fase beta
- VMWare
 - Actualmente en fase alfa
- vSphere
 - Actualmente en fase alfa

Paralelamente para otras plataformas se está planificando su compatibilidad.

Podemos encontrar información del proyecto en la web de GitHub: <https://github.com/kubernetes/kops>

17.1. Requisitos para su uso

Al ser un proyecto bastante reciente, la recomendación es utilizar siempre la última versión

liberada estable de Kops, con siempre la misma versión de kubernetes en todos los nodos que compondrán el clúster.

A fecha de hoy, para que Kops pueda operar de forma correcta:

- La versión mínima necesaria de kubernetes: 1.12.x
- La versión máxima posible de kubernetes: 1.16.x

Kops está diseñado para mantener retrocompatibilidad con versiones anteriores.

En cuanto al uso de Kubernetes, se sugiere utilizar alguna de las 3 versiones "minor version".

Con el transcurso del tiempo, Kops deja de ser compatible con versiones anteriores (ahora mismo las anteriores a la versión 1.12)

Respecto a la compatibilidad hay una excepción, Kops admite el número de versión menor de Kubernetes.

Una versión menor es el segundo dígito en el número de versión.

La versión 1.13.0 de Kops tiene una versión menor de 13.

La numeración sigue la especificación de versiones semánticas <MAJOR.MINOR.PATCH>

En resumen:

- Una versión de Kops 1.12.x
 - Compatible con Kubernetes hasta la versión 1.12x
 - Puede funcionar con: Kubernetes 1.12
- Una versión de Kops 1.13.x
 - Compatible con Kubernetes hasta la versión 1.13.x
 - Puede funcionar con: Kubernetes 1.12 | Kubernetes 1.13
- Una versión de Kops 1.14.x
 - Compatible con Kubernetes hasta la versión 1.14.x
 - Puede funcionar con: Kubernetes 1.12 | Kubernetes 1.13 | Kubernetes 1.14

17.2. RoadMap de liberación de versiones

El RoadMap del proyecto de Kops no sigue la agenda de liberación de las versiones de Kubernetes.

Kops tiene como objetivo proporcionar una experiencia confiable para Kubernetes y generalmente se publica aproximadamente un mes después del lanzamiento de la versión de Kubernetes correspondiente o incluso más.

La idea, es que primero se libere la versión de Kubernetes, se evalúe y resuelvan las posibles incidencias detectadas, para posteriormente garantizar que se puedan admitir las nuevas funcionalidades que el propio Kubernetes traiga.

Kops irá lanzando versiones preliminares alfa y beta para quien quiera ir probando el software, pero **no se recomienda en absoluto su uso en producción.**



Capítulo 18. Lab: Federación con Kops

Mediante este laboratorio, pondremos en marcha un clúster en AWS (Amazon Web Services) compuesto por algunas máquinas que tendrán el rol de master y otras que tendrán el rol de minions (workers).

18.1. Instalación del cliente de Amazon (awscli)

Para poder interactuar con AWS, lo primero que necesitamos es tener instalado el cliente de consola (awscli).

Lo primero que vamos a comprobar es que tenemos python instalado, ejecutamos el siguiente comando:

```
$ python -V  
Python 2.7.5
```

A continuación, instalamos el gestor de paquetes de Python:

```
$ sudo yum -y install python2-pip
```

Seguidamente, instalamos el módulo awscli:

```
$ sudo pip install awscli
```

Llevamos a cabo una comprobación de que la herramienta se encuentre perfectamente instalada:

```
$ aws --version  
aws-cli/1.17.2 Python/2.7.5 Linux/3.10.0-1062.4.1.el7.x86_64 botocore/1.14.2
```

18.2. Configuración del cliente de Amazon (awscli)

Para llevar a cabo la configuración del cliente de consola, vamos a ejecutar el siguiente comando:

```
$ aws configure  
AWS Access Key ID [None]: *****  
AWS Secret Access Key [None]: *****  
Default region name [None]: us-east-2  
Default output format [None]: json
```

- Indicamos nuestra Access Key
- Indicamos nuestra Secret Key

- Indicamos como nombre de la región **us-east-2**
- Indicamos como formato de salida de consola **json**

A continuación, vamos a consultar las instancias ec2 que tengamos activas (en principio no debería de haber ninguna):

```
[aws@localhost ~]$ aws ec2 describe-instances
{
  "Reservations": []
}
```

18.3. Configurando recursos (Gestión de identidades y accesos IAM)

Ahora, vamos a crear algunos recursos de Gestión de identidades y accesos (IAM).

Aunque se pueda crear el clúster con el usuario que utilizamos para registrarnos en AWS, es una buena práctica crear una cuenta a parte que contenga solo los privilegios necesarios para ejecutar las siguientes instrucciones.

Ejecutamos la siguiente instrucción para crear un grupo IAM llamado kops:

```
$ aws iam create-group --group-name kops
{
  "Group": {
    "Path": "/",
    "CreateDate": "2020-03-18T19:31:48Z",
    "GroupId": "AGPA3U4WDRVVQIYHRND5",
    "Arn": "arn:aws:iam::800789335147:group/kops",
    "GroupName": "kops"
  }
}
```

Ahora, ajustamos políticas de control de acceso del grupo kops a los recursos de Amazon que vamos a necesitar:

- EC2 (Instancias de computación)
- S3 (Almacenamiento)
- VPC (Red)
- IAM (Control de acceso)

Ejecutamos las siguientes instrucciones para añadir las políticas al grupo Kops:

```
$ aws iam attach-group-policy --group-name kops \
--policy-arn arn:aws:iam::aws:policy/AmazonEC2FullAccess

$ aws iam attach-group-policy --group-name kops \
--policy-arn arn:aws:iam::aws:policy/AmazonS3FullAccess

$ aws iam attach-group-policy --group-name kops \
--policy-arn arn:aws:iam::aws:policy/AmazonVPCFullAccess

$ aws iam attach-group-policy --group-name kops \
--policy-arn arn:aws:iam::aws:policy/IAMFullAccess
```

18.4. Configurando el usuario de Amazon

Una vez tenemos el grupo con los permisos suficientes, creamos el nuevo usuario que se añadirá a este:

```
$ aws iam create-user --user-name kopsuser
{
  "User": {
    "UserName": "kopsuser",
    "Path": "/",
    "CreateDate": "2020-03-18T19:35:41Z",
    "UserId": "AIDA3U4WDRRVRFGGX3WL2",
    "Arn": "arn:aws:iam::800789335147:user/kopsuser"
  }
}
```

Ahora, añadimos el usuario de aws al grupo kops:

```
$ aws iam add-user-to-group --user-name kopsuser --group-name kops
```

A continuación, generamos unas claves de acceso para el nuevo usuario que hemos creado, sin ellas no podríamos actuar en nombre de dicho usuario:

```
$ aws iam create-access-key \
--user-name kopsuser > kopsuser-credentials
```

Las claves de acceso se han creado y almacenado en el fichero con el nombre que le hemos indicado "kops-credentials", le echamos un vistazo al contenido del fichero:

```
$ cat kopsuser-credentials
{
  "AccessKey": {
    "UserName": "kopsuser",
    "Status": "Active",
    "CreateDate": "2020-03-18T19:37:56Z",
    "SecretAccessKey": "UY8vT4A6GaQyLaHUY++SelAnMQEx8y07sm11GL3O",
    "AccessKeyId": "AKIA3U4WDRRVSYKHZVU"
  }
}
```

Ahora necesitamos tomar el valor de las entradas SecretAccessKey y AccessKeyId. El siguiente paso es parsear el contenido del fichero kopsuser-credentials y almacenar los dos valores como variables de entorno AWS_ACCESS_KEY_ID y AWS_SECRET_ACCESS_KEY.

La correspondencia con las credenciales de Amazon sería la siguiente:

- **AWS_ACCESS_KEY_ID**
 - AccessKeyId
- **AWS_SECRET_ACCESS_KEY**
 - SecretAccessKey

Reconfiguramos ahora el acceso a Amazon, de nuevo lanzamos el comando:

```
$ aws configure
```

- Cuando nos pregunte la consola, introducimos las nuevas AWS_ACCESS_KEY_ID y AWS_SECRET_ACCESS_KEY

El siguiente paso será decidir qué zonas de disponibilidad vamos a usar, para ello podemos ver las zonas disponibles dentro de la región que hemos pasado por la variable, en este caso **us-east-2**:

```
$ aws ec2 describe-availability-zones --region us-east-2
{
  "AvailabilityZones": [
    {
      "OptInStatus": "opt-in-not-required",
      "Messages": [],
      "ZoneId": "use2-az1",
      "GroupName": "us-east-2",
      "State": "available",
      "NetworkBorderGroup": "us-east-2",
      "ZoneName": "us-east-2a",
      "RegionName": "us-east-2"
    },
    {
      "OptInStatus": "opt-in-not-required",
      "Messages": [],
      "ZoneId": "use2-az2",
      "GroupName": "us-east-2",
      "State": "available",
      "NetworkBorderGroup": "us-east-2",
      "ZoneName": "us-east-2b",
      "RegionName": "us-east-2"
    },
    {
      "OptInStatus": "opt-in-not-required",
      "Messages": [],
      "ZoneId": "use2-az3",
      "GroupName": "us-east-2",
      "State": "available",
      "NetworkBorderGroup": "us-east-2",
      "ZoneName": "us-east-2c",
      "RegionName": "us-east-2"
    }
  ]
}
```

Como podemos ver existen tres zonas de disponibilidad

18.5. Configurando par de claves SSH

Para empezar a crear el cluster, vamos a crear un par de claves para hacer la conexión SSH y acceder a las instancias EC2.

Vamos a crear un directorio dedicado para la creación del cluster.

```
mkdir -p cluster
cd cluster
```

Creamos las claves SSH con el comando aws ec2:

```
$ aws ec2 create-key-pair --key-name proyecto --query 'KeyMaterial' --output text > proyecto.pem
```

- El comando indicado, dejará únicamente el contenido de una Key en formato JSON que se llama 'KeyMaterial' (La clave privada)

Por razones de seguridad cambiamos los permisos del fichero para que solo pueda ser leído por el usuario actual:

```
$ chmod 400 proyecto.pem
```

Finalmente, solo necesitamos el segmento público de las claves generada, para extraer dicha parte usaremos ssh-keygen:

```
$ ssh-keygen -y -f proyecto.pem > proyecto.pub
```

- Ahora ya tenemos la clave pública (proyecto.pub) y la privada (proyecto.pem)

Nos falta crear el bucket, obtenemos la URL de acceso a los recursos para el proyecto, para crear el bucket en AWS, utilizamos la siguiente instrucción:

```
$ aws s3api create-bucket --bucket proyecto-kops --create-bucket-configuration LocationConstraint=us-east-2
{
  "Location": "http://proyecto-kops.s3.amazonaws.com/"
}
```

18.6. Instalando Kops

A continuación, vamos a proceder a instalar Kops.

Descargamos la release que nos interese:

```
$ wget https://github.com/kubernetes/kops/releases/download/v1.16.0/kops-linux-amd64
```

Otorgamos permisos de ejecución al archivo:

```
$ chmod +x kops-linux-amd64
```

Movemos el binario a la carpeta de binarios locales del sistema, para poder ejecutar kops desde cualquier ubicación:

```
$ sudo mv kops-linux-amd64 /usr/local/bin/kops
```

Ahora, verificamos que el binario de Kops se ejecuta correctamente, comprobamos la versión de Kops:

```
$ kops version  
Version 1.16.0 (git-4b0e62b82)
```

Llegados a este punto, toca crear el clúster, ejecutamos el siguiente comando en la consola:



```
$ kops create cluster --name=projeto.k8s.local --master-count 3 --node-count 2 --node-size t2.small  
--master-size t2.small --zones=us-east-2a --master-zones=us-east-2a --ssh-public-key proyecto.pub --networking  
kubenet --kubernetes-version v1.16.0 --state s3://projeto-kops --authorization RBAC --yes
```

```
I0318 22:06:05.107454 12254 create_cluster.go:1568] Using SSH public key: proyecto.pub  
I0318 22:06:06.860432 12254 create_cluster.go:562] Inferred --cloud=aws from zone "us-east-2a"  
W0318 22:06:06.860720 12254 create_cluster.go:772] Running with masters in the same AZs; redundancy will be reduced  
I0318 22:06:07.467027 12254 subnets.go:184] Assigned CIDR 172.20.32.0/19 to subnet us-east-2a
```

```
*****
```

A new kubernetes version is available: 1.16.7
Upgrading is recommended (try kops upgrade cluster)

More information: https://github.com/kubernetes/kops/blob/master/permalinks/upgrade_k8s.md#1.16.7

```
*****
```

```
I0318 22:06:13.803444 12254 apply_cluster.go:556] Gossip DNS: skipping DNS validation  
I0318 22:06:17.096688 12254 executor.go:103] Tasks: 0 done / 102 total; 48 can run  
I0318 22:06:20.094213 12254 vfs_castore.go:729] Issuing new certificate: "etcd-manager-ca-main"  
I0318 22:06:21.050250 12254 vfs_castore.go:729] Issuing new certificate: "etcd-peers-ca-main"  
I0318 22:06:21.959568 12254 vfs_castore.go:729] Issuing new certificate: "apiserver-aggregator-ca"  
I0318 22:06:22.343158 12254 vfs_castore.go:729] Issuing new certificate: "ca"  
I0318 22:06:23.186730 12254 vfs_castore.go:729] Issuing new certificate: "etcd-manager-ca-events"  
I0318 22:06:23.446929 12254 vfs_castore.go:729] Issuing new certificate: "etcd-peers-ca-events"  
I0318 22:06:23.589733 12254 vfs_castore.go:729] Issuing new certificate: "etcd-clients-ca"  
I0318 22:06:24.723134 12254 executor.go:103] Tasks: 48 done / 102 total; 24 can run  
I0318 22:06:28.058938 12254 vfs_castore.go:729] Issuing new certificate: "kube-controller-manager"  
I0318 22:06:29.369568 12254 vfs_castore.go:729] Issuing new certificate: "kops"  
I0318 22:06:29.542139 12254 vfs_castore.go:729] Issuing new certificate: "kube-scheduler"  
I0318 22:06:31.266648 12254 vfs_castore.go:729] Issuing new certificate: "apiserver-proxy-client"  
I0318 22:06:31.841421 12254 vfs_castore.go:729] Issuing new certificate: "kubelet"  
I0318 22:06:32.122398 12254 vfs_castore.go:729] Issuing new certificate: "kubecfg"  
I0318 22:06:32.974919 12254 vfs_castore.go:729] Issuing new certificate: "apiserver-aggregator"  
I0318 22:06:33.068927 12254 vfs_castore.go:729] Issuing new certificate: "kube-proxy"  
I0318 22:06:33.955775 12254 vfs_castore.go:729] Issuing new certificate: "kubelet-api"  
I0318 22:06:35.076709 12254 executor.go:103] Tasks: 72 done / 102 total; 22 can run  
I0318 22:06:37.722518 12254 executor.go:103] Tasks: 94 done / 102 total; 5 can run  
I0318 22:06:40.030018 12254 vfs_castore.go:729] Issuing new certificate: "master"  
I0318 22:06:41.246870 12254 executor.go:103] Tasks: 99 done / 102 total; 3 can run  
W0318 22:06:48.156757 12254 executor.go:128] error running task "LoadBalancerAttachment/api-master-us-east-2a-3" (9m53s  
remaining to succeed): error attaching autoscaling group to ELB: ValidationError: Provided Load Balancers may not be  
valid. Please ensure they exist and try again.
```

status code: 400, request id: 94c6ad55-d132-428d-84c8-10a9e7f57f6f

```
W0318 22:06:48.156955 12254 executor.go:128] error running task "LoadBalancerAttachment/api-master-us-east-2a-2" (9m53s  
remaining to succeed): error attaching autoscaling group to ELB: ValidationError: Provided Load Balancers may not be  
valid. Please ensure they exist and try again.
```

status code: 400, request id: a5b259a0-f42c-4bfc-9749-c64979303634

```
I0318 22:06:48.157057 12254 executor.go:103] Tasks: 100 done / 102 total; 2 can run  
I0318 22:06:48.744589 12254 executor.go:103] Tasks: 102 done / 102 total; 0 can run  
I0318 22:06:49.154392 12254 update_cluster.go:305] Exporting kubecfg for cluster  
kops has set your kubectl context to projeto.k8s.local
```

Cluster is starting. It should be ready in a few minutes.

Suggestions:

- * validate cluster: kops validate cluster
- * list nodes: kubectl get nodes --show-labels
- * ssh to the master: ssh -i ~/.ssh/id_rsa admin@api.projeto.k8s.local
- * the admin user is specific to Debian. If not using Debian please use the appropriate user based on your OS.
- * read about installing addons at: <https://github.com/kubernetes/kops/blob/master/docs/operations/addons.md>.

- Una vez finalizada la ejecución del comando, tendremos nuestro clúster operando y totalmente

funcional con la configuración que hemos indicado.

Obtenemos información del cluster:

```
$ kops get cluster --state s3://proyecto-kops
```

NAME	CLOUD ZONES
proyecto.k8s.local	aws us-east-2a

A continuación, pasamos a consultar los nodos del clúster con la herramienta kubectl, como si hubiéramos aprovisionado el clúster con kubeadm:

```
$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
ip-172-20-35-57.us-east-2.compute.internal	Ready	master	104s	v1.16.0
ip-172-20-41-200.us-east-2.compute.internal	Ready	master	45s	v1.16.0
ip-172-20-49-110.us-east-2.compute.internal	Ready	node	25s	v1.16.0
ip-172-20-51-172.us-east-2.compute.internal	Ready	master	90s	v1.16.0
ip-172-20-56-154.us-east-2.compute.internal	Ready	node	28s	v1.16.0

- Observamos los 6 servidores que hemos aprovisionado con kops, tanto los 3 máster, como los 2 minions

Por último, para borrar el clúster con todos los componentes que la herramienta Kops haya podido crear, ejecutamos el siguiente comando:

```
$ kops delete cluster --name=proyecto.k8s.local --state=s3://proyecto-kops --yes
```

Capítulo 19. kubectl describe node

La operación de describe nos permitirá conocer el estado del nodo a todo detalle

19.1. Listando los nodos

Desde cualquier nodo maestro, podemos ejecutar la instrucción que nos muestre la lista de nodos por los que está compuesto el clúster

```
$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
kubemaster	Ready	master	23h	v1.16.0
kubeminion1	Ready	<none>	22h	v1.16.0
kubeminion2	Ready	<none>	22h	v1.16.0



19.2. Etiquetando los roles de los nodos

Como observamos, en el listado de nodos que nos aparece, existe una columna que se llama **ROLES**, esta columna lo que nos quiere decir, son las etiquetas que tiene cada nodo (labels)

Kubernetes ha etiquetado el nodo maestro con la etiqueta "master" por defecto

Las etiquetas en los nodos se aplican en par (key - value)

Ahora vamos a proceder a etiquetar el nodo minion1 y el nodo minion2 con las etiquetas "worker" como key, y "worker" como value

Ejecutamos en el nodo maestro el etiquetado del nodo **kubeminion1**



```
$ kubectl label node kubeminion1 node-role.kubernetes.io/worker=worker
```

node/kubeminion1 labeled

Ejecutamos en el nodo maestro el etiquetado del nodo **kubeminion2**

```
$ kubectl label node kubeminion2 node-role.kubernetes.io/worker=worker
```

node/kubeminion2 labeled

Finalmente, comprobamos de nuevo las etiquetas que ahora tienen los nodos

```
$ kubectl get nodes

NAME      STATUS ROLES AGE VERSION
kubemaster Ready  master  8d   v1.16.0
kubeminion1 Ready  worker  8d   v1.16.0
kubeminion2 Ready  worker  8d   v1.16.0
```

19.3. Describiendo el estado de un nodo

Podemos obtener información detallada del estado de un nodo mediante una operación de descripción, ejecutamos en el nodo maestro la descripción del nodo con nombre **kubemaster**

```
$ kubectl describe node kubemaster

Name:           kubemaster
Roles:          master
Labels:         beta.kubernetes.io/arch=amd64
                beta.kubernetes.io/os=linux
                kubernetes.io/arch=amd64
                kubernetes.io/hostname=kubemaster
                kubernetes.io/os=linux
                node-role.kubernetes.io/master=
Annotations:   kubeadm.alpha.kubernetes.io/cri-socket: /var/run/dockershim.sock
                node.alpha.kubernetes.io/ttl: 0
                volumes.kubernetes.io/controller-managed-attach-detach: true
CreationTimestamp: Mon, 23 Sep 2019 20:17:36 +0200
Taints:         node-role.kubernetes.io/master:NoSchedule
Unschedulable:  false
Conditions:
  Type        Status LastHeartbeatTime          LastTransitionTime
  Reason      Message
  -----
  NetworkUnavailable False  Mon, 23 Sep 2019 21:06:15 +0200  Mon, 23 Sep 2019 21:06:15 +0200
  WeaveIsUp       Weave pod has set this
  MemoryPressure  False  Tue, 24 Sep 2019 19:56:39 +0200  Mon, 23 Sep 2019 20:17:36 +0200
  KubeletHasSufficientMemory  kubelet has sufficient memory available
  DiskPressure    False  Tue, 24 Sep 2019 19:56:39 +0200  Mon, 23 Sep 2019 20:17:36 +0200
  KubeletHasNoDiskPressure  kubelet has no disk pressure
  PIDPressure    False  Tue, 24 Sep 2019 19:56:39 +0200  Mon, 23 Sep 2019 20:17:36 +0200
  KubeletHasSufficientPID   kubelet has sufficient PID available
  Ready          True   Tue, 24 Sep 2019 19:56:39 +0200  Mon, 23 Sep 2019 21:06:20 +0200
  KubeletReady     kubelet is posting ready status
Addresses:
  InternalIP: 192.168.15.100
  Hostname:   kubemaster
Capacity:
  cpu:        1
```

ephemeral-storage: 51175Mi
 hugepages-2Mi: 0
 memory: 5895864Ki
 pods: 110
 Allocatable:
 cpu: 1
 ephemeral-storage: 48294789041
 hugepages-2Mi: 0
 memory: 5793464Ki
 pods: 110

System Info:

Machine ID:	313cf0a5b6e049e3b348548b72a05b95
System UUID:	06D8C458-8273-4B05-AF2D-080F9E941BA1
Boot ID:	d53d82f3-4504-4dcf-bc7b-39a45efa9ecf
Kernel Version:	3.10.0-957.12.2.el7.x86_64
OS Image:	CentOS Linux 7 (Core)
Operating System:	linux
Architecture:	amd64
Container Runtime Version:	docker://18.6.3
Kubelet Version:	v1.16.0
Kube-Proxy Version:	v1.16.0

Non-terminated Pods: (9 in total)

Namespace	Name	CPU Requests	
CPU Limits	Memory Requests	Memory Limits	AGE
kube-system	coredns-5644d7b6d9-b8g2p	100m (10%)	0
(0%) 70Mi (1%)	170Mi (3%)	23h	
kube-system	coredns-5644d7b6d9-bmlbw	100m (10%)	0
(0%) 70Mi (1%)	170Mi (3%)	23h	
kube-system	etcd-kubemaster	0 (0%)	0
(0%) 0 (0%)	0 (0%)	23h	
kube-system	kube-apiserver-kubemaster	250m (25%)	0
(0%) 0 (0%)	0 (0%)	23h	
kube-system	kube-controller-manager-kubemaster	200m (20%)	0
(0%) 0 (0%)	0 (0%)	23h	
kube-system	kube-proxy-xlfrv	0 (0%)	0
(0%) 0 (0%)	0 (0%)	23h	
kube-system	kube-scheduler-kubemaster	100m (10%)	0
(0%) 0 (0%)	0 (0%)	23h	
kube-system	kubernetes-dashboard-7c54d59f66-txxwg	0 (0%)	0
(0%) 0 (0%)	0 (0%)	22h	
kube-system	weave-net-6wjw8	20m (2%)	0
(0%) 0 (0%)	0 (0%)	22h	

Allocated resources:

(Total limits may be over 100 percent, i.e., overcommitted.)

Resource	Requests	Limits
cpu	770m (77%)	0 (0%)
memory	140Mi (2%)	340Mi (6%)
ephemeral-storage	0 (0%)	0 (0%)

Describimos las secciones más relevantes:

Bloque General

Datos generales del nodo

- **Name**
 - Nombre del nodo
- **Roles**
 - Rol que tiene asignado el nodo en el clúster
- **Labels**
 - Etiquetas que tiene asignado el nodo
 - Típicamente observamos arquitectura hardware, sistema operativo, etc.
- **CreationTimeStamp**: Si tenemos dudas, aquí podemos consultar la fecha/hora exacta en la que el nodo se unió al clúster

Bloque Addresses

Configuración en cuanto a direccionamiento de red

- **HostName**: El nombre de host del nodo que ha informado el Kernel del S.O
 - Puede ser sobreescrito invocando kubelet con el parámetro --hostname-override
- **InternalIP**
 - La IP interna del clúster, sólo accesible a nivel interno
- **ExternalIP**
 - La IP externa del clúster, típicamente usada para que el propio clúster enrute tráfico hacia el exterior

Bloque Conditions

Describe el estado de los elementos que están operando en el nodo

Cada nodo posee una serie de conditions, cada una con un significado

- **OutOfDisk**
 - True: Indica que no hay espacio libre suficiente para añadir nuevos Pods
 - False: Hay espacio suficiente para una operatividad normal
- **MemoryPressure**
 - True: Significa que el todo está bajo presión excesiva en cuanto a la RAM, disponemos de poca memoria disponible
 - False: Hay memoria RAM suficiente para una operatividad normal

• **PIDPressure**

- True: Indica que el nodo está bajo presión excesiva en cuanto al número de procesos PID gestionados por el Kernel
- False: El número de procesos PID que está gestionando el nodo está en una operatividad normal

• **DiskPressure**

- True: Indica que el espacio de disco para el nodo, operando con kubernetes no es suficiente para una operativa normal
- False: El tamaño del disco es correcto para una operatividad normal

• **NetworkUnavailable**

- True: La red en el nodo está teniendo problemas, no está bien configurada para que el nodo pueda comunicarse con el resto de nodos
- False: La red del nodo está operando de manera normal

• **Ready**

- True: El nodo está saludable, preparado para recibir Pods
- False: El nodo no está en condiciones de poder recibir Pods
- Unknown: El Kube Controller Manager no tiene noticias del estado del nodo, por defecto el checking se realiza cada 40 segundos, se podría ajustar con el parámetro inline **node-monitor-grace-period**

Si transcurridos 5 minutos no hay conexión con el nodo, el kube controller manager da la orden de eliminación de los pods, indicará como status del Pod **Terminating**

 Hasta que no se restablecida la comunicación, la decisión que el kube controller manager ha determinado de eliminar todos los Pods no podrá ser ejecutada en el nodo en cuestión, llevando a cabo la eliminación de los Pods que allí hubiera desplegados una vez la comunicación quede restablecida

Aún habiendo conexión de red, esta situación puede darse, por que el Kube API Server no sea capaz de comunicarse con el agente Kubelet antes de que la

• **Capacity**

- Indica la capacidad total del nodo
- Puede que todo lo que está disponible no pueda ser usado por los Pods, por procesos internos del propio kernel del sistema, etc.
- Indica los recursos que el nodo dispone para su operatividad, como CPU, RAM y maximo número de pods que pueden ser controlados por el kube scheduler en el nodo

• **Allocatable**

- Esta sección nos indica los recursos asignables a los Pods respecto a la capidad de recursos del nodo

- **System Info**

- Describe información general sobre el nodo como la versión del kernel
- Versión del kubelet
- Versión del kube-proxy
- Versión del Docker Engine
- Nombre del sistema operativo

19.4. Obteniendo información del clúster

Kubernetes provee un comando para poder obtener información relativa al nodo maestro así como los espacios de nombres empleados

```
$ kubectl cluster-info
```

```
Kubernetes master is running at https://192.168.15.100:6443  
KubeDNS is running at https://192.168.15.100:6443/api/v1/namespaces/kube-  
system/services/kube-dns:dns/proxy
```

Por otra parte, podemos obtener una traza mucho más detallada sobre el estado del clúster para proceder a su posterior análisis, utilizando el redireccionador, volcamos directamente a fichero

```
$ kubectl cluster-info dump > cluster-status-log.txt
```

También podemos obtener otro resumen de configuración del clúster, mediante el siguiente comando:

```
$ kubectl config view

apiVersion: v1
clusters:
- cluster:
  certificate-authority-data: DATA+OMITTED
  server: https://192.168.15.100:6443
  name: kubernetes
contexts:
- context:
  cluster: kubernetes
  user: kubernetes-admin
  name: kubernetes-admin@kubernetes
current-context: kubernetes-admin@kubernetes
kind: Config
preferences: {}
users:
- name: kubernetes-admin
  user:
    client-certificate-data: REDACTED
    client-key-data: REDACTED
```



Capítulo 20. Kubectl taint

Con la operación taint de kubernetes, podemos "manchar" un nodo con un par key-value de forma que podamos ser selectivos a la hora de desplegar Pods

La idea es indicar al nodo, una etiqueta con un valor en concreto para poder establecer una restricción a los Pods de manera que el Kube Scheduler sólo envíe Pods que cumplan la regla

20.1. Efectos del taint en el nodo

- **NoSchedule**

- El sistema no desplegará un Pod en el nodo si no encuentra equivalencia del mismo con un taint
 - Se puede interpretar como "No ejecutar el Pod a menos que se cumpla la regla..."

- **PreferNoSchedule**

- El sistema intentará evitar colocar un Pod que no tenga coincidencia con la tolerancia, pero no es obligatorio (podría llegar a colocar algún Pod que no cumpliera la tolerancia)

- **NoExecute**

- Si el nodo tiene al menos un taint con efecto NoExecute, el Pod será expulsado del nodo (si en el nodo ya hubiera un Pod ejecutándose)
 - Se puede interpretar como "No ejecutar el Pod si encuentro la regla"

En un nodo podemos indicar tantos taints como queramos

Kubernetes procesa los taints como un filtro. Obtiene primero todos los taints del nodo. Luego descarta taints que no cumplen la regla del Pod

Si el nodo tiene un taint, la puesta en marcha de nuevos Pods estará siempre en función de los taints, esto quiere decir que si intentamos ejecutar de nuevo pods sin especificar reglas de tolerancia de taints, los Pods siempre se quedarán en estado **Pending**

En un clúster de kubernetes, los nodos maestros están manchados con una marca especial para no ejecutar Pods de usuario, únicamente los de sistema en el espacio de nombres kube-system

Ejecutamos el siguiente comando para obtener la descripción del nodo kubemaster:

```
$ kubectl describe node kubemaster  
...  
Taints:      node-role.kubernetes.io/master:NoSchedule  
...
```

Observamos la mancha en cuestión que le impide ejecutar Pods de usuario

20.2. Creando el taint en el nodo kubeminion1 (NoSchedule)

Los casos de uso del taint pueden ser de los más variados:

- Nuestro clúster está compuesto por una serie de nodos, y que cada nodo pertenece a un cliente, queremos tener desplegado en cada nodo únicamente ciertos Pods
- Disponemos de ciertos nodos con altas capacidades de GPU y deseamos que ciertos Pods únicamente se ejecuten en esos tipos de nodos
- Etc.
- Vamos a crearle un taint al nodo kubeminion1
- Indicamos como key **customerMachine**
- Indicamos como value **C1**
- Indicamos al Kube Scheduler el efecto, en este caso que no envíe Pods a no ser que cumplan la condición **NoSchedule**

```
$ kubectl taint nodes kubeminion1 customerMachine=C1:NoSchedule  
node/kubeminion1 tainted
```

20.3. Creando el taint en el nodo kubeminion2 (NoExecute)

- Vamos a crear un taint al nodo kubeminion2
- Indicamos como key **specialHardwareGPU**
- Indicamos como value **nvidia**
- Indicamos al Kube Scheduler el efecto, en este caso que no envíe Pods a no ser que cumplan la condición **NoExecute**

```
$ kubectl taint nodes kubeminion2 specialHardwareGPU=nvidia:NoExecute  
node/kubeminion2 tainted
```

- Efectos al detectar el nodo que tiene una key con el efecto **NoExecute**
 - Si hay algún Pod presente en ejecución, se analizará su **tolerationSeconds**, de forma que el Pod permanecerá en el nodo hasta que el tiempo establecido haya sido cumplido
 - Si los Pods presentes en el nodo, no tienen regla de tolerancia ni tienen especificado **tolerationSeconds** son inmediatamente eliminados

- Los nuevos Pods no se alojarán en dicho nodo
- El Nodo únicamente mostrará que dispone de los contenedores del namespace kube-system, concretamente los de Kube Proxy

20.4. Comprobando que el taint está presente en el nodo kubeminion1

Para comprobar que el nodo tiene el taint que acabamos de indicar, vamos a proceder a realizar una descripción del nodo y vamos a buscar la etiqueta **Taints**:

```
$ kubectl describe node kubeminion1

Name:           kubeminion1
Roles:          <none>
Labels:         beta.kubernetes.io/arch=amd64
                beta.kubernetes.io/os=linux
                kubernetes.io/arch=amd64
                kubernetes.io/hostname=kubeminion1
                kubernetes.io/os=linux
Annotations:   kubeadm.alpha.kubernetes.io/cri-socket: /var/run/dockershim.sock
                node.alpha.kubernetes.io/ttl: 0
                volumes.kubernetes.io/controller-managed-attach-detach: true
CreationTimestamp: Mon, 23 Sep 2019 21:12:06 +0200
Taints:         customerMachine=C1:NoSchedule
Unschedulable:  false
Conditions:
  Type        Status  LastHeartbeatTime          LastTransitionTime
  Reason      Message
  -----
  NetworkUnavailable False   Tue, 24 Sep 2019 17:03:37 +0200  Tue, 24 Sep 2019 17:03:37 +0200
  WeaveIsUp     Weave pod has set this
  MemoryPressure False   Wed, 25 Sep 2019 10:01:16 +0200  Tue, 24 Sep 2019 17:02:41 +0200
  KubeletHasSufficientMemory kubelet has sufficient memory available
  DiskPressure   False   Wed, 25 Sep 2019 10:01:16 +0200  Tue, 24 Sep 2019 17:02:41 +0200
  KubeletHasNoDiskPressure kubelet has no disk pressure
  PIDPressure   False   Wed, 25 Sep 2019 10:01:16 +0200  Tue, 24 Sep 2019 17:02:41 +0200
  KubeletHasSufficientPID kubelet has sufficient PID available
  Ready         True    Wed, 25 Sep 2019 10:01:16 +0200  Tue, 24 Sep 2019 17:02:41 +0200
  KubeletReady   Kubelet is posting ready status
Addresses:
  InternalIP:  192.168.15.101
  Hostname:    kubeminion1
Capacity:
  cpu:         1
  ephemeral-storage: 51175Mi
  hugepages-2Mi: 0
  memory:      1014968Ki
```

```

pods:           110
Allocatable:
cpu:            1
ephemeral-storage: 48294789041
hugepages-2Mi:  0
memory:         912568Ki
pods:           110
System Info:
Machine ID:      4e5ff38346fd448f83f67f78b4e8a99a
System UUID:    C4C915E5-B0A5-4B85-9073-8430C02CC585
Boot ID:          73350d75-f017-46c3-bd25-e678bb742883
Kernel Version:   3.10.0-957.12.2.el7.x86_64
OS Image:         CentOS Linux 7 (Core)
Operating System: linux
Architecture:     amd64
Container Runtime Version: docker://19.3.2
Kubelet Version:  v1.16.0
Kube-Proxy Version: v1.16.0
Non-terminated Pods: (2 in total)
  Namespace        Name        CPU Requests CPU Limits Memory
  Requests Memory Limits AGE
  -----
  kube-system      kube-proxy-j45h9  0 (0%)    0 (0%)    0 (0%)
  0 (0%) 36h
  kube-system      weave-net-z94kj  20m (2%)   0 (0%)    0 (0%)
  0 (0%) 36h
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
Resource      Requests Limits
  -----
  cpu            20m (2%)  0 (0%)
  memory         0 (0%)   0 (0%)
  ephemeral-storage 0 (0%)  0 (0%)
Events:
  Type  Reason        Age        From        Message
  ----  ---          --  --          --
  Normal NodeNotSchedulable 26m (x2 over 38m) kubelet, kubeminion1 Node
  kubeminion1 status is now: NodeNotSchedulable
  Normal NodeSchedulable   17m (x2 over 36m) kubelet, kubeminion1 Node
  kubeminion1 status is now: NodeSchedulable

```

20.5. Comprobando que el taint está presente en el nodo kubeminion2

Para comprobar que el nodo tiene el taint que acabamos de indicar, vamos a proceder a realizar una descripción del nodo y vamos a buscar la etiqueta **Taints**:

```
$ kubectl describe node kubeminion2
```

Name: kubeminion2
Roles: <none>
Labels:
 beta.kubernetes.io/arch=amd64
 beta.kubernetes.io/os=linux
 kubernetes.io/arch=amd64
 kubernetes.io/hostname=kubeminion2
 kubernetes.io/os=linux
Annotations:
 kubeadm.alpha.kubernetes.io/cri-socket: /var/run/dockershim.sock
 node.alpha.kubernetes.io/ttl: 0
 volumes.kubernetes.io/controller-managed-attach-detach: true
CreationTimestamp: Mon, 23 Sep 2019 21:12:28 +0200
Taints: specialHardwareGPU=nvidia:NoExecute
Unschedulable: false
Conditions:

Type	Status	LastHeartbeatTime	LastTransitionTime
Reason	Message		
-----	-----	-----	-----

 NetworkUnavailable False Tue, 24 Sep 2019 17:03:44 +0200 Tue, 24 Sep 2019 17:03:44 +0200 WeaveIsUp Weave pod has set this
 MemoryPressure False Wed, 25 Sep 2019 10:35:21 +0200 Tue, 24 Sep 2019 21:39:52 +0200 KubeletHasSufficientMemory kubelet has sufficient memory available
 DiskPressure False Wed, 25 Sep 2019 10:35:21 +0200 Tue, 24 Sep 2019 21:39:52 +0200 KubeletHasNoDiskPressure kubelet has no disk pressure
 PIDPressure False Wed, 25 Sep 2019 10:35:21 +0200 Tue, 24 Sep 2019 21:39:52 +0200 KubeletHasSufficientPID kubelet has sufficient PID available
 Ready True Wed, 25 Sep 2019 10:35:21 +0200 Tue, 24 Sep 2019 21:39:52 +0200 KubeletReady kubelet is posting ready status
Addresses:
 InternalIP: 192.168.15.102
 Hostname: kubeminion2
Capacity:
 cpu: 1
 ephemeral-storage: 51175Mi
 hugepages-2Mi: 0
 memory: 1014968Ki
 pods: 110
Allocatable:
 cpu: 1
 ephemeral-storage: 48294789041
 hugepages-2Mi: 0
 memory: 912568Ki
 pods: 110
System Info:
 Machine ID: 4e5ff38346fd448f83f67f78b4e8a99a
System UUID: 5F9A639C-7F9C-4932-91C0-184639F6DC36
 Boot ID: b7f16368-c484-4c5c-bbef-59ad3c8716ed
 Kernel Version: 3.10.0-957.12.2.el7.x86_64
 OS Image: CentOS Linux 7 (Core)
 Operating System: linux

```

Architecture:      amd64
Container Runtime Version: docker://19.3.2
Kubelet Version:    v1.16.0
Kube-Proxy Version: v1.16.0
Non-terminated Pods: (1 in total)
  Namespace       Name            CPU Requests  CPU Limits  Memory
  Requests   Memory Limits AGE
  -----
  kube-system   kube-proxy-24k62  0 (0%)     0 (0%)     0 (0%)
  0 (0%)      37h

Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  Resource      Requests  Limits
  -----
  cpu          0 (0%)  0 (0%)
  memory       0 (0%)  0 (0%)
  ephemeral-storage 0 (0%)  0 (0%)
Events:        <none>

```



Fondos Europeos



20.6. Operadores del taint en el Pod

En cuanto a la tolerancia, podemos especificar 2 tipos de operadores:

- **Exists**
 - Con que exista la key en el nodo es suficiente, no se comprueba el valor que tenga
- **Equal**
 - El nodo tiene que tener la key especificada y el valor especificado
- Caso específico, únicamente se especifica el operador, sin key
 - Con que se encuentre un taint en cualquier nodo, y con cualquier valor asociado vale para desplegar el Pod ahí

```

tolerations:
  - operator: "Exists"

```

- Caso específico, se especifica la key sin value
 - Con que encuentre la key es suficiente para iniciar el despliegue

```

tolerations:
  -key: "key"
  operator: "Exists"

```



Cofinanciado por
la Unión Europea



20.7. Creando un Pod con regla del taint (NoSchedule)

A continuación, vamos a crear un Pod que tenga una regla de taint, para que el Kube Scheduler pueda localizar el nodo en cuestión y proceder al despliegue en el nodo que cumpla la regla

Creamos el archivo **pod-with-taint-no-schedule.yml**

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-taint-no-schedule
spec:
  tolerations:
    - key: "customerMachine"
      operator: "Equal"
      value: "C1"
      effect: "NoSchedule"
  containers:
    - name: nginx
      image: nginx:latest
      ports:
        - containerPort: 80
```



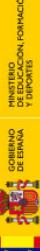
Ejecutamos la orden de creación

```
$ kubectl apply -f pod-with-taint-no-schedule.yml
```

```
pod/pod-with-taint-no-schedule created
```

Creamos el archivo **pod-with-taint-no-execute.yml**

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-taint-no-execute
spec:
  tolerations:
    - key: "specialHardwareGPU"
      operator: "Equal"
      value: "nvidia"
      effect: "NoExecute"
      tolerationSeconds: 60
  containers:
    - name: nginx
      image: nginx:latest
      ports:
        - containerPort: 80
```



- Indicamos una tolerancia de 60 segundos antes de que Kubernetes se cargue el Pod, en caso de que en el nodo aparezca un taint de efecto **NoExecute**

Ejecutamos la orden de creación

```
$ kubectl apply -f pod-with-no-execute.yml
pod/pod-with-no-execute created
```

Creamos otro archivo **pod-with-no-taint.yml**

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-no-taint
spec:
  containers:
    - name: nginx
      image: nginx:latest
      ports:
        - containerPort: 80
```

Ejecutamos la orden de creación

```
$ kubectl apply -f pod-with-no-taint.yml
pod/pod-with-no-taint created
```

20.8. Eliminar el taint de un nodo

Eliminamos del nodo kubeminion2 el taint **specialHardwareGPU=nvidia:NoExecute**

```
$ kubectl taint nodes kubeminion2 specialHardwareGPU:NoExecute-
node/kubeminion2 untainted
```

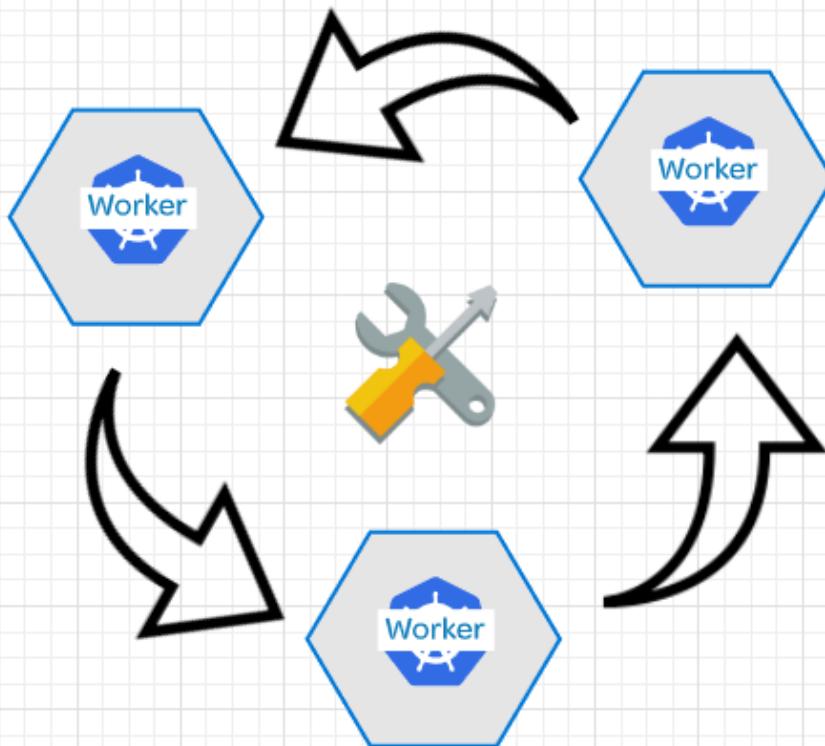
Eliminamos del nodo kubeminion1 el taint **customerMachine:NoSchedule-**

```
$ kubectl taint nodes kubeminion1 customerMachine:NoSchedule-
```

Capítulo 21. Mantenimiento de los nodos

Node Maintenance

Due to the pod eviction timeout set by the controller manager, pods are terminated after five minutes by default, unless you are using ReplicaSets.



El transcurso de ciclo de vida de un clúster de kubernetes, a lo largo de tiempo, van a surgirnos necesidades como...

- Actualizar paquetes del sistema operativo de un nodo
- Actualizar la versión de los componentes de kubernetes
- Quitar una fuente de alimentación
- Aumentar la memoria RAM física del servidor
- Informar a kubernetes que no envíe más trabajo al nodo (por que hemos detectado algo raro en esa máquina, apagones, cortes de red, etc.)
- Sacar todos los Pods que estuvieran computando en el servidor en cuestión, y localizar otro para restituirlos
- Etc.

Kubernetes nos va a proveer de una serie de operaciones, que debemos de conocer y nosotros

aplicar en el orden de secuencia correcto que vayamos a necesitar

21.1. Operación de Acordonado (cordon)



La operación de acordonado en kubernetes, tenemos que imaginárnosla como una especie de escudo que el clúster le otorga a un nodo, de forma que el **kube-scheduler** no podrá enviarle nunca en este estado, más trabajo del que adicionalmente ya estuviera en ejecución.

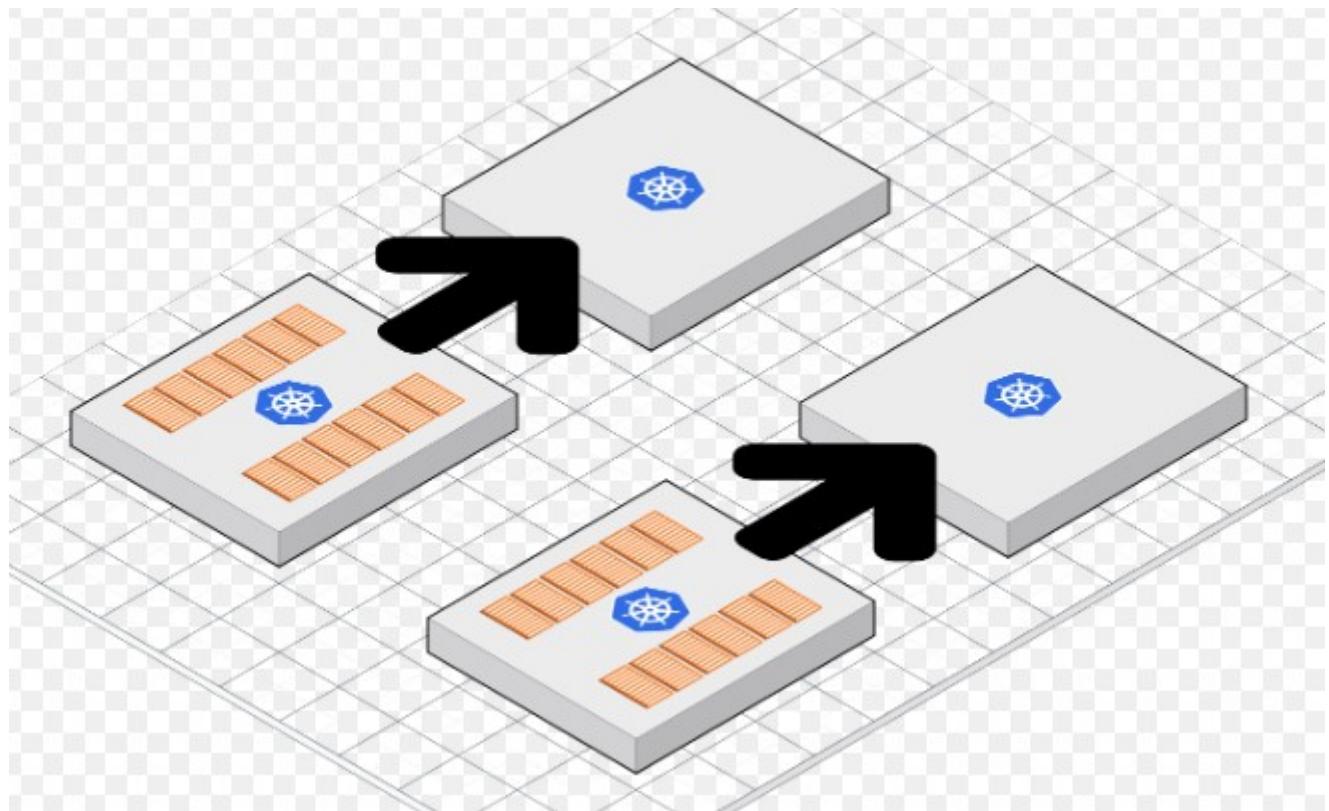
El trabajo que se quedara dentro, antes de poner "el escudo", sigue su operación normal sin ningún tipo de alteración

21.2. Operación de Desacordonado (uncordon)



La operación de descordonado, va a permitir, el llenado de nuevo del nodo con nuevos pods que el planificador **kube-scheduler** tenga pendientes de asignación y puedan alojarse en el mismo

21.3. Operación de Dreando (Drain)



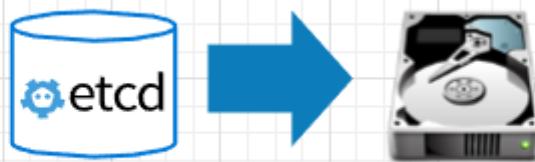
La operación de drenaje, nos permite, una vez que un nodo se encuentra acordonado, llevar a cabo un vaciado del mismo, en el sentido, de desalojar todos los pods que hubieran computando y

localizar una nueva máquina del cluster donde poder realojar de nuevo esos pods

21.4. Operación Backup & Restauración

Back Up etcd

etcd is where all cluster updates exist. If you're not replicating etcd, it's a good idea to take a periodic snapshot.



Será conveniente llevar a cabo operaciones de copia de seguridad del componente etcd, para poder realizar la restauración futura del clúster antes situaciones de emergencia.

Este componente etcd es crítico que mantenga una buena política de copias de seguridad, especialmente si disponemos únicamente de un nodo máster

Capítulo 22. Lab: Mantenimiento de los nodos

22.1. Operación acordonamiento del nodo

En determinadas ocasiones, nos interesa indicar a un nodo, que el kube scheduler no le suministre más tareas a realizar, procesando este únicamente las tareas que tenga asignadas hasta el momento

Realizamos la operación con el nodo 1

```
$ kubectl cordon kubeminion1  
node/kubeminion1 cordoned
```

Listamos a continuación los nodos disponibles y vamos a observar como el nodo indica que no está disponible a efectos del kube scheduler

```
$ kubectl get nodes  
NAME      STATUS      ROLES      AGE      VERSION  
kubemaster Ready      master    37h      v1.16.0  
kubeminion1 Ready,SchedulingDisabled <none> 36h      v1.16.0  
kubeminion2 Ready      <none>    36h      v1.16.0
```

A continuación, volvemos a indicar que el nodo si está disponible para recibir nuevas tareas por parte del kube scheduler

```
$ kubectl uncordon kubeminion1  
node/kubeminion1 uncordoned
```

Comprobamos nuevamente que el nodo ha cambiado su estado

```
$ kubectl get nodes  
NAME      STATUS      ROLES      AGE      VERSION  
kubemaster Ready      master    37h      v1.16.0  
kubeminion1 Ready      <none>    36h      v1.16.0  
kubeminion2 Ready      <none>    36h      v1.16.0
```

22.2. Operación drenado del nodo

Puede ocurrir que tengamos que realizar un mantenimiento hardware del servidor físico, sustituir

una fuente de alimentación dañada, memoria RAM, etc.

No debemos de desconectar "a lo rambo" el servidor, sino, que previamente debemos de informar al clúster que el nodo va a ser desconectado, para que el orquestador gestione de forma eficaz la creación de contenedores de los sistemas que están operando en otros nodos

Una vez la operación de drenado haya finalizado, podremos desconectar el nodo del clúster sin mayor problema

Vamos a realizar la operación de drenado en el nodo kubeminion1, especificando un periodo de gracia de 1 minuto

```
$ kubectl drain kubeminion1 --grace-period=60
node/kubeminion1 cordoned
error: unable to drain node "kubeminion1", aborting command...
There are pending nodes to be drained:
kubeminion1
cannot delete Pods not managed by ReplicationController, ReplicaSet, Job, DaemonSet or
StatefulSet (use --force to override): default/pod-with-labels
cannot delete DaemonSet-managed Pods (use --ignore-daemonsets to ignore): kube-system
/kube-proxy-j45h9, kube-system/weave-net-z94kj
```

Si en el nodo tenemos elementos que no están controlados por un ReplicationController, ReplicaSet, Job, DaemonSet, etc. No vamos a poder hacer la operación, quedando el nodo en un estado como si le hubiéramos realizado un cordon



Por otra parte, si forzamos la operación, los Pods que desaparezcan en un nodo, no volverán a crearse en otros nodos, como hemos comentado, no disponemos de ninguna unidad de control de replicación que posibilite esto

Para poder llevar a cabo la operación, debemos de forzar la acción con el argumento `--force`

Ejecutamos la operación de drenado aún no habiendo ningún controlador de replicación presente (Pods sueltos)

```
$ kubectl drain kubeminion1 --grace-period=60 --force --ignore-daemonsets
node/kubeminion1 already cordoned
WARNING: deleting Pods not managed by ReplicationController, ReplicaSet, Job,
DaemonSet or StatefulSet: default/pod-with-labels; ignoring DaemonSet-managed Pods:
kube-system/kube-proxy-j45h9, kube-system/weave-net-z94kj
evicting pod "pod-with-labels"
pod/pod-with-labels evicted
node/kubeminion1 evicted
```

Por último, podemos volver a reactivar el nodo para que vuelva a poder recibir peticiones del kube-scheduler

```
$ kubectl uncordon kubeminion1
```

Comprobamos nuevamente que el nodo ha cambiado su estado

```
$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
kubemaster	Ready	master	37h	v1.16.0
kubeminion1	Ready	<none>	36h	v1.16.0
kubeminion2	Ready	<none>	36h	v1.16.0



22.3. Eliminación de nodo + generación de nuevo token de acceso

Otro caso que nos puede suceder, es que queramos eliminar una máquina del clúster de forma definitiva por alguna razón, e incorporar una nueva máquina al mismo, generando un nuevo token de acceso

Vamos a proceder a eliminar de forma correcta el nodo kubeminion2 y posteriormente lo volveremos a incorporar al clúster

Primero, lanzamos una operación de acordonado al nodo, para que el kube-scheduler no le envíe nuevos trabajos a realizar:

```
$ kubectl cordon kubeminion2  
node/kubeminion2 cordoned
```



Comprobamos el estado de los nodos:

```
$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
kubemaster	Ready	master	20d	v1.17.0
kubeminion1	Ready	minion	20d	v1.17.0
kubeminion2	Ready,SchedulingDisabled	minion	20d	v1.17.0



Ahora, ejecutamos el drenaje, para literalmente secar el nodo, y desalojar todos los pods que hubiera computando dentro del mismo:

```
$ kubectl drain kubeminion2
evicting pod "deployment-for-services-6fdb84645-vd6mt"
pod/deployment-for-services-6fdb84645-vd6mt evicted
node/kubeminion2 evicted
```

La unidad replicadora se ha encargado de recrear nuevos pods en los nodos del clúster que si están operativos, en nuestro caso, el nodo que nos queda disponible es el kubeminion1

Comprobamos que los pods están sanos y están computando en el nodo kubeminion1:

```
$ kubectl get pods -o wide
NAME                  READY STATUS RESTARTS AGE   IP
NODE      NOMINATED NODE READINESS GATES
deployment-for-services-6fdb84645-5n79n  2/2   Running  0     2m52s
10.46.0.5  kubeminion1 <none>    <none>
deployment-for-services-6fdb84645-p55gj  2/2   Running  0     47h
10.46.0.3  kubeminion1 <none>    <none>
```

Ahora, podemos proceder a eliminar el nodo kubeminion2 del clúster de forma segura, sin que se produzcan desequilibrios temporales en la computación:

```
$ kubectl delete node kubeminion2
node "kubeminion2" deleted
```

Listamos de nuevo los nodos, y observamos que sólo queda el master y un minion:

```
$ kubectl get nodes
NAME      STATUS ROLES AGE VERSION
kubemaster Ready master 20d v1.17.0
kubeminion1 Ready minion 20d v1.17.0
```

En el nodo máster, generamos un nuevo token de forma que nos aparezca por la consola:

```
# kubeadm token generate
13ztmw.ptrr846sfa0jj8zg
```

A continuación, una vez generado el token, vamos a decirle a kubernetes que nos cree una llave de acceso al clúster, a partir del hash del token generado.

Paralelamente, vamos a indicar que el tiempo de vida del token va a ser de 2 horas, si superado ese

tiempo queremos utilizar la instrucción de inclusión de una máquina en el clúster, el token no será válido y no podremos llevar a cabo la operación:

```
# kubeadm token create 13ztmw.ptrr846sfa0jj8zg --ttl 2h --print-join-command  
kubeadm join 192.168.15.100:6443 --token 13ztmw.ptrr846sfa0jj8zg --discovery-token  
-ca-cert-hash sha256:af26bf7f57ba8dfa0e91761f9c42a15ed24d08b4d9969bd1d266b54b9d03177
```

Nos conectamos por ssh al nodo kubeminion2, este nodo, previamente ya estaba dentro del clúster, debemos de borrar manualmente los siguientes archivos:

- /etc/kubernetes/kubelet.conf
- /etc/kubernetes/pki/ca.crt

```
# rm /etc/kubernetes/kubelet.conf  
# rm /etc/kubernetes/pki/ca.crt
```

También, necesitamos lanzar sobre el nodo una operación de reseteo en lo que respecta al clúster:

```
# kubeadm reset
```

Ahora, ejecutamos en el servidor kubeminion 2 el siguiente comando:

```
# kubeadm join 192.168.15.100:6443 --token 13ztmw.ptrr846sfa0jj8zg --discovery-token-ca-cert-hash sha256:af26bf7f57ba8dfa0e91761f9c42a15ed24d08b4d9969bd1d266b54b9d03177

kubeadm join 192.168.15.100:6443 --token 13ztmw.ptrr846sfa0jj8zg --discovery-token-ca-cert-hash sha256:af26bf7f57ba8dfa0e91761f9c42a15ed24d08b4d9969bd1d266b54b9d03177
W0105 20:32:56.227263 7022 join.go:346] [preflight] WARNING: JoinControlPlane .controlPlane settings will be ignored when control-plane flag is not set.
[preflight] Running pre-flight checks
  [WARNING IsDockerSystemdCheck]: detected "cgroupfs" as the Docker cgroup driver.
The recommended driver is "systemd". Please follow the guide at https://kubernetes.io/docs/setup/cri/
[preflight] Reading configuration from the cluster...
[preflight] FYI: You can look at this config file with 'kubectl -n kube-system get cm kubeadm-config -oyaml'
[kubelet-start] Downloading configuration for the kubelet from the "kubelet-config-1.17" ConfigMap in the kube-system namespace
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"
[kubelet-start] Writing kubelet environment file with flags to file "/var/lib/kubelet/kubeadm-flags.env"
[kubelet-start] Starting the kubelet
[kubelet-start] Waiting for the kubelet to perform the TLS Bootstrap...
```

This node has joined the cluster:

- * **Certificate** signing request was sent to apiserver and a response was received.
- * The Kubelet was informed of the **new** secure connection details.

Run '**kubectl get nodes**' on the control-plane to see this node join the cluster.

Finalmente, en el nodo máster, ejecutamos un listado de nodos para ver si de nuevo lo tenemos en el clúster:

```
$ kubectl get nodes

NAME      STATUS  ROLES   AGE  VERSION
kubemaster  Ready  master  20d  v1.17.0
kubeminion1 Ready  minion  20d  v1.17.0
kubeminion2 Ready  <none>  82s  v1.17.0
```

22.4. Backup del componente etcd

Para llevar a cabo la gestión de la copia de seguridad del componente etcd, vamos a descargar una herramienta de línea de comandos a modo de cliente, para tratar con el etcd:

```
$ wget https://github.com/etcd-io/etcd/releases/download/v3.3.12/etcd-v3.3.12-linux-amd64.tar.gz
```

Descomprimimos el archivo:

```
$ tar xvf etcd-v3.3.12-linux-amd64.tar.gz
```

Ahora, movemos los archivos binarios **etcd** y **etcdctl** a la carpeta **/usr/bin**:

```
$ sudo mv etcd-v3.3.12-linux-amd64/etcd* /usr/bin
```

Comprobamos la versión que tenemos de la herramienta cliente:

```
$ etcdctl --version
```

etcdctl version: 3.3.12

API version: 2

Podemos indicarle al binario, que nos muestre la ayuda de la herramienta en formato de versión 3 si escribimos el siguiente comando por consola:

```
$ ETCDCTL_API=3 etcdctl --help
```

Ahora, vamos a hacer un snapshot del almacén etcd, indicando:

- **snapshot.db**
 - El nombre del archivo de copia de seguridad que se generará
- **--cacert /etc/kubernetes/pki/etcd/server.crt**
 - El certificado de seguridad que vamos a utilizar (el del propio etcd)
- **--cert /etc/kubernetes/pki/etcd/ca.crt**
 - La entidad certificadora
- **--key /etc/kubernetes/pki/etcd/ca.key**
 - La clave privada para poder utilizar el certificado

```
$ sudo ETCDCTL_API=3 etcdctl snapshot save snapshot.db --cacert /etc/kubernetes/pki/etcd/server.crt --cert /etc/kubernetes/pki/etcd/ca.crt --key /etc/kubernetes/pki/etcd/ca.key
```

Snapshot saved at snapshot.db

Una vez realizado el backup, podemos comprobar el estado del mismo, para verificar cuánto ocupa, ejecutaremos el siguiente comando por consola:

```
$ ETCDCTL_API=3 etcdctl --write-out=table snapshot status snapshot.db
```

HASH	REVISION	TOTAL KEYS	TOTAL SIZE
dc46e7fa	313816	1224	8.0 MB

22.5. Restaurando backup del componente etcd

Una vez disponemos de un snapshot de la base de datos del etcd, podemos llevar a cabo su restauración en caso de necesidad.

La operación de restauración, sobreescribe el dato del member-id y cluster-id del etcd

Realmente la operación de restauración, va a conllevar la creación (a nivel del etcd) de un cluster nuevo, reemplazando a posteriori ciertos datos con lo que hay guardado en el archivo snapshot.db que previamente hemos creado

A bajo nivel, también tendremos que tener en cuenta, que ante una operación de desastre del servidor donde se almacena el etcd, deberemos de asignarle la misma IP que originalmente tenía, ya que este dato también está anotado en el almacén del etcd para propósito de comunicaciones

```
$ ETCDCTL_API=3 etcdctl snapshot restore snapshot.db --name m1 --initial-cluster m1
=https://192.168.15.100:2380,m2=https://192.168.15.101:2380,m3=https://192.168.15.102:
2380 --initial-cluster-token etcd-cluster --initial-advertise-peer-urls
https://192.168.15.100:2380
2020-01-05 22:21:40.337354 I | mvcc: restore compact to 312882
2020-01-05 22:21:40.369016 I | etcdserver/membership: added member 1720f3b2dd5f5ed4
[https://192.168.15.100:2380] to cluster 53dc01912b6a3c83
2020-01-05 22:21:40.369087 I | etcdserver/membership: added member c61fff6ce4711426
[https://192.168.15.102:2380] to cluster 53dc01912b6a3c83
2020-01-05 22:21:40.369124 I | etcdserver/membership: added member ed34bbc155f3dcff
[https://192.168.15.101:2380] to cluster 53dc01912b6a3c83
```

Capítulo 23. Actualización de clúster de Kubernetes

Otra de las tareas típicas que nos tocará realizar, es llevar a cabo una actualización de kubernetes en nuestros servidores, para con el tiempo, ir implementando las mejoras y correcciones de errores que se vayan produciendo

23.1. Comprobando la versión del agente kubelet

Para comprobar las versiones que tenemos del agente kubelet en cada uno de los nodos, ejecutamos el siguiente comando:

```
$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
kubemaster	Ready	master	20d	v1.17.0
kubeminion1	Ready	minion	20d	v1.17.0
kubeminion2	Ready	minion	20d	v1.17.0

Observamos que tenemos la versión v1.17.0 en cada uno de los nodos

23.2. Comprobando la versión de kubectl + kube-api-server

También, podemos comprobar la versión que tenemos tanto del cliente kubectl, como del servidor kube-api-server, ejecutando el siguiente comando:

```
$ kubectl version --short
```

Client Version: v1.17.0
Server Version: v1.17.0

- **Client Version**
 - Versión del intérprete de cliente kubectl
- **Server Version**
 - Versión del kube-api-server que tenemos presente en el control-plane

23.3. Comprobando la versión del kube-proxy

Por otra parte, también podemos llevar a cabo la comprobación de la versión que tenemos en cada nodo, en lo respecta al componente kube-proxy, ejecutando el siguiente comando:

```
$ kubectl describe nodes

...
Kubelet Version:      v1.17.0
Kube-Proxy Version:   v1.17.0
...
```

Observaremos en los nodos, entre otra información, las siguientes entradas:

- **Kubelet Version**
 - Versión del agente kubelet en el nodo
- **Kube-Proxy Version**
 - Versión del componente kube-proxy presente en el nodo



23.4. Comprobando la versión del kube-controller-manager

Para comprobar la versión de este componente, tenemos que recordar que está presente en el namespace específico kube-system

Primero consultamos los Pods que están en operación, en el namespace kube-system:

```
$ kubectl get pods --namespace kube-system

coredns-6955765f44-bzz62      1/1  Running  0      46h
coredns-6955765f44-klj6b      1/1  Running  2      20d
etcd-kubemaster                1/1  Running  2      20d
kube-apiserver-kubemaster     1/1  Running  3      20d
kube-controller-manager-kubemaster 1/1  Running  9      20d
kube-proxy-bhf4v               1/1  Running  2      20d
kube-proxy-q24bh               1/1  Running  4      20d
kube-proxy-w2jsj               1/1  Running  2      20d
kube-scheduler-kubemaster     1/1  Running  9      20d
weave-net-48fbp                2/2  Running  6      20d
weave-net-52qlp                2/2  Running  6      20d
weave-net-gm5sg                2/2  Running  9      20d
```

Observamos el pod que nos interesa, **kube-controller-manager-kubemaster**, a continuación ejecutamos la siguiente instrucción, que obtendrá la representación en formato YAML, mediante la cual podremos observar una entrada del tipo image... Que nos dirá la versión concreta:

```
$ kubectl get pods kube-controller-manager-kubemaster -o yaml --namespace kube-system
```

```
...
image: k8s.gcr.io/kube-controller-manager:v1.17.0
...
```

23.5. Actualizando paquetes del sistema, sin tocar kubernetes

En un sistema CentOS, para actualizar paquetes el sistema sin alterar un paquete en concreto, disponemos del modificador en la línea de comandos de `--exclude=<package>`

También debemos de recordar, que tenemos que llevar a la par, la versión del motor de contenedores Docker, acompañada con la versión del orquestador de Kubernetes

En nuestro caso, podemos actualizar el sistema indicando el siguiente comando:

```
# yum --exclude=docker* --exclude=kubernetes* update
```

23.6. Actualizando kubernetes (Nodos maestros)

En primer lugar, vamos a comprobar las versiones que tenemos disponibles en nuestro repositorio local del paquete **kubeadm**:

```
$ yum list kubeadm --showduplicates | sort -r

* updates: mirror.airenetworks.es
Loaded plugins: fastestmirror, langpacks
kubeadm.x86_64           1.17.0-0          @kubernetes
Installed Packages
 * extras: mirror.airenetworks.es
 * epel: ftp.uma.es
Determining fastest mirrors
 * base: ftp.csuc.cat
```

Observamos que tenemos una, la versión 1.17.0-0

23.7. Instalando una versión específica de kubeadm

A continuación, vamos a indicar que queremos instalar una versión de paquete específica del componente **kubeadm**:

```
># yum install kubeadm-1.17.0-0
```

En caso de que la versión que tengamos en nuestros servidores, sea más antigua que esa, se procederá a llevar a cabo la actualización

El siguiente paso, sería comprobar las versiones disponibles para actualizar y validar el clúster:

```
># kubeadm upgrade plan

[upgrade/config] Making sure the configuration is correct:
[upgrade/config] Reading configuration from the cluster...
[upgrade/config] FYI: You can look at this config file with 'kubectl -n kube-system get cm kubeadm-config -oyaml'
[preflight] Running pre-flight checks.
[upgrade] Making sure the cluster is healthy:
[upgrade] Fetching available versions to upgrade to
[upgrade/versions] Cluster version: v1.17.0
[upgrade/versions] kubeadm version: v1.17.0
[upgrade/versions] Latest stable version: v1.17.0
[upgrade/versions] Latest version in the v1.17 series: v1.17.0

Awesome, you're up-to-date! Enjoy!
```

En nuestro caso, estamos actualizados en lo que respecta a los componentes del clúster, estamos en la última versión

En caso contrario, el sistema nos informaría con un resultado parecido a este (caso de que tuviéramos la versión 1.13.8 y quisiéramos actualizar a la 1.14.1):



Fondos Europeos



MINISTERIO
GOBIERNO DE ESPAÑA
DE EDUCACIÓN, FORMACIÓN PROFESIONAL
Y DEPORTE

Cofinanciado por
la Unión Europea



```
># kubeadm upgrade plan

[upgrade/config] Making sure the configuration is correct:
[upgrade/config] Reading configuration from the cluster...
[upgrade/config] FYI: You can look at this config file with 'kubectl -n kube-system
get cm kubeadm-config -oyaml'
[upgrade/config] FYI: You can look at this config file with 'kubectl --namespace kube-
system get cm kubeadm-config -oyaml'
[upgrade] Fetching available versions to upgrade to
[upgrade/versions] Cluster version: v1.13.8
[upgrade/versions] kubeadm version: v1.14.1
[upgrade/versions] Latest stable version: v1.14.4
[upgrade/versions] Latest version in the v1.13 series: v1.13.8
```

Components that must be upgraded manually after you have upgraded the control plane with '`kubeadm upgrade apply`':

COMPONENT	CURRENT	AVAILABLE
Kubelet	3 x v1.13.5	v1.14.4

Upgrade to the latest stable version:

COMPONENT	CURRENT	AVAILABLE
API Server	v1.13.8	v1.14.4
Controller Manager	v1.13.8	v1.14.4
Scheduler	v1.13.8	v1.14.4
Kube Proxy	v1.13.8	v1.14.4
CoreDNS	1.2.6	1.3.1
Etcd	3.2.24	3.3.10

You can now apply the upgrade by executing the following command:

```
kubeadm upgrade apply v1.14.4
```

Note: Before you can perform this upgrade, you have to update kubeadm to v1.14.4

23.8. Actualizando mediante kubeadm

Mediante este proceso, kubernetes descargará las imágenes de los componentes del clúster a la versión que le hemos indicado, y los sustituirá por los nuevos de forma automática

Ejecutamos la instrucción que hemos observado en el paso anterior:

```
># kubeadm upgrade apply v1.14.4
```

Tras lanzar el comando anterior, comprobamos las versiones que tenemos operando, tanto del cliente kubectl, como del componente kube-api-server

```
$ kubectl version --short
```

Client Version: v1.13.5

Server Version: v1.14.1

Observamos que la versión del componente kube-api-server si que se ha actualizado correctamente, pero faltaría la parte de cliente, que como observamos sigue en la versión antigua

23.9. Actualizando la versión de kubectl

Ahora, toca actualizar el paquete de kubectl para que esté nivelado con la versión que disponemos del componente kube-api-server:

```
># yum install kubectl-1.14.1-0
```

Finalmente, comprobamos la actualización, y observaríamos lo siguiente:

```
$ kubectl version --short
```

Client Version: v1.14.1

Server Version: v1.14.1

23.10. Actualizando el agente kubelet

Otro punto importante, es llevar a cabo la actualización del agente kubelet, procedemos de la siguiente forma:

```
># yum install kubelet-1.14.1-0
```

23.11. Comprobando la versión de kubernetes en los nodos

Una vez realizado el proceso de actualización en el control plane, procedemos a comprobar si el listado de nodos, nos muestra la versión correcta:

```
$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
kubemaster	Ready	master	20d	v1.14.1
kubeminion1	Ready	minion	20d	v1.13.5
kubeminion2	Ready	minion	20d	v1.13.5

23.12. Actualizando kubernetes (Nodos Minions)

Ejecutamos en cada nodo minion lo siguiente:

Primero, actualizamos el paquete kubeadm:

```
># yum install kubeadm-1.17.0-0
```

A continuación, actualizamos el intérprete kubectl:

```
># yum install kubectl-1.14.1-0
```

Y por último, actualizamos el agente kubelet:

```
># yum install kubelet-1.14.1.0
```



En los nodos minions, a diferencia, de los nodos maestros, que conforman el control-plane, no es necesario lanzar la instrucción kubeadm upgrade apply...

Por último, comprobamos la versión de los nodos, que efectivamente se ha nivelado de forma correcta:

```
$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
kubemaster	Ready	master	20d	v1.14.1
kubeminion1	Ready	minion	20d	v1.14.1
kubeminion2	Ready	minion	20d	v1.14.1

Capítulo 24. Pods

- En Docker, la unidad atómica es un contenedor, en Kubernetes es un Pod
- Un Pod puede contener uno o más contenedores.
- Utilizado para definir todos los contenedores que sean altamente acoplados.
 - Por ejemplo, que te interese desplegar de forma conjunta pares de "frontends" y "backends"

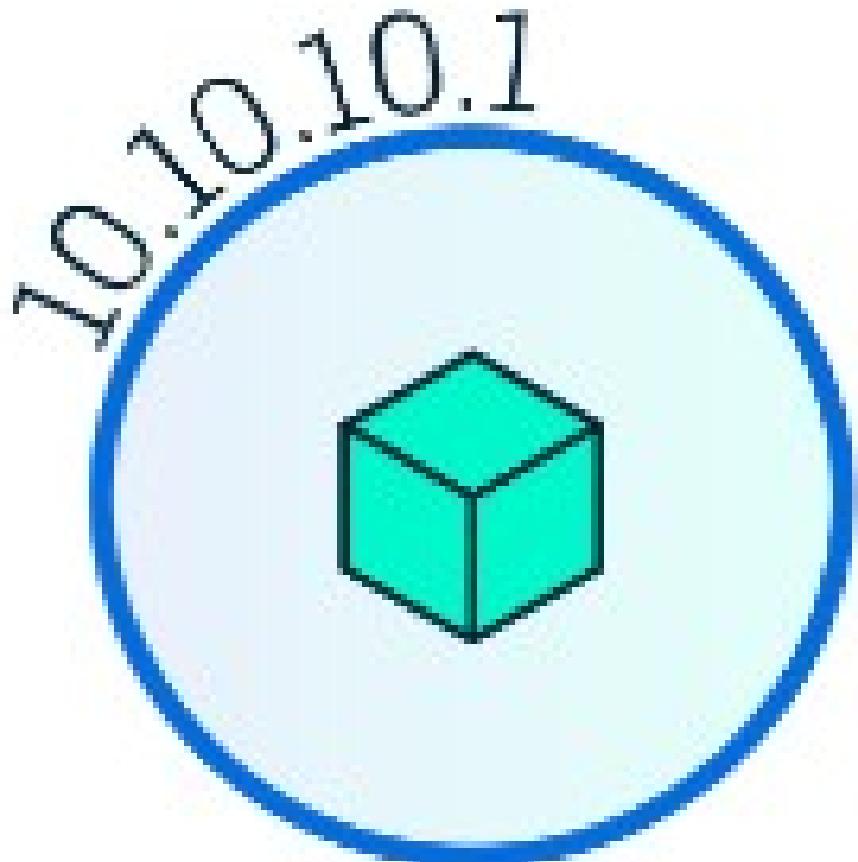


Figure 9. 1 Pod con 1 container

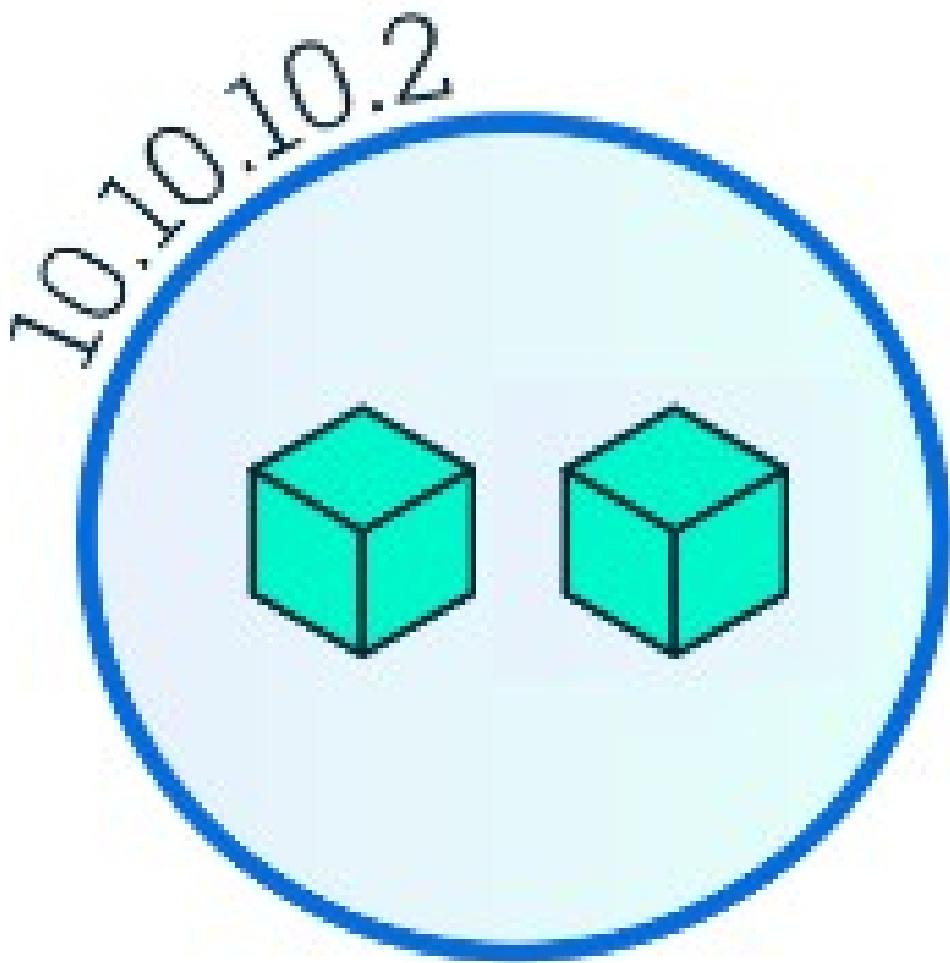


Figure 10. 1 Pod con 2 containers

24.1. Implicaciones de múltiples contenedores en un mismo Pod

- Todos los contenedores alojados en el mismo Pod, comparten la misma interfaz de red
 - Si dos contenedores dentro del pod, ocupan el mismo puerto a nivel de despliegue... Se provocará un error al intentar desplegarlo (Ej.: Levantar 2 contenedores de nginx en el mismo Pod)
- En kubernetes, se considera que los contenedores de un pod comparten un mismo namespace de red (netns)
- A nivel de Pod, todos los contenedores comparten los mismos recursos (cgroups, namespaces, volúmenes, etc.)



Cuando desplegamos un pod con más de un container, tenemos que tener la precaución de que operen en puertos distintos, a nivel interno del container, los pods comparten la red, lo que significa que si intentamos levantar en este caso, en lugar de un nginx y un servicer tomcat, dos servidores nginx en el mismo puerto... Habría colisión y un fallo en la secuencia de arranque del pod

24.2. Comunicaciones

- La comunicación entre pods es muy sencilla.
- Cada pod puede hablar con cualquier pod en la red
- Por defecto, dentro de un pod, las comunicaciones entre contenedores se realizan en localhost.
- Las comunicaciones entre pods se realizan por medio de la interfaz de red externa del pod exponiendo los puertos de los contenedores.

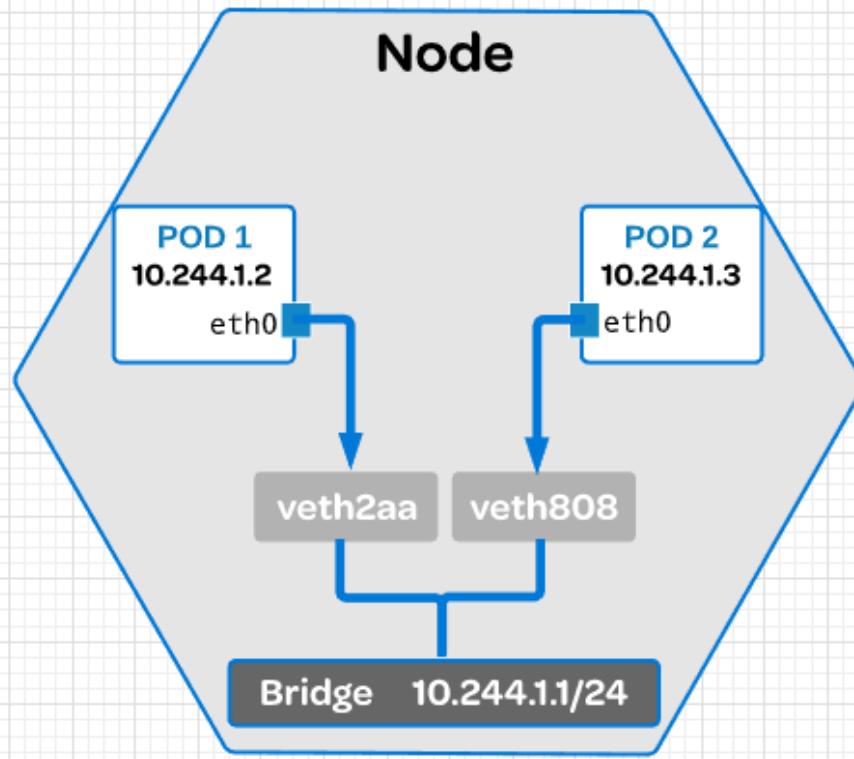
24.3. Comunicación interpods (Mismo Nodo)

Cuando en un nodo se ponen en marcha diferentes Pods, en cuanto a la interfaz de red, por cada pod se crea un puente de red, con el prefijo **vethXXX**

Eso puentes de red confluyen y están gestionados por una interfaz de red virtual de tipo bridge que también se crea en el nodo en cuestión

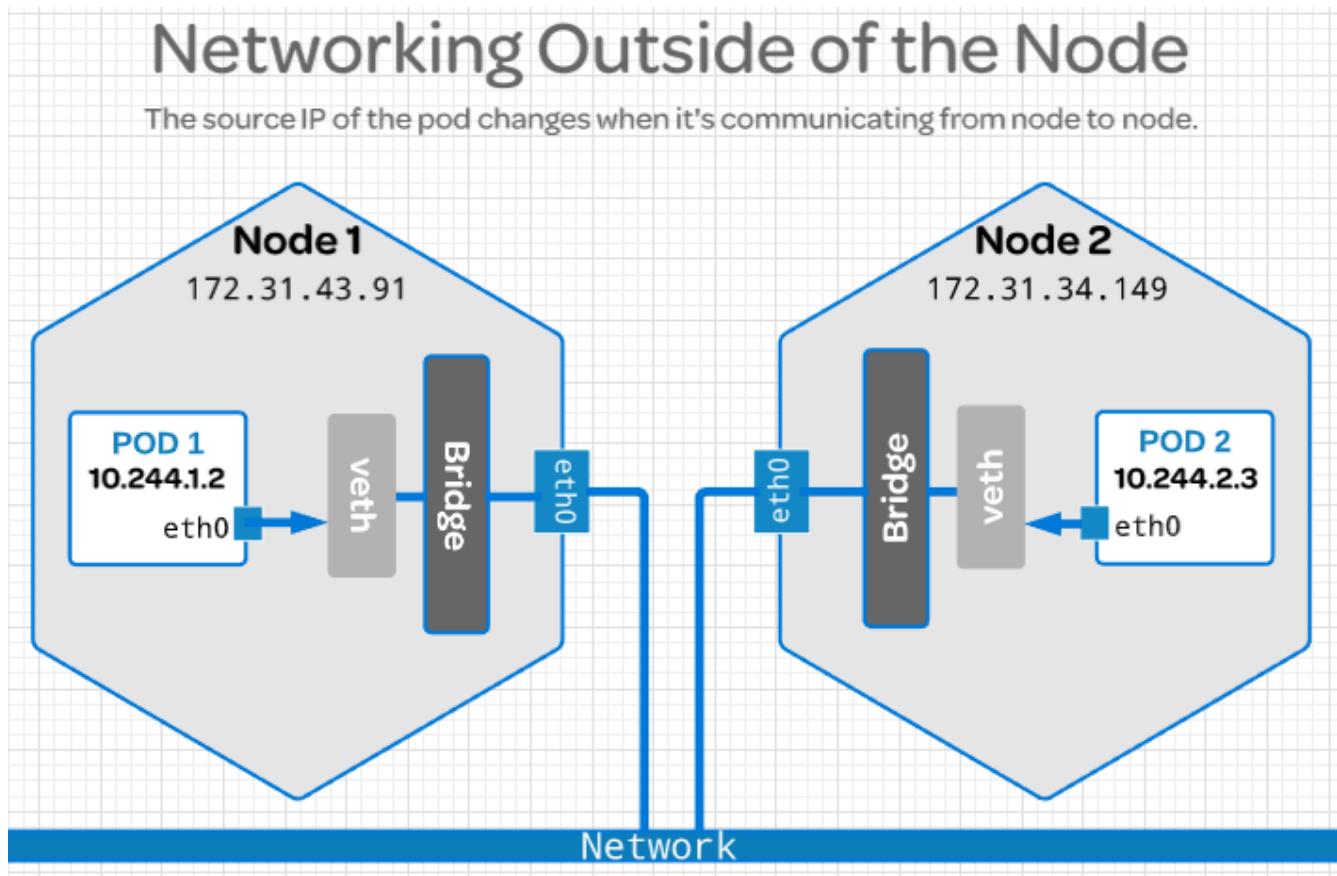
Networking Within a Node

A virtual ethernet interface pair is created for the container: one for the node's namespace and one for the container's network namespace.



24.4. Comunicación interpods (Distintos Nodos)

Cuando un Pod quiere comunicarse con el exterior, por ejemplo para llevar a cabo una salida a internet, el tráfico fluiría desde la red virtual bridge, hacia la tarjeta de red lógica o física del nodo en cuestión, representada en el esquema como la **eth0**



24.5. Ciclo de vida

El estado de un Pod en Kubernetes viene representado por un objeto denominado **PodStatus**, dicho objeto, tiene un campo que podemos consultar, llamado **phase**

La fase de un Pod es simple, se trata de un resumen de alto nivel donde se indica en la fase en la que se encuentra el Pod

La información de la fase de un Pod no pretende ser un bloque completo de observaciones del estado de los contenedores internos o el pod, ni pretende ser una máquina de estado totalmente integral

- **Pending**
 - Indica que el Pod ha sido aceptado por el sistema Kubernetes, pero uno o más de sus contenedores aún no han sido creados
 - Esta fase incluye el tiempo que Kubernetes tarda en llevar a cabo la programación con el Kube Scheduler, así como el tiempo que necesita para descargar imágenes a través de la red
 - Esta fase podría llevar cierto tiempo en ser completada

- **Running**

- Indica que el Pod ha sido vinculado a un nodo del clúster específico
- En esta fase, todos los contenedores que pudiera contener un Pod han sido creados
- Esta fase también indica que al menos un contenedor se está ejecutando o está en proceso de inicio/reinicio

- **Succeeded**

- Indica que todos los contenedores del Pod han arrancado de forma correcta y no es necesario ningún reinicio de los mismos

- **Failed**

- Indica que todos los contenedores del Pod han finalizado su ejecución
- Al menos un contenedor del Pod ha finalizado con un fallo
- En términos Docker - Sistema Operativo, el contendor en cuestión que falla finalizó con un código de error distinto de 0 por parte del sistema operativo

- **Unknown**

- Indica que por alguna razón no se puede obtener el estado del Pod
- Si en alguna ocasión encontramos esta fase en un Pod, es síntoma de que hay un error en la comunicación del propio host con el Pod

Los Pods realmente nunca se paran y se inician, sólo se crean y se destruyen



Kubernetes adopta la misma filosofía que el uso de Docker puro, sale más barato destruir y crear de nuevo :)

Posiblemente observaremos otros estados del Pod, que no forman explícitamente parte del ciclo de vida del Pod, pero indicarían ciertas operaciones que se están llevando a cabo

- **Terminating**

- El Pod está siendo eliminado

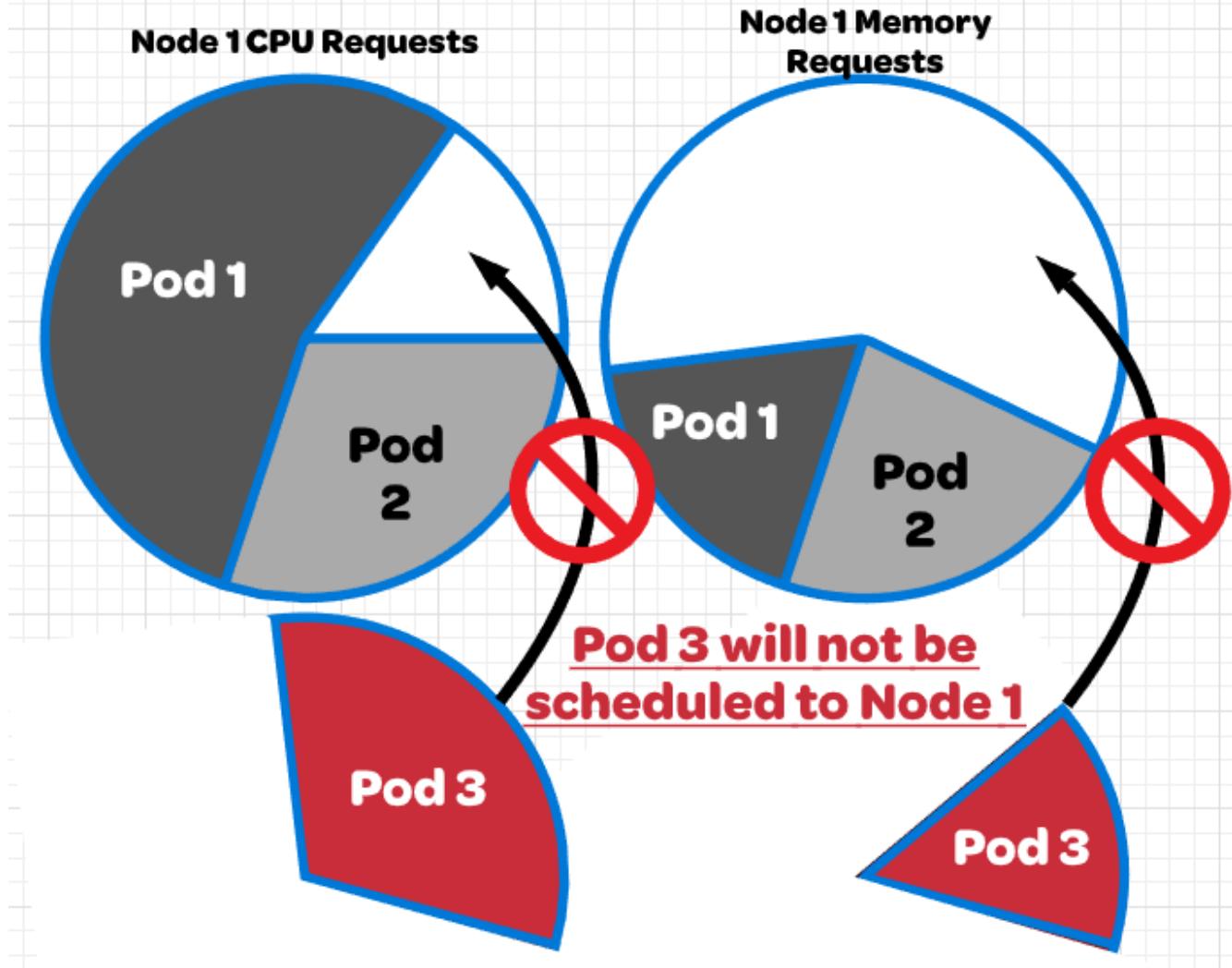
- **ContainerCreating**

- Indica que el contenedor dentro del Pod está siendo creado

24.6. Acotamiento de recursos

CPU and Memory Requests

Kubernetes allows you to use multiple schedulers with different rules simultaneously.



Por otra parte, podemos especificar cuotas de uso de recursos a los contenedores que se ejecutan dentro de los pods

Si un nodo no cumple las especificaciones de recursos necesarias disponibles por un pod, bien por CPU o RAM, el pod no será alojado en el nodo

Capítulo 25. Lab: Pods

Mediante esta laboratorio, vamos a trabajar toda la operatoria que es posible realizarle a los Pods

25.1. Creando 1 Pod con 1 container

Vamos a crear un primer Pod con un container dentro, el container será de una imagen Docker nginx en su última versión (latest)

Cuando creamos un Pod y no indicamos espacio de nombres alguno, el espacio de nombres por defecto es **default**

Si queremos realizar actualizaciones sobre el elemento que hemos creado, en este caso un pod, en lugar de crearlo con la orden kubectl create, podemos utilizar el comando kubectl apply



El efecto sería el mismo y en la práctica, resulta más cómodo crear elementos con kubectl apply ya que podemos actualizarlos, en lugar de estar creando y destruyendo elementos

- Mediante archivo de manifiesto

Creamos el archivo **pod-with-1-container.yml** e indicamos el siguiente contenido

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-1-container
spec:
  containers:
    - name: nginx
      image: nginx:latest
    ports:
      - containerPort: 80
```

Describimos la metadata indicada:

- **apiVersion**
 - Indicamos la versión de la api que queremos utilizar, actualmente estable la **v1**
- **kind**
 - Elemento de kubernetes que queremos utilizar, en nuestro caso indicamos **Pod**
- **metadata**
 - Indicamos el nombre del Pod
- **spec**
 - Sección que va a especificar características del Pod

- **spec → containers:**

- Sección que va a especificar el o los contenedores que tendrá alojados el Pod

- **spec → containers → name**

- Nombre del contenedor Docker

- **spec → containers → image**

- Nombre de la imagen Docker que queremos usar

- **spec → containers → ports**

- Sección que especifica los puertos a nivel interno que utilice el container (Indicamos el o los puertos que se indicaron en un Expose del Dockerfile)

- **spec → containers → ports → containerPort**

- Puerto que a nivel interno utiliza el container Docker que se encuentra dentro del Pod

Ejecutamos la creación

```
$ kubectl apply -f pod-with-1-container.yml
```

```
pod/pod-with-1-container created
```

- Mediante instrucción en línea

```
$ kubectl run nginx --image=nginx:latest --generator=run-pod/v1 --port=80
```

```
pod/nginx created
```

25.2. Listando los Pods

Para listar los pods del espacio de nombres por defecto **default**, indicamos lo siguiente

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
nginx	1/1	Running	0	23s
pod-with-1-container	1/1	Running	0	6m58s

También podemos ser explícitos, indicando que queremos listar un pod específico y no mostrando el listado completo

```
$ kubectl get pods/pod-with-1-container
```

NAME	READY	STATUS	RESTARTS	AGE
pod-with-1-container	1/1	Running	0	76m

25.3. Actualizando el Pod que tiene desplegado 1 container

Vamos a intentar actualizar el archivo anteriormente creado **pod-with-1-container.yml**, de forma que vamos a añadirle un segundo container, en este caso vamos a intentar añadirle un tomcat, para que el Pod aumentara a 2 containers

Indicamos el siguiente contenido al archivo **pod-with-1-container.yml**

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-1-container
spec:
  containers:
    - name: nginx
      image: nginx:latest
      ports:
        - containerPort: 80
    - name: tomcat
      image: tomcat:latest
      ports:
        - containerPort: 8080
```

Intentamos ejecutar la actualización

```
$ kubectl apply -f pod-with-1-container.yml
The Pod "pod-with-1-container" is invalid: spec.containers: Forbidden: pod updates may not add or remove containers
```

Nos encontramos con uno de los primeros problemas en nuestra casuística, efectivamente, el elemento Pod, una vez está en funcionamiento, no puede actualizarse agregando nuevos contenedores, tendría que crear un nuevo Pod con sendos containers y eliminar el antiguo

25.4. Creando 1 Pod con 2 containers

En este caso, vamos a crear 2 Pods que están íntimamente asociados, como podría ser el caso de un "frontend" que funciona con nginx y un "backend" que funciona con tomcat

- Mediante archivo de manifiesto

Creamos el archivo **pod-with-2-containers.yml** e indicamos el siguiente contenido

```

apiVersion: v1
kind: Pod
metadata:
  name: pod-with-2-container
spec:
  containers:
    - name: nginx
      image: nginx:latest
      ports:
        - containerPort: 80
    - name: tomcat
      image: tomcat:latest
      ports:
        - containerPort: 8080

```

Ejecutamos la creación

```
$ kubectl apply -f pod-with-2-containers.yml
pod/pod-with-2-container created
```

25.5. Añadiendo variables de entorno al Pod

Ahora, vamos a crear un nuevo Pod, que tendrá 1 container de la imagen de nginx, y le agregaremos una variable de entorno



Recordamos que las unidades Pod no pueden ser actualizadas en cuanto a estructura de contenedores

Creamos el archivo **pod-with-1-container-and-env.yml** y agregamos lo siguiente

```

apiVersion: v1
kind: Pod
metadata:
  name: pod-with-1-container-and-env
spec:
  containers:
    - name: nginx
      image: nginx:latest
      env:
        - name: DATABASE_IP
          value: 192.168.1.1
      ports:
        - containerPort: 80

```

Ejecutamos la creación

```
$ kubectl apply -f pod-with-1-container-and-env.yml
```

```
pod/pod-with-1-container-and-env created
```

25.6. Describiendo el Pod

Vamos a realizar una operación de obtención de metadata del Pod que tiene 2 containers asociados, el Pod que pusimos en ejecución con el archivo **pod-with-2-containers.yml**

```
$ kubectl describe pod pod-with-2-container

Name:      pod-with-2-container
Namespace:  default
Priority:   0
Node:      kubeminion2/192.168.15.102
Start Time: Tue, 24 Sep 2019 15:43:17 +0200
Labels:    <none>
Annotations: kubectl.kubernetes.io/last-applied-configuration:
            {"apiVersion":"v1","kind":"Pod","metadata":{"annotations":{},"name":"pod-with-2-
            container","namespace":"default"},"spec":{"containers":[{"...
Status:     Running
IP:        10.44.0.1
 IPs:
 IP: 10.44.0.1
Containers:
nginx:
  Container ID: docker://f21c73fb1842fdc5e44e877a9e1d7e554e7ba09b8e5044f6982788e60b14218a
  Image:        nginx:latest
  Image ID:    docker-pullable://nginx@sha256:9688d0dae8812dd2437947b756393eb0779487e361aa2ffbc3a529dca61f102c
  Port:        80/TCP
  Host Port:  0/TCP
  State:      Running
  Started:   Tue, 24 Sep 2019 15:43:28 +0200
  Ready:      True
  Restart Count: 0
  Environment: <none>
  Mounts:
    /var/run/secrets/kubernetes.io/serviceaccount from default-token-hpbhg (ro)
tomcat:
  Container ID: docker://0749ae275a80ce605e3de797571156ae4a4fcc5937ab78fb476bd2a96518093b
  Image:        tomcat:latest
  Image ID:    docker-pullable://tomcat@sha256:bb4ceffaf5aa2eba6c3ee0db46d863c8b23b263cb547dec0942e757598fd0c24
  Port:        8080/TCP
  Host Port:  0/TCP
  State:      Running
  Started:   Tue, 24 Sep 2019 15:49:08 +0200
  Ready:      True
  Restart Count: 0
  Environment: <none>
  Mounts:
    /var/run/secrets/kubernetes.io/serviceaccount from default-token-hpbhg (ro)
Conditions:
Type      Status
Initialized  True
Ready      True
ContainersReady  True
PodScheduled  True
Volumes:
default-token-hpbhg:
  Type:  Secret (a volume populated by a Secret)
```

```

SecretName: default-token-hpbbg
Optional: false
QoS Class: BestEffort
Node-Selectors: <none>
Tolerations: node.kubernetes.io/not-ready:NoExecute for 300s
    node.kubernetes.io/unreachable:NoExecute for 300s
Events:
  Type  Reason  Age   From           Message
  ----  -----  --  --  -----
  Normal Scheduled <unknown>  default-scheduler  Successfully assigned default/pod-with-2-container to kubeminion2
  Normal Pulling  28m   kubelet, kubeminion2  Pulling image "nginx:latest"
  Normal Pulled   28m   kubelet, kubeminion2  Successfully pulled image "nginx:latest"
  Normal Created  28m   kubelet, kubeminion2  Created container nginx
  Normal Started  28m   kubelet, kubeminion2  Started container nginx
  Normal Pulling  28m   kubelet, kubeminion2  Pulling image "tomcat:latest"
  Normal Pulled   23m   kubelet, kubeminion2  Successfully pulled image "tomcat:latest"
  Normal Created  23m   kubelet, kubeminion2  Created container tomcat
  Normal Started  23m   kubelet, kubeminion2  Started container tomcat

```

25.7. Obteniendo manifiesto YAML a partir de un objeto desplegado en kubernetes

Una vez desplegado un objeto en kubernetes, podemos obtener la descripción del mismo en formato YAML

Vamos a obtener la descripción del objeto previamente desplegado **pod-with-2-container**

```
$ kubectl get pod pod-with-2-container -o yaml
```

```

apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"v1","kind":"Pod","metadata":{"annotations":{},"name":"pod-with-2-
      container","namespace":"default"},"spec":{"containers":[{"image":"nginx:latest","name":"nginx","ports":[{"containerPort":80}]}},{"image":"tomcat:alpine","name":"tomcat","ports":[{"containerPort":8080}]}]}}
```

...

25.8. Probando conexiones HTTP con el Pod

Una vez hemos llevado a cabo un describe al Pod con nombre **pod-with-2-container**, observamos que ambos contenedores, comparten la misma IP interna que Kubernetes ha otorgado al Pod, en este caso obtenemos la siguiente IP: **10.44.0.1**

Hacemos un curl a esa IP con el puerto 80 y deberíamos de observar la página index.html del propio nginx

```
$ curl 10.44.0.1:80
```

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
body {
    width: 35em;
    margin: 0 auto;
    font-family: Tahoma, Verdana, Arial, sans-serif;
}
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

25.9. Conectándonos dentro del Pod (Origen Mode :D)

Imaginemos ahora, que queremos conectarnos dentro del Pod, por que queremos llevar a cabo ciertas operaciones de comprobación, o bien para detectar un posible error que se esté produciendo en el funcionamiento de la aplicación

Vamos a conectaros dentro del Pod que tiene desplegados 2 contenedores, de forma que accederemos a una bash específica como usuario root



Fondos Europeos



MINISTERIO
GOBIERNO DE ESPAÑA
DE EDUCACIÓN, FORMACIÓN PROFESIONAL
Y DEPORTE

Cofinanciado por
la Unión Europea



```
$ kubectl exec -it pod-with-2-container bash

root@pod-with-2-container:/# ls -la
total 8
drwxr-xr-x 1 root root 40 Sep 24 13:43 .
drwxr-xr-x 1 root root 40 Sep 24 13:43 ..
-rw xr-xr-x 1 root root 0 Sep 24 13:43 .dockerenv
drwxr-xr-x 2 root root 4096 Sep 10 00:00 bin
drwxr-xr-x 2 root root 6 Aug 30 12:31 boot
drwxr-xr-x 5 root root 360 Sep 24 13:43 dev
drwxr-xr-x 1 root root 66 Sep 24 13:43 etc
drwxr-xr-x 2 root root 6 Aug 30 12:31 home
drwxr-xr-x 1 root root 56 Sep 12 14:37 lib
drwxr-xr-x 2 root root 34 Sep 10 00:00 lib64
drwxr-xr-x 2 root root 6 Sep 10 00:00 media
drwxr-xr-x 2 root root 6 Sep 10 00:00 mnt
drwxr-xr-x 2 root root 6 Sep 10 00:00 opt
dr-xr-xr-x 141 root root 0 Sep 24 13:43 proc
drwx----- 1 root root 27 Sep 24 14:06 root
drwxr-xr-x 1 root root 38 Sep 24 13:43 run
drwxr-xr-x 2 root root 4096 Sep 10 00:00 sbin
drwxr-xr-x 2 root root 6 Sep 10 00:00 srv
dr-xr-xr-x 13 root root 0 Sep 24 14:17 sys
drwxrwxrwt 1 root root 6 Sep 12 14:37 tmp
drwxr-xr-x 1 root root 66 Sep 10 00:00 usr
drwxr-xr-x 1 root root 19 Sep 10 00:00 var
root@pod-with-2-container:/#
```

- Ahora vamos a conectar dentro del contenedor específico de nginx
 - Utilizamos previamente el comando **kubectl describe...**, de esa forma podemos conocer el nombre de cada contenedor

```
$ kubectl exec -it pod-with-2-container -c nginx
```

Una vez dentro del container, indicamos que nos muestre la versión del servidor web para verificar que efectivamente estamos conectados

```
root@pod-with-2-container:/# nginx -v
```

```
nginx version: nginx/1.17.3
```

- Ahora vamos a conectar dentro del contenedor específico de tomcat
 - Utilizamos previamente el comando **kubectl describe...**, de esa forma podemos conocer el nombre de cada contenedor

```
$ kubectl exec -it pod-with-2-container -c tomcat bash
```

Dentro del container, ejecutamos un listado de directorio, mostrando en principio la ruta por defecto del servidor de aplicaciones tomcat

También ejecutamos que nos muestre la versión de la JDK

```
root@pod-with-2-container:/usr/local/tomcat# ls -la
total 120
drwxr-sr-x 1 root staff 42 Sep 20 01:40 .
drwxrwsr-x 1 root staff 20 Sep 14 01:41 ..
-rw-r--r-- 1 root root 19318 Sep 16 18:19 BUILDING.txt
-rw-r--r-- 1 root root 5407 Sep 16 18:19 CONTRIBUTING.md
-rw-r--r-- 1 root root 57011 Sep 16 18:19 LICENSE
-rw-r--r-- 1 root root 1726 Sep 16 18:19 NOTICE
-rw-r--r-- 1 root root 3255 Sep 16 18:19 README.md
-rw-r--r-- 1 root root 7139 Sep 16 18:19 RELEASE-NOTES
-rw-r--r-- 1 root root 16262 Sep 16 18:19 RUNNING.txt
drwxr-xr-x 2 root root 331 Sep 20 01:40 bin
drwxr-sr-x 1 root root 22 Sep 24 13:49 conf
drwxr-sr-x 2 root staff 78 Sep 20 01:40 include
drwxr-xr-x 2 root root 4096 Sep 20 01:40 lib
drwxrwxrwx 1 root root 177 Sep 24 13:49 logs
drwxr-sr-x 3 root staff 151 Sep 20 01:40 native-jni-lib
drwxrwxrwx 2 root root 30 Sep 20 01:40 temp
drwxr-xr-x 7 root root 81 Sep 16 18:17 webapps
drwxrwxrwx 1 root root 22 Sep 24 13:49 work
```

```
root@pod-with-2-container:/usr/local/tomcat# java -version
openjdk version "1.8.0_222"
OpenJDK Runtime Environment (build 1.8.0_222-b10)
OpenJDK 64-Bit Server VM (build 25.222-b10, mixed mode)
```

```
root@pod-with-2-container:/usr/local/tomcat#
```

25.10. Obteniendo logs del Pod

Otro aspecto importante en el mundo de la orquestación de contenedores, es la obtención de los logs de los contenedores

En el caso de que el Pod contenga el despliegue de 2 contenedores, deberemos de ser explícitos, indicando sobre cual contenedor queremos obtener los logs

Vamos a obtener los logs del Pod que tiene desplegados dos contenedores **pod-with-2-container**, concretamente del contenedor nginx

- Obtenemos los logs de 1 Pod que tiene dos contenedores, indicamos que deseamos logs del contenedor nginx

- Indicamos en el navegador <http://10.44.0.1:80>, y recargamos la página varias veces esto creará varias peticiones HTTP que serán mostradas en la consola

```
$ kubectl logs pod-with-2-container -c nginx
10.32.0.1 - [24/Sep/2019:15:07:51 +0000] "GET / HTTP/1.1" 304 0 "-" "Mozilla/5.0 (X11; Linux x86_64; rv:60.0)
Gecko/20100101 Firefox/60.0" "-"
10.32.0.1 - [24/Sep/2019:15:07:54 +0000] "GET / HTTP/1.1" 304 0 "-" "Mozilla/5.0 (X11; Linux x86_64; rv:60.0)
Gecko/20100101 Firefox/60.0" "-"
```

El comando kubectl logs admite varios modificadores, entre otros podemos indicar:

- **-f**: Modo "follow", obtendremos traza en streaming
- **--tail=2**: Corte por la cola, indicamos el número de líneas más recientes a partir de las cuales visualizar el log, en el ejemplo, las 2 últimas líneas en orden cronológico descendente

```
$ kubectl logs -f --tail=2 pod-with-2-container -c nginx
```

- Obtenemos los logs de 1 Pod que tiene desplegados 2 contenedores
 - Queremos obtener todos los logs de todos los contenedores
 - Observamos traza tanto del contenedor con nginx como del contenedor tomcat

```
$ kubectl logs pod-with-2-container --all-containers=true
...
10.32.0.1 - [24/Sep/2019:15:07:51 +0000] "GET / HTTP/1.1" 304 0 "-" "Mozilla/5.0 (X11; Linux x86_64; rv:60.0)
Gecko/20100101 Firefox/60.0" "-"
10.32.0.1 - [24/Sep/2019:15:07:54 +0000] "GET / HTTP/1.1" 304 0 "-" "Mozilla/5.0 (X11; Linux x86_64; rv:60.0)
Gecko/20100101 Firefox/60.0" "-"
24-Sep-2019 15:05:58.143 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Server version name: Apache
Tomcat/8.5.46
24-Sep-2019 15:05:58.215 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Server built: Sep 16
2019 18:16:19 UTC
24-Sep-2019 15:05:58.216 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Server version number:
8.5.46.0
24-Sep-2019 15:05:58.216 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log OS Name: Linux
24-Sep-2019 15:05:58.217 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log OS Version: 3.10.0-
957.12.2.el7.x86_64
...
```

25.11. Attach a un Pod

Nos puede interesar mantenernos a la escucha de un Pod, de forma que podamos adjuntarnos al proceso y obtener streaming de traza en consola

Este comando sería parecido a establecer una escucha de traza en modo streaming

Ejecutamos en consola

```
$ kubectl attach pod-with-1-container
Defaulting container name to nginx.
Use 'kubectl describe pod/pod-with-1-container -n default' to see all of the containers in this pod.
If you don't see a command prompt, try pressing enter.

10.32.0.1 -- [24/Sep/2019:16:03:44 +0000] "GET / HTTP/1.1" 304 0 "-" "Mozilla/5.0 (X11; Linux x86_64; rv:60.0)
Gecko/20100101 Firefox/60.0" "-"
10.32.0.1 -- [24/Sep/2019:16:04:11 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (X11; Linux x86_64; rv:60.0)
Gecko/20100101 Firefox/60.0" "-"
```

Ahora, accedemos a la URL del navegador con la IP interna del Pod al puerto 8080 y deberíamos de observar como va apareciendo traza del proceso

Para liberar la consola, únicamente presionamos Ctrl + C

25.12. Forwarding de puerto a un Pod

Otra opción que también resulta de bastante interés, sobre todo cuando queremos llevar a cabo un testeo rápido de contenedores y acceder a su contenido, es llevar a cabo una redirección de puertos del host hacia el contenedor

Realizamos un port forwarding de forma que indicamos el nombre del Pod y abrimos el puerto 8081 del anfitrión para que se conecte con el puerto 80 del contenedor

```
$ kubectl port-forward pod-with-1-container 8081:80
```

```
Forwarding from 127.0.0.1:8081 -> 80
Forwarding from [::1]:8081 -> 80
Handling connection for 8081
Handling connection for 8081
```

Ahora, abrimos el navegador y ponemos <http://localhost:8081>, accederemos al contenido del Pod, en nuestro caso observaremos el index.html del nginx

25.13. Selección del nodo para el Pod

Ahora queremos conseguir el efecto de regular el emplazamiento donde se ejecutan los Pods, deseamos ser selectivos, indicar que el primer Pod va a alojarse en el nodo minion1 y el segundo Pod que vamos a definir, va a alojarse en el nodo minion2

Lo primero, va a ser, etiquetar el nodo minion1 con la etiqueta **app=frontend**

```
$ kubectl label nodes kubeminion1 app=frontend
```

A continuación, etiquetamos el nodo minion2 con la etiqueta **app=backend**

```
$ kubectl label nodes kubeminion2 app=backend
```

Creamos el archivo Creamos el archivo **pod-to-frontend-node.yml** e indicamos el siguiente contenido

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-to-frontend-node
spec:
  containers:
    - name: nginx
      image: nginx:latest
      ports:
        - containerPort: 80
  nodeSelector:
    app: frontend
```

Ejecutamos la orden de despliegue

```
$ kubectl apply -f pod-to-frontend-node.yml
pod/pod-to-frontend-node created
```

A continuación, creamos el archivo **pod-to-backend-node.yml** e indicamos el siguiente contenido

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-to-backend-node
spec:
  containers:
    - name: tomcat
      image: tomcat:alpine
      ports:
        - containerPort: 8080
  nodeSelector:
    app: backend
```

Ejecutamos la orden de despliegue

```
$ kubectl apply -f pod-to-backend-node.yml
pod/pod-to-backend-node created
```

Por último, obtenemos información detallada del despliegue de Pods que tenemos ahora, observamos que cada Pod ha caído en un nodo diferente

```
$ kubectl get pods -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE	NOMINATED NODE	READINESS GATES
pod-to-backend-node	1/1	Running	0	62s	10.44.0.1	kubeminion2	<none>	<none>
pod-to-frontend-node	1/1	Running	0	6m47s	10.36.0.1	kubeminion1	<none>	<none>

25.14. Afinidad del Pod (Node affinity)

La característica de afinidad consiste en 2 tipos (Node Affinity e Inter Pod Affinity)

Node affinity * Permite restringir en qué nodos el pod es elegible para ser puesto en marcha, según etiquetas que tenga el nodo * Lo indicamos en el Pod con el tag **nodeAffinity**

Actualmente existen 2 tipos de afinidad de nodo:

- **requiredDuringSchedulingIgnoredDuringExecution**

- Especifica las reglas que deben cumplirse para que un Pod se programe en un nodo
- Es parecido a utilizar el tag **nodeSelector**, pero empleando una sintaxis mucho más expresiva
- Se interpreta como "Sólo ejecuta el Pod en el nodo con la etiqueta que te indico, sino no"

- **preferredDuringSchedulingIgnoredDuringExecution**

- Especifica las preferencias que el Kube Scheduler intentará imponer, pero no está garantizado
- Se interpreta como "Intenta ejecutar el Pod en un nodo cuya etiqueta te indico, si no es posible, intenta ejecutarlo en otro nodo"



El término **IgnoredDuringExecution**, indica que si las etiquetas de un nodo cambian mientras se está ejecutando el Pod y ya no se cumplen las reglas de afinidad del Pod, este continuará ejecutándose en el Nodo



Si especificamos en un mismo Pod, reglas de **nodeSelector** y **nodeAffinity**, ambas reglas deben de ser satisfechas por el Pod para que el Kube Scheduler indique cual es nodo candidato



Si especificamos en un mismo Pod, múltiples **nodeSelectorTerms** asociados a un elemento **nodeAffinity**, el Pod podrá desplegarse en un nodo que al menos cumpla una de las reglas



Si especificamos en un mismo Pod, múltiples **matchExpressions** asociados a un elemento **nodeSelectorTerms**, el Pod podrá desplegarse en un nodo que cumpla todas las reglas

Si eliminamos o cambiamos la etiqueta del nodo donde el Pod está ejecutándose, el Pod no será eliminado



La afinidad de selección de nodo, sólamente opera en el momento de poner en marcha el Pod, no una vez este está ya en funcionamiento en un nodo en cuestión

Creamos el archivo **pod-with-node-affinity.yml** e indicamos el siguiente contenido

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-node-affinity
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: kubernetes.io/os
                operator: In
                values:
                  - linux
                  - mac
      preferredDuringSchedulingIgnoredDuringExecution:
        - weight: 1
          preference:
            matchExpressions:
              - key: kubernetes.io/hostname
                operator: In
                values:
                  - kubeminion2
  containers:
    - name: nginx-with-node-affinity
      image: nginx:latest
```

- Indicamos que queremos desplegar un pod con un container de nginx
- Indicamos que el Pod únicamente puede ser emplazado en un nodo cuyo sistema operativo sea linux o mac
- Se soportan los operadores In, NotIn, Exists, DoesNotExist, Gt, Lt
 - Para producir un efecto "anti afinidad", podemos utilizar los operadores NotInt y DoesNotExists
- Adicionalmente, entre los nodos que cumplen ese criterio, se deben preferir los nodos con una etiqueta cuya clave sea el nombre del host (hostname) y cuyo valor sea minion2
- Indicamos como peso 1
 - El peso puede indicarse en el rango 1 - 100
 - Para cada nodo que cumpla con todos los requisitos de programación (solicitud de recursos,

expresiones de afinidad, etc.) el Kube Scheduler suma iterando a través de los elementos de expresión de este campo

- Agregará una ponderación a la suma de expresiones que vaya encontrando en los nodos, si el nodo coincide con el correspondiente **matchExpressions**
- Las puntuaciones que calcula kubernetes se combinan con las puntuaciones de otras funciones prioritarias para el nodo como recursos disponibles, etc.
- Los nodos con la puntuación total más alta son los más preferidos

Ejecutamos la creación:

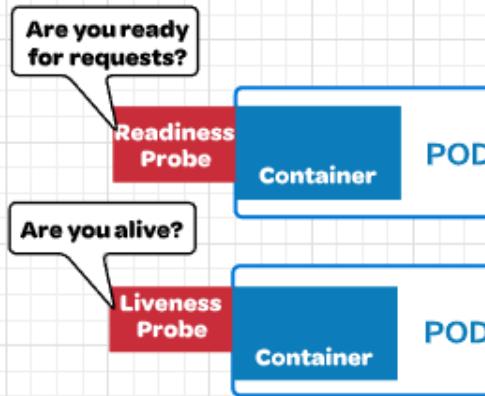
```
$ kubectl apply -f pod-with-node-affinity.yml
```

```
pod/pod-with-node-affinity created
```

25.15. Health Check: livenessProbe

Liveness and Readiness Probes

Liveness and readiness probes can be used to automatically restart containers if they are failing or automatically remove them from a service endpoint.



En ocasiones, nuestras aplicaciones necesitarán aplicar ciertos testeos para verificar que el sistema está funcionando de forma correcta

Si no indicamos ningún tipo de regla de chequeo, a pesar de que la aplicación a nivel interno tenga un problema (por ejemplo, fallo de respuesta HTTP 500), si el PID del proceso en el Kernel se

encuentra levantado... Para kubernetes, la aplicación estará bien

La configuración del **livenessProbe** va a solucionarlos la vida en cuanto a verificación de contenedores de pods que no están sanos por que no funcionan de la forma que esperamos

Vamos a implementar una regla de chequeo, para que cada cierto tiempo, kubernetes auto-compruebe nuestra aplicación, de forma que:

- Si recibimos un código HTTP de respuesta entre 200 - 400 nuestra aplicación estará bien
- Si recibimos un código HTTP fuera de ese rango, kubernetes de forma automática, iniciará el reinicio del Pod

Creamos el archivo **pod-with-health-check-livenessProbe.yml** e indicamos el siguiente contenido:

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: pod-with-health-check-livenessProbe
spec:
  containers:
    - name: nginx
      image: nginx:alpine
      livenessProbe:
        httpGet:
          path: /
          port: 80
      initialDelaySeconds: 3
      periodSeconds: 3
```

- Indicamos que queremos desplegar un Pod
- Indicamos una etiqueta al pod, con el par clave valor **test: liveness**
- Indicamos como nombre del pod **liveness-nginx-http**
- Indicamos que al contenedor, aplicamos una regla de salud basada en **httpGet**
- Indicamos que el path de ejecución de la aplicación, debe de ser /
- Indicamos que el puerto del container al cual debe de realizar peticiones de salud, es el 80
- Indicamos el tiempo de espera antes de dar por fallido el test de salud **initialDelaySeconds**
- Indicamos cada cuanto tiempo queremos que la regla de chequeo de salud se ejecute **periodSeconds**

Ejecutamos la creación del pod:

```
$ kubectl apply -f pod-with-health-check-livenessProbe.yml
```

```
pod/pod-with-health-check-livenessProbe created
```

25.16. Health Check: readinessProbe

Otro aspecto que nos puede resultar bastante útil de configurar, es indicar cuando un contenedor está listo para operar.

Casos de uso de aplicación, podrían ser, un contenedor que necesita inicializar una caché de datos antes de comenzar a operar... Un contenedor que necesita realizar una operación de levantamiento pesada antes de poder recibir tráfico entrante, como un servidor de aplicaciones, etc.

La configuración del **readinessProbe** va a realizar comprobaciones cada cierto tiempo, de forma que kubernetes va a poder tener un mecanismo especializado por contenedor, para saber si puede enviarle tráfico al contenedor en cuestión o no.

Con la regla `readinessProbe` que vamos a implementar, vamos a conseguir lo siguiente:

- Si recibimos un código HTTP de respuesta entre 200 - 400 nuestra aplicación estará lista para recibir tráfico
- Si recibimos un código HTTP fuera de ese rango, el Pod no recibirá tráfico, por que kubernetes entenderá que "todavía no se ha levantado la aplicación"

Creamos el archivo `pod-with-health-check-readinessprobe.yml` e indicamos el siguiente contenido:

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-health-check-readinessprobe
spec:
  containers:
    - name: tomcat
      image: tomcat:alpine
      ports:
        - containerPort: 8080
  readinessProbe:
    periodSeconds: 120
    httpGet:
      path: /
      port: 8080
```

- Cada 120 segundos se realiza una invocación http a la url "/" del contenedor, mediante el puerto 8080 y se comprueba si la aplicación puede recibir tráfico

Ejecutamos la creación del Pod:

```
$ kubectl apply -f pod-with-health-check-readinessprobe.yml
```

```
pod/pod-with-health-check-readinessprobe created
```

Mediante esta regla de salud, estamos indicando al clúster, si el tráfico debe de fluir al contenedor o no (si está Running)



En caso de que la aplicación tuviera un fallo y dejara de dar servicio, por ejemplo, por que se produjera un fallo de NullPointerException interno, la regla del readinessProbe determinaría que el contenedor no está sano, pero no habría ningún tipo de reinicio al respecto, por que de eso... Se encarga el **livenessProbe**

25.17. Health Check: startupProbe + sideCar

Cuando a un Pod le configuramos la sección de **startupProbe**, vamos a indicar el tiempo máximo de cortesía que vamos a otorgar al contenedor para que se levante y entre en operatividad plena.

El factor de tiempo de la sección **startupProbe** se configura mediante la multiplicación de 2 variables:

- failureThreshold
- periodSeconds

Resultando el cálculo del tiempo máximo: $\text{failureThreshold} * \text{periodSeconds}$

Vamos a implementar de forma conjunta un patrón sidecar, de forma que tendremos 1 contenedor que será un servidor de aplicaciones y pegado al mismo, 1 contenedor que ejecutará cada cierto tiempo una ejecución de un comando específico.

Sería el típico caso de tener 1 Pod que actúa de servidor backend, y al lado un programa que actúa procesando archivos de ese primer contenedor, para enviarlos por correo, hacer cálculos, guardar en base de datos, etc.

Creamos el archivo **pod-with-health-check-startupprobe-sidecar.yml** e indicamos el siguiente contenido:

```

apiVersion: v1
kind: Pod
metadata:
  name: pod-with-health-check-startupprobe-sidecar
spec:
  containers:
    - name: tomcat
      image: tomcat:alpine
      ports:
        - containerPort: 8080
      livenessProbe:
        httpGet:
          path: /
          port: 8080
        periodSeconds: 5
        initialDelaySeconds: 5
      readinessProbe:
        periodSeconds: 120
        httpGet:
          path: /
          port: 8080
      startupProbe:
        httpGet:
          path: /
          port: 8080
        failureThreshold: 30
        periodSeconds: 10
    - name: file-processor
      image: alpine:latest
      command:
        - sh
        - "-c"
        - "while true;do ls;sleep 5; done"
  restartPolicy: OnFailure

```

- Indicamos 1 primer contenedor que actúa de backend, con un tomcat
 - Le indicamos como regla de salud **livenessProbe**, que realice una petición al raíz /, a través del puerto 8080, cada 5 segundos, con inicialmente un retraso al comenzar el proceso de liveness de 5 segundos
 - Le indicamos como regla de salud **readinessProbe**, que realice una petición al raíz /, a través del puerto 8080, cada 120 segundos, para determinar si el contenedor está listo para operar
 - Le indicamos como regla de salud **startupProbe**, que para que el contenedor arranque, realice peticiones al raíz /, a través del puerto 8080, y que como máximo esté haciendo esto 5 minutos ($30 * 10 = 300$ segundos)
- Indicamos 1 segundo contenedor, con un mero juego de instrucciones alpine, que ejecute un script cada 5 segundos, en nuestro caso, cada 5 segundos, aparecerá en la estándar output el

listado de archivos interno del raíz del contenedor

- Como política de reinicio del Pod **OnFailure**, únicamente debe de realizarse, si los contenedores sufren algún fallo
 - En el caso del contenedor con nombre file-processor, este arrancará y se parará, pero eso no es un fallo y no podemos dejar que kubernetes aplique la política del Always por que estaríamos reiniciando un Pod completo cuando no tiene fallos

Ejecutamos la creación del Pod:

```
$ kubectl apply -f pod-with-health-check-startupprobe-sidecar.yml  
pod/pod-with-health-check-startupprobe-sidecar created
```



25.18. Límites de recursos

Cuando desplegamos Pods en nuestro clúster, debería ser obligatorio el acotamiento de los recursos que desplegamos en el mismo

Si no acotamos recursos a los Pods, estos podrán crecer hasta la totalidad de recursos del servidor, por otra parte, el planificador kube-scheduler, tampoco podrá llevar a cabo una buena fase de planificación en cuanto a la puesta en marcha de los Pods

Vamos a crear el archivo **pod-with-limit-resources.yml** e indicamos el siguiente contenido:

```
apiVersion: v1  
kind: Pod  
metadata:  
  labels:  
    app: test-resources  
    name: pod-with-limit-resources  
spec:  
  containers:  
    - name: tomcat  
      image: tomcat:8  
      resources:  
        limits:  
          memory: "512Mi"  
          cpu: "1000m"  
        requests:  
          memory: "50Mi"  
          cpu: "100m"
```

MINISTERIO
DE EDUCACIÓN, FORMACIÓN PROFESIONAL
Y DEPORTE

GOBiERNO
DE ESPAÑA
Cofinanciado por
la Unión Europea



Dentro del bloque de resources, observamos 2 secciones (limits y requests).

- **limits**

- El uso de CPU, está indicado en unidades "milicore", por lo que 1000m sería lo que

- corresponde al uso de un núcleo de CPU
- El uso de RAM, está indicado en unidades "MB", por lo que 512Mi corresponden a 512MB
 - Podemos indicar que estos datos de memoria RAM y CPU sería el "techo" máximo que el contenedor podría adquirir
- **requests**
 - El uso de CPU, está indicado en unidades "milicore", por lo que 100m sería lo que corresponde al uso de 0.1 núcleo de CPU
 - El uso de RAM, está indicado en unidades "MB", por lo que 50Mi corresponden a 50MB
 - Podemos indicar que estos datos de memoria RAM y CPU sería el "suelo" mínimo que el contenedor necesitaría para poder comenzar a operar

Ejecutamos la creación del pod:

```
$ kubectl apply -f pod-with-limit-resources.yml  
pod/pod-with-limit-resources created
```

Para comprobar que correctamente los límites han sido ajustados, podemos realizar un describe del recurso:

```
$ kubectl describe pod pod-with-limit-resources
```

25.19. Eliminando los Pods

Podemos eliminar Pods uno por uno, o también suministrando el nombre de todos en una única instrucción

- Si queremos eliminar uno por uno

```
$ kubectl delete pod <POD_NAME>
```

- Si queremos eliminar varios pods de golpe

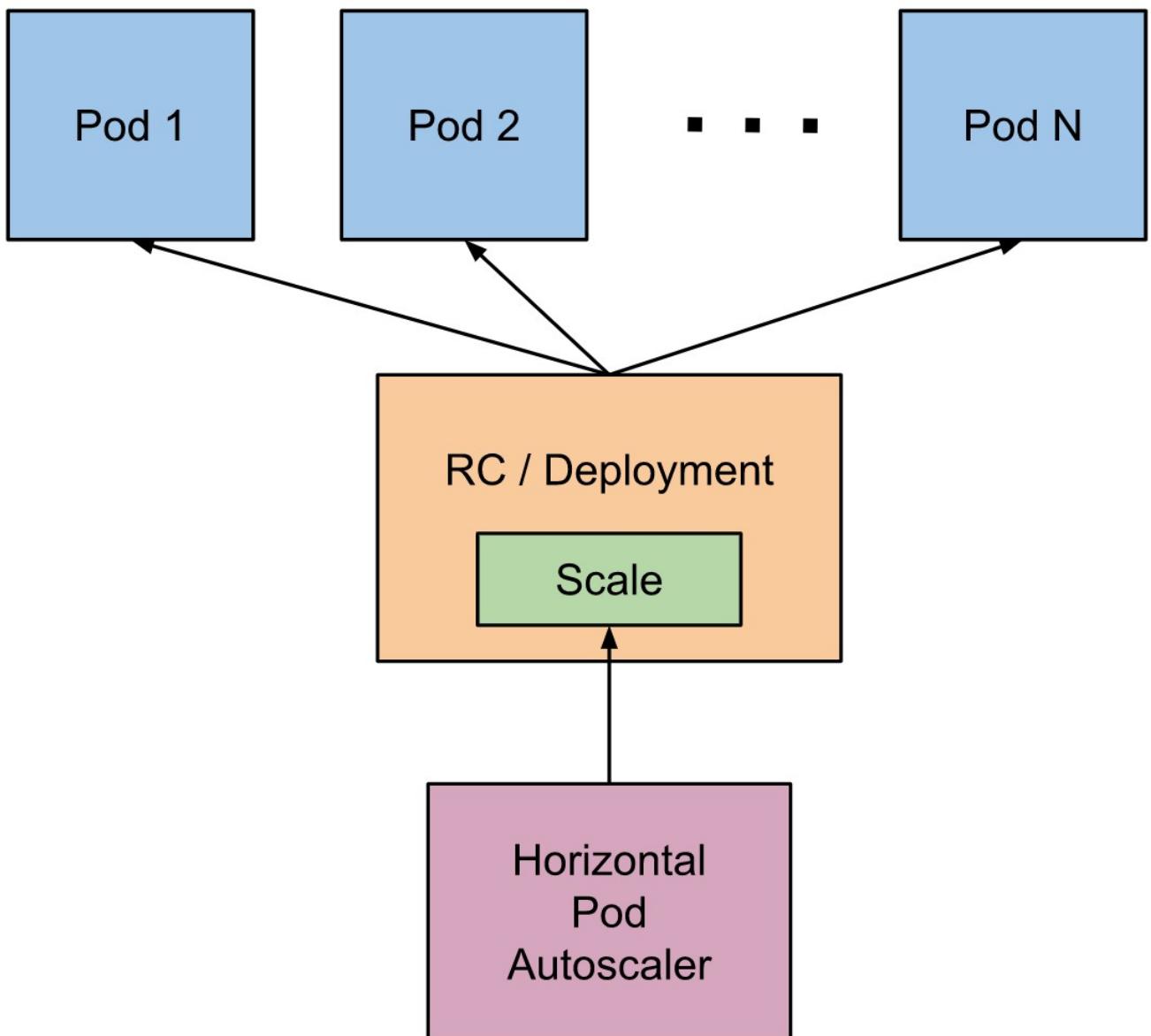
```
$ kubectl delete pod/<POD_NAME> pod/<POD_NAME> ...
```

- Si queremos eliminar todos los pods desplegados del namespace

```
$ kubectl delete pods --all=true
```

Capítulo 26. Autoescalado

26.1. Horizontal Pod Autoscaler



Mediante este tipo de escalado, kubernetes es capaz de automáticamente incrementar el número de pods de:

- Controladores de replicación
 - Como ReplicationController, ReplicaSet o Deployment

Se basa en la observación del consumo de la CPU (o con soporte de otro tipo de métricas personalizadas que la aplicación en sí pueda proporcionar)



Debemos de tener en cuenta que el autoescalado automático de Pod horizontal no se aplica a objetos que no pueden escalarse, como por ejemplo los DaemonSets

26.1.1. ¿Cómo funciona?

El Horizontal Pod Autoscaler es implementado en Kubernetes como un recurso objeto más del API, y en si actúa como un controlador.

El recurso, determina el comportamiento del controlador.

El controlador ajusta periódicamente el número de réplicas en un controlador de replicación o deployment, el objetivo, que el número de replicas se adapte al promedio de carga de CPU que hayamos indicado.

El escalador funciona como un bucle de control, por defecto se ejecuta cada 15 segundos.

Cuando se ejecuta, el controlador consulta la utilización de recursos contra las métricas especificadas en cada definición del objeto HorizontalPodAutoscaler.

Obtiene las métricas de recursos a través de:

- El API (para las métricas de recursos por Pod)
- El API de métricas personalizadas (para todas las demás métricas)

Para las métricas de recursos por Pod (Como CPU), el controlador obtiene las métricas del API para cada elemento objetivo del HorizontalPodAutoscaler.

Al establecer un valor de utilización objetivo, el controlador realiza un cálculo del valor de utilización como un porcentaje de la solicitud de recursos equivalente en los contenedores de cada Pod.

Si alguno de los contenedores del Pod no tiene un conjunto de solicitudes de recursos relevante, la utilización de la CPU para el Pod no se definirá y el autoscaler no tomará ninguna medida para esa métrica.

Esto quiere decir, que deberemos de ajustar a los pos la configuración de **resource request**

Desde una perspectiva básica, el controlador HorizontalPodAutoscaler opera en la relación entre el valor métrico deseado y el valor métrico actual, con la siguiente fórmula:

```
desiredReplicas = [currentReplicas * ( currentMetricValue / desiredMetricValue )]
```

Capítulo 27. Lab: Autoescalado Horizontal Pod Autoscaler

Mediante este laboratorio, vamos a poner en marcha el autoescalado horizontal de Pods, basándonos en métricas de uso de CPU.

Para que el sistema de autoescalado funcione correctamente, debemos de tener ajustados 2 elementos fundamentales:

- Recursos acotados en los Pods
 - Limits (Lo máximo a consumir por réplica)
 - Requests (Lo mínimo necesario por réplica para que funcione)
- Un servidor de recolección de métricas, para que los autoescaladores puedan consultar las estadísticas de computación en todo momento



En el capítulo de monitoreo tenemos el laboratorio de instalación de un servidor de métricas

27.1. Creando manifiesto de autoescalado

Lo primero que vamos a hacer, es crear un manifiesto donde indicaremos los siguientes elementos:

- Un Deployment con un servidor web nginx
- Un Service de tipo ClusterIP para acceder a los Pods del Deployment
- Un Autoescalador basado en regla de CPU, indicando un nivel de alerta de uso de 100m (0,1 Núcleos de CPU), con un umbral de Pods en computación entre 1 y 10

Creamos el archivo **horizontal-pod-autoscaling.yml** y le agregamos el siguiente contenido:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deployment-for-horizontal-autoscaler
spec:
  replicas: 2
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
        - name: nginx
```

```

image: nginx:latest
ports:
- containerPort: 80
resources:
limits:
cpu: 200m
memory: 128Mi
requests:
cpu: 100m
memory: 64Mi
---
apiVersion: v1
kind: Service
metadata:
name: nginx-service
spec:
selector:
app: frontend
type: ClusterIP
ports:
- name: http
protocol: TCP
port: 8181
targetPort: 80
---
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
name: autoscaler-for-nginx
spec:
scaleTargetRef:
apiVersion: apps/v1
kind: Deployment
name: deployment-for-horizontal-autoscaler
minReplicas: 1
maxReplicas: 10
metrics:
- type: Resource
resource:
name: cpu
target:
type: AverageValue
averageValue: 100m

```

Aplicamos el manifiesto al clúster:

```
$ kubectl apply -f horizontal-pod-autoscaling.yml
```



27.2. Obteniendo la IP del servicio

A continuación, debemos de obtener la IP del servicio para realizarle peticiones masivas, ejecutamos el siguiente comando:

```
$ kubectl get svc
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	138m
nginx-service	ClusterIP	10.97.187.107	<none>	8181/TCP	102m

- Observamos que tenemos la IP de tipo CLUSTER-IP 10.97.187.107

27.3. Realizando peticiones con JMeter

A continuación, vamos a descargarnos el software JMeter, para poder realizar simulaciones de peticiones durante cierto tiempo al servicio.

Podemos descargarlo desde la siguiente URL: https://jmeter.apache.org/download_jmeter.cgi

Parametrizamos con lo siguiente:

- Sampler de tipo HTTPRequest
 - Protocol: http
 - IP: 10.97.187.107
 - Port: 8181
- Thread Group
 - Usuarios: 1000
 - Tiempo de levantamiento: 120s
 - Iteraciones: Infinitas

27.4. Observando el comportamiento del auto-escalado

Iniciamos JMeter para que bombardee el sistema con peticiones, realizamos una primera consulta para ver las reglas de auto-escalado:

```
$ kubectl get hpa --watch
```

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS		
AGE							
autoscaler-for-nginx	Deployment/deployment-for-horizontal-autoscaler	<unknown>/100m	1	10	2		
59m							
autoscaler-for-nginx	Deployment/deployment-for-horizontal-autoscaler	200m/100m	1	10	2		
59m							
autoscaler-for-nginx	Deployment/deployment-for-horizontal-autoscaler	200m/100m	1	10	4		
60m							
autoscaler-for-nginx	Deployment/deployment-for-horizontal-autoscaler	200m/100m	1	10	8		
60m							
autoscaler-for-nginx	Deployment/deployment-for-horizontal-autoscaler	200m/100m	1	10	10		
60m							

- Observamos como se analiza un TARGET de métricas actuales, donde sube a un uso de cpu de 200m y el número de réplicas comienza a subir hasta el máximo indicado, de 10.

Detenemos JMeter y dejamos transcurrir algunos minutos para ver como el escalador ordena la bajada de Pods hasta los niveles mínimos:

```
$ kubectl get hpa --watch
```

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS		
AGE							
autoscaler-for-nginx	Deployment/deployment-for-horizontal-autoscaler	200m/100m	1	10	10	73m	
autoscaler-for-nginx	Deployment/deployment-for-horizontal-autoscaler	56m/100m	1	10	10	73m	
autoscaler-for-nginx	Deployment/deployment-for-horizontal-autoscaler	0/100m	1	10	10	74m	
autoscaler-for-nginx	Deployment/deployment-for-horizontal-autoscaler	0/100m	1	10	10	79m	
autoscaler-for-nginx	Deployment/deployment-for-horizontal-autoscaler	0/100m	1	10	1	80m	

- Observamos como el número de réplicas decrece cuando la utilización de CPU baja del umbral de uso de 100m hasta alcanzar de nuevo el nivel mínimo de replicación

Capítulo 28. Jobs

Un Job crea uno o más Pods y garantiza que un número específico de ellos finalice con éxito.

A medida que los Pods se van completando con éxito, el Job rastrea las finalizaciones que han terminado con éxito.

Cuando se alcanza un número específico de finalizaciones con éxito, el Job queda completado.

Si eliminamos un Job, limpiaremos los Pods que este elemento de kubernetes haya creado.

Un caso de aplicación podría ser:

Asegurar que un Pod finaliza su trabajo con éxito, a pesar de fallos de hardware en el nodo, d reinicios del propio nodo que pueda interrumpir su trabajo, etc.

Podríamos utilizar también un Job para ejecutar varios Pods en paralelo.

Un job define por defecto un parámetro en la metadata **backoffLimit**, es ajustado a un valor por defecto de 6 si no decimos lo contrario, este parámetro nos indica el número de reintentos antes de considerar que el Job ha fallado en su ejecución.

28.1. Tipos de Jobs

- **Jobs simples**

- Normalmente sólo se arranca un Pod, a menos que el Pod falle, con lo que se intentará arrancar otro diferente
- El Job es completado lo más pronto posible, el Pod indica que ha terminado el trabajo correctamente cuando el proceso finaliza
- Podemos dejar sin configurar los elementos (si no los configuramos, por defecto están ajustados a 1):
 - **.spec.completions**
 - **.spec.parallelism**

- **Jobs en paralelo (Con un número fijo de ejecuciones)**

- Tenemos que especificar los siguientes elementos:
 - Un valor distinto de cero en el campo **.spec.completions** (cantidad de iteraciones finalizadas con éxito para dar el Job por completado)
- En este caso, el Job representa la tarea general
- El Job asume que se ha completado cuando haya un Pod que haya acabado con éxito X iteraciones, o sea, según lo que hayamos puesto en el campo **.spec.completions**



No está aún implementado, el que le podamos decir a un conjunto de Jobs en paralelo, valores diferentes para cada uno con **.spec.completions**

- **Jobs en paralelo (Funcionando como una cola)**

- En este caso tenemos que aplicar la siguiente configuración:
 - No debemos de especificar en el manifiesto valor para **.spec.completions**
 - Especificamos **.spec.parallelism** con un valor entero no negativo
- Los Pods deben de coordinarse entre ellos, o con un servicio externo para determinar en qué debería trabajar cada uno, por ejemplo, un Pod podría recuperar X elementos de una cola de trabajo
- Cada pod es capaz de determinar independientemente si todos sus "compañeros" han o no acabado, y por tanto si todo el job al completo está finalizado
- Cuando cualquier Pod acaba con éxito (success), no se crean nuevos Pods
- Una vez que al menos un Pod ha finalizado con éxito y todos los Pod han finalizado, entonces el Job se toma como finalizado con éxito (success)
- Una vez que cualquier Pod ha salido con éxito, ningún otro Pod debería estar haciendo ningún trabajo para esta tarea o escribiendo ningún resultado (todos deberían estar en proceso de **existing**)

28.2. Tratando errores en contenedores fallidos

Un contenedor en un Pod puede fallar por una serie de razones:

- Código de error de salida del sistema operativo distinto de 0
- El contenedor fue eliminado por exceder un límite de memoria establecido
- Etc.

Si sucede un fallo en el contenedor, y hemos especificado el valor en el manifiesto YAML de **.template.spec.restartPolicy = "OnFailure"**, estaríamos consiguiendo que el Pod permaneciera en el Nodo, pero que el contenedor volviera a ejecutarse.

Tenemos dos opciones para tratar errores de ejecución internos de los contenedores:

- El programa que ejecuta el contenedor, debe de manejar un caso de reinicio del propio servidor
- Especificar en el manifiesto **.template.spec.restartPolicy = "Never"** (Que no haya reinicios ante fallos) y **.spec.backoffLimit = X** (Que se relance la ejecución del Job un número de veces determinado)

Un Pod completo también puede fallar por otras razones:

- Cuando este es expulsado del nodo (El nodo se actualiza, reinicia, se elimina del clúster, se le añade una mancha "taint" que hace inviable seguir ejecutando en ese nodo el Pod, etc.)
- Si ocurre un fallo a nivel de contenedor y tenemos una **restartPolicy=Never** y no hemos especificado **backoffLimit**

Cuando un Pod falla, el controlador del Job, inicia un nuevo Pod



Es responsabilidad del que desarrolla el software que se está ejecutando dentro del Pod, controlar a nivel de ejecución qué archivos se estaban procesando, si había archivos temporales, bloqueos de algún tipo, resultados incompletos derivados de ejecuciones anteriores, etc.



Los Jobs de kubernetes, no nos van a resolver "mágicamente" el problema de continuar un procesamiento previamente lanzado e interrumpido, comenzando a partir del punto de fallo... Los Pods comenzarían a ejecutar los programas de nuevo de 0



Tenemos siempre que tener en cuenta, que, la política de reinicio, no aplica al Job en sí, sino, al Pod que se crea.

También tenemos que tener en cuenta que si incluso especificamos `.spec.parallelism=1` y `.spec.completions=1` y `.spec.template.spec.restartPolicy="Never"`, el mismo programa a veces se puede iniciar más de una vez, por circunstancias que hemos comentado ante fallos de hardware, etc.

Si especificamos `.spec.parallelism` y `.spec.completions`, ambos, a valores superiores a 1, entonces podría haber múltiples Pods ejecutándose a la vez de forma simultánea.

Es de nuevo, responsabilidad de la aplicación que se está ejecutando dentro de cada contenedor, tolerar ejecuciones en paralelo ante recursos que se vayan a tratar, como por ejemplo, procesamiento de archivos, gestión de sockets de red, etc.

28.3. Política backoffLimit

Hay situaciones en las que deseamos que fallen los Jobs después de haber realizado una serie de reintentos, debido a un error lógico, errores por configuración, etc.

Para conseguir ese efecto, debemos de establecer el parámetro `.spec.backoffLimit` para indicar el número de reintentos antes de considerar un Job como fallido

De forma predeterminada, si no indicamos lo contrario, tiene un valor de 6

Los controladores de los Jobs, recrean los Pods en caso necesario con un retraso exponencial de (10s, 20s, 40s ...)

El recuento del trabajo realizado se establece si no aparecen nuevos Pods fallidos antes de la próxima verificación del estado del Job.

Si el Job tiene configuración de **restartPolicy="OnFailure**, tenemos que tener en cuenta que el contenedor que ejecute el trabajo finalizará una vez que se haya alcanzado el límite del backoffLimit



Si ajustamos esa política de reinicio, tenemos que tener en cuenta que puede dificultar la depuración del programa interno del propio contenedor.

En principio, lo recomendable es ajustar la política de reinicio a lo siguiente **restartPolicy="Never"**, esto producirá el efecto de que al depurar el Job o al usar el sistema de log para comprobar que la salida de los trabajos con errores no se pierde sin darnos cuenta.

28.4. Limpieza de Jobs finalizados

Cuando un Job se completa, no se crean más Pods, pero los Pods que han finalizado, no son eliminados por defecto del clúster.

Mantener los Jobs nos permite en principio observar registros de los Pods completados para verificar si existe algún tipo de error, advertencias u otro resultado con fines de diagnóstico

El Job en sí, además de los Pods asociados también permanecen después que el propio Job se haya completado, de nuevo, para que se pueda ver su estado.

Es responsabilidad del propio operador eliminar los Jobs antiguos que han sido completados después de verificar (si es necesario) el estado de los mismos

Los jobs pueden ser borrados utilizando el siguiente comando:

```
$ kubectl delete job <JOB_NAME>
```



Cuando se elimina un Job, también son eliminados de forma automática los Pods asociados al Job

28.5. Terminación de Jobs

De manera predeterminada, un Job se ejecutará sin interrupción a menos que un Pod falle (**restartPolicy=Never**), o bien, un contenedor salga por error (**restartPolicy=OnFailure**)

Cuando sucede lo que acabamos de comentar, el Job evalúa lo que indica la configuración del manifiesto **.spec.backoffLimit**

Una vez se hayan alcanzado el número de reintentos del valor de **backoffLimit**, el Job se marcará como fallido y cualquier Pod que estuviera en ejecución en ese momento será finalizado

Otra forma de finalizar un Job es, estableciendo un tiempo de "deadLine".

Esto podemos configurarlo, añadiendo al Job de nuestro manifiesto el campo

.spec.activeDeadlineSeconds

El campo del `deadLine` tiene relación con el tiempo en el que se aplica la duración del trabajo, sin importar el número de Pods que se creen.

Una vez que el Job alcanza el valor de tiempo transcurrido especificado en `activeDeadlineSeconds`, todos los Pods que hubiera en ejecución finalizan y el estado del Job indicará **Failed with reason: DeadlineExceeded**

Debemos de tener en cuenta que el parámetro `.spec.activeDeadlineSeconds` de un Job, tiene prioridad sobre la definición de `.spec.backoffLimit`.



Esto significa que un Job que está reintentando uno o más Pods no desplegará Pods adicionales una vez que alcance el límite de tiempo especificado por `activeDeadlineSeconds`, incluso, aunque no hayamos sobrepasado el valor de reintentos especificados en `.spec.backoffLimit`

28.6. Limpieza automática de Jobs finalizados

Los Jobs finalizados, normalmente ya no son necesarios en el sistema.

Mantenerlos en el sistema ejercerá presión sobre el Kube API Server, ya que será recursos que que tendrá que mantener a pesar de no estar usándose.

Una forma de eliminar los Jobs que hayan finalizado (Complete o Failed), de forma automática, podría ser especificando lo que se conoce el TTL (Time To Live).

La configuración la llevamos a cabo mediante el parámetro `.spec.ttlSecondsAfterFinished`, está expresando en segundos, y corresponderán al tiempo máximo que kubernetes dejará ese elemento antes de ser eliminado de forma automática.

Si indicamos como configuración al parámetro `.spec.ttlSecondsAfterFinished` 0, estaremos indicando que en cuanto el Job sea finalizado, sin tiempo de espera de cortesía, este sea eliminado (conjuntamente con los Pods asociados)



Cuando el controlador del TTL se active para limpiar el Job, el Job se eliminará en cascada, esto significa que se eliminarán los objetos asociados dependientes, como los Pods, junto con el propio Job.

Cuando se elimina un Job se respetarán de forma ordenada los correspondientes ciclos de vida de los Pods.



Actualmente según nos indica Kubernetes, esta funcionalidad de auto-limpiado de los Jobs, fue implementada a partir de la versión 1.12, y que actualmente se encuentra en estado **Alpha**, por lo que es probable que todavía quizás no funcione del todo bien en todos los casos.

Capítulo 29. Lab Jobs

Mediante este laboratorio, vamos a implementar algunos casos de uso con los Jobs de Kubernetes

29.1. Creando un Job que calcula 2000 dígitos del número Pi

Vamos a crear un Job que calcule los 2000 primero dígitos del número pi.

Vamos a crear un archivo con nombre **job-calculate-pi.yml** y le agregamos el siguiente contenido:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: job-calculate-pi
spec:
  template:
    spec:
      containers:
        - name: pi
          image: perl
          command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]
```

- **restartPolicy: Never**

- Indica que el Job no debe de ser reiniciado en caso de fallo

- **backoffLimit: 4**

- Indica que antes de dar el Job por fallido, intentará llevar a cabo su ejecución como máximo 4 veces

Ejecutamos la creación del job:

```
$ kubectl apply -f job-calculate-pi.yml
job.batch/pi created
```

A continuación, vamos a listar los Jobs que el cluster tiene operativos:

```
$ kubectl get jobs
NAME  COMPLETIONS DURATION  AGE
pi    0/1        77s       77s
```

Y también listamos los pods:



```
$ kubectl get pods
```

NAME	READY	STATUS	RESTART
pi-xjkk8	0/1	ContainerCreating	0

Dejamos transcurrir un minuto aproximadamente, y volvemos a listar los Pods del sistema:

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTART	AGE
pi-xjkk8	0/1	Completed	0	3m26s

De forma análoga, visualizamos los Jobs del clúster para ver en qué estado se encuentran:

```
$ kubectl get jobs
```

NAME	COMPLETIONS	DURATION	AGE
pi	1/1	3m16s	5m12s

- Observamos que el job indica que ha sido completado

Por último, visualizamos los logs del contenedor del Pod, para observar los dígitos calculados del número Pi:

```
$ kubectl logs pi-xjkk8
```

```
3.14159265...
```

29.2. Creando un Job que calcula 2000 dígitos del número Pi (Número de reintentos + fallo provocado)

En esta ocasión, vamos a ejecutar de nuevo un proceso que calcule los 2.000 primeros dígitos del número PI, pero vamos a especificar un número de reintentos máximo antes de dar el Job por finalizado.

Provocaremos un cambio en el script de ejecución, para que el propio Pod al ejecutarse, tenga un fallo en el command.

Vamos a crear un archivo con nombre **job-calculate-backofflimit-error-pi.yml** y le agregamos el siguiente contenido:

```

apiVersion: batch/v1
kind: Job
metadata:
  name: job-calculate-backofflimit-error-pi
spec:
  template:
    spec:
      containers:
        - name: pi
          image: perl
          command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)AAA"]
          restartPolicy: Never
    backoffLimit: 4

```

Ejecutamos la creación del job:

```

$ kubectl apply -f job-calculate-backofflimit-error-pi.yml
job.batch/job-calculate-backofflimit-error-pi created

```

Consultamos los Jobs presentes en el sistema:

```

$ kubectl get jobs
NAME           COMPLETIONS   DURATION   AGE
job-calculate-backofflimit-error-pi   0/1       4m24s   4m24s

```

Describimos el estado del Job:

```

$ kubectl describe job job-calculate-backofflimit-error-pi
...
Events:
  Type  Reason     Age   From     Message
  ----  ----     --   --     --
  Normal SuccessfulCreate 6m12s  job-controller  Created pod: job-calculate-backofflimit-error-pi-h77ch
  Normal SuccessfulCreate 6m6s   job-controller  Created pod: job-calculate-backofflimit-error-pi-wlsgs
  Normal SuccessfulCreate 5m56s  job-controller  Created pod: job-calculate-backofflimit-error-pi-kvfnz
  Warning BackoffLimitExceeded 5m16s  job-controller  Job has reached the specified backoff limit

```

- Observamos que el comando describe del Job, nos indica que se ha alcanzado el número máximo de reintentos

Por último, listamos los Pods que se encuentren en el sistema:

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
job-calculate-backofflimit-error-pi-h77ch	0/1	Error	0	12m
job-calculate-backofflimit-error-pi-kvfnz	0/1	Error	0	11m
job-calculate-backofflimit-error-pi-wlsgs	0/1	Error	0	12m

29.3. Creando Jobs en paralelo (Cantidad de ejecuciones)

Ahora, nuestro objetivo será poner en marcha el Job, e indicar que para dar el Job como finalizado, este tiene que ejecutarse 3 veces con éxito.

En este caso, el paralelismo lo dejamos a 1

Vamos a crear un archivo con nombre **job-calculate-with-completions-pi.yml** y le agregamos el siguiente contenido:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: job-calculate-with-completions-pi
spec:
  template:
    spec:
      containers:
        - name: pi
          image: perl
          command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: Never
    backoffLimit: 4
    completions: 3
```

- **backoffLimit: 4**
 - Indicamos que como máximo, haya 4 reintentos en llevar a cabo el proceso, antes de dar el Job como fallido
- **completions: 3**
 - Indicamos que para que el Job se considere como finalizado, debe de realizarse el proceso 3 veces con éxito

Ejecutamos la creación del job:

```
$ kubectl apply -f job-calculate-with-completions-pi.yml
```

```
job.batch/job-calculate-with-completions-pi created
```

Consultamos los Jobs presentes en el sistema:

```
$ kubectl get jobs
```

NAME	COMPLETIONS	DURATION	AGE
job-calculate-with-completions-pi	3/3	99s	106s

Por último, listamos los Pods que se encuentren en el sistema:

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
job-calculate-with-completions-pi-dgjtx	0/1	Completed	0	3m7s
job-calculate-with-completions-pi-mgs2j	0/1	Completed	0	2m33s
job-calculate-with-completions-pi-q4ttm	0/1	Completed	0	2m

29.4. Creando Jobs en paralelo (Cantidad de ejecuciones + Cantidad de paralelismo)

En este ejercicio, nuestro objetivo será poner en marcha el Job, e indicar que para dar el Job como finalizado, este tiene que ejecutarse 3 veces con éxito y que de forma paralela habrá 2 Pods en ejecución.

Vamos a crear un archivo con nombre **job-calculate-with-completions-and-parallelism-pi.yml** y le agregamos el siguiente contenido:

```

apiVersion: batch/v1
kind: Job
metadata:
  name: job-calculate-with-completions-and-parallelism-pi
spec:
  template:
    spec:
      containers:
        - name: pi
          image: perl
          command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]
          restartPolicy: Never
    backoffLimit: 4
    completions: 3
    parallelism: 2

```

- **backoffLimit: 4**

- Indicamos que como máximo, haya 4 reintentos en llevar a cabo el proceso, antes de dar el Job como fallido

- **completions: 3**

- Indicamos que para que el Job se considere como finalizado, debe de realizarse el proceso 3 veces con éxito

- **parallelism: 2**

- Indicamos que el Job ejecutará de forma paralela hasta 2 Pods para completar el Job

Ejecutamos la creación del job:

```
$ kubectl apply -f job-calculate-with-completions-and-parallelism-pi.yml
job.batch/job-calculate-with-completions-and-parallelism-pi created
```

Listamos los Pods que se encuentren en el sistema:

```
$ kubectl get pods
NAME                               READY   STATUS    RESTART AGE
job-calculate-with-completions-and-parallelism-pi-fv55r  1/1    Running   0      54s
job-calculate-with-completions-and-parallelism-pi-plj8r  1/1    Running   0      54s
```

- Observamos como en un momento determinado, de forma simultánea, se ejecutan 2 Pods del mismo Job, en lugar de como en el caso anterior, ejecutarse de uno en uno, hasta completar las 3 ejecuciones necesarias que se han indicado

29.5. Creando Jobs (Modo de operación, cola de procesos)

Ahora, vamos a hacer uso del Job, como si de una cola de procesos se tratara.

Vamos a crear un archivo con nombre **job-calculate-as-work-queue-pi.yml** y le agregamos el siguiente contenido:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: job-calculate-as-work-queue-pi
spec:
  template:
    spec:
      containers:
        - name: pi
          image: perl
          command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: Never
  backoffLimit: 4
  parallelism: 10
```



Fondos Europeos

MINISTERIO DE EDUCACIÓN, FORMACIÓN PROFESIONAL Y DEPORTE

Cofinanciado por la Unión Europea



- **backoffLimit: 4**
 - 4 reintentos como máximo antes de dar el Job por fallido
- **parallelism: 10**
 - En paralelo, deseamos 10 Pod de manera simultánea

Ejecutamos la creación del job:

```
$ kubectl apply -f job-calculate-as-work-queue-pi.yml
job.batch/job-calculate-as-work-queue-pi created
```

Consultamos los Jobs presentes en el sistema:

```
$ kubectl get jobs
NAME              COMPLETIONS  DURATION   AGE
job-calculate-as-work-queue-pi  0/1 of 10  98s       99s
```

Por último, listamos los Pods que se encuentren en el sistema:

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTART	AGE
job-calculate-with-completions-pi-dgjtx	1/1	Running	0	101s
job-calculate-with-completions-pi-asjtx	1/1	Running	0	101s
job-calculate-with-completions-pi-dgbdx	1/1	Running	0	101s
job-calculate-with-completions-pi-dbdtx	1/1	Running	0	101s
job-calculate-with-completions-pi-agt3x	1/1	Running	0	101s
job-calculate-with-completions-pi-jhjtx	1/1	Running	0	101s
job-calculate-with-completions-pi-g89gx	1/1	Running	0	101s
job-calculate-with-completions-pi-8xts3	1/1	Running	0	101s
job-calculate-with-completions-pi-6438x	1/1	Running	0	101s
job-calculate-with-completions-pi-xgs3g	1/1	Running	0	101s

Si volvemos a realizar sendas consultas unos minutos después, obtendríamos resultados similares a estos:

```
$ kubectl get jobs
```

NAME	COMPLETIONS	DURATION	AGE
job-calculate-as-work-queue-pi	10/1 of 10	3m42s	5m46s

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTART	AGE
job-calculate-with-completions-pi-dgjtx	0/1	Completed	0	5m50s
job-calculate-with-completions-pi-asjtx	0/1	Completed	0	5m50s
job-calculate-with-completions-pi-dgbdx	0/1	Completed	0	5m50s
job-calculate-with-completions-pi-dbdtx	0/1	Completed	0	5m50s
job-calculate-with-completions-pi-agt3x	0/1	Completed	0	5m50s
job-calculate-with-completions-pi-jhjtx	0/1	Completed	0	5m50s
job-calculate-with-completions-pi-g89gx	0/1	Completed	0	5m50s
job-calculate-with-completions-pi-8xts3	0/1	Completed	0	5m50s
job-calculate-with-completions-pi-6438x	0/1	Completed	0	5m50s
job-calculate-with-completions-pi-xgs3g	0/1	Completed	0	5m50s

29.6. Creando Jobs (Autolimpieza de Job finalizado)

Mediante el siguiente ejercicio, vamos a poner en marcha la auto-limpieza de Jobs que han finalizado su ejecución.

Vamos a crear un archivo con nombre **job-calculate-autoclean-pi.yml** y le agregamos el siguiente contenido:

```

apiVersion: batch/v1
kind: Job
metadata:
  name: job-calculate-autoclean-pi
spec:
  template:
    spec:
      containers:
        - name: pi
          image: perl
          command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]
          restartPolicy: Never
    backoffLimit: 4
    parallelism: 10
    ttlSecondsAfterFinished: 60

```

- **ttlSecondsAfterFinished: 60**

- Indicamos en cuanto al tiempo de limpieza, que se produzca 60 segundos una vez finalizado el Job

Ejecutamos la creación del job:

```
$ kubectl apply -f job-calculate-autoclean-pi.yml
job.batch/job-calculate-autoclean-pi created
```

Capítulo 30. Lab: Kube-Scheduler

Mediante este laboratorio, vamos a poner en práctica la utilización de varios Schedulers, para poder agendar diferentes Pods en cada uno de ellos con reglas específicas de operación

Podemos utilizar el kube-scheduler que viene por defecto, el nuestro específico, o una combinación de ambos, de forma simultánea

30.1. Creando nuestro scheduler personalizado

Vamos a crear el archivo **kube-scheduler-custom.yml** y le agregamos el siguiente contenido:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: my-scheduler
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: my-scheduler-as-kube-scheduler
subjects:
- kind: ServiceAccount
  name: my-scheduler
  namespace: kube-system
roleRef:
  kind: ClusterRole
  name: system:kube-scheduler
  apiGroup: rbac.authorization.k8s.io
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    component: scheduler
    tier: control-plane
  name: my-scheduler
  namespace: kube-system
spec:
  selector:
    matchLabels:
      component: scheduler
      tier: control-plane
  replicas: 1
  template:
    metadata:
      labels:
        component: scheduler
```

```

tier: control-plane
version: second
spec:
  serviceAccountName: my-scheduler
  containers:
    - command:
        - /usr/local/bin/kube-scheduler
        - --address=0.0.0.0
        - --leader-elect=false
        - --scheduler-name=my-scheduler
      image: chadmcowell/custom-scheduler
      livenessProbe:
        httpGet:
          path: /healthz
          port: 10251
        initialDelaySeconds: 15
      name: kube-second-scheduler
      readinessProbe:
        httpGet:
          path: /healthz
          port: 10251
      resources:
        requests:
          cpu: '0.1'
      securityContext:
        privileged: false
      volumeMounts: []
    hostNetwork: false
    hostPID: false
    volumes: []

```

30.2. Creando el ClusterRole

A continuación, vamos a crear un ClusterRole

Vamos a crear un archivo con nombre **kube-scheduler-custom-cluster-role.yml** y le agregamos el siguiente contenido:

```

apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRole
metadata:
  name: csinodes-admin
rules:
  - apiGroups: ["storage.k8s.io"]
    resources: ["csinodes"]
    verbs: ["get", "watch", "list"]

```

- Indicamos como regla de grupo de recursos **storage.k8s.io**

- Indicamos como binding a recursos el tipo **csinodes**
- Indicamos que el rol de clúster, tiene permitidas las operaciones de la API rest de kubernetes, **get, watch y list**

Ejecutamos la creación del ClusterRole:

```
$ kubectl apply -f kube-scheduler-custom-cluster-role.yml
clusterrole.rbac.authorization.k8s.io/csinodes-admin created
```

30.3. Creando el ClusterRoleBinding

A continuación, vamos a crear un ClusterRoleBinding, para poder bindear ese rol de cluster a un espacio de nombres concreto

Vamos a crear un archivo con nombre **kube-scheduler-custom-cluster-role-binding.yml** y le agregamos el siguiente contenido:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: read-csinodes-global
subjects:
- kind: ServiceAccount
  name: my-scheduler
  namespace: kube-system
roleRef:
  kind: ClusterRole
  name: csinodes-admin
  apiGroup: rbac.authorization.k8s.io
```

- Como nombre del binding, hemos indicado **read-csinodes-global**
- El tipo de binding es **ServiceAccount**
- El ámbito de operatividad del binding, el namespace **kube-system**
- Hacemos referencia al ClusterRole que previamente hemos indicado, mediante el elemento **roleRef**

Ejecutamos la creación del binding con el ClusterRole:

```
$ kubectl apply -f kube-scheduler-custom-cluster-role-binding.yml
clusterrolebinding.rbac.authorization.k8s.io/read-csinodes-global created
```

30.4. Creando el Rol

Ahora, toca crear el propio Rol de operación

Creamos un archivo con nombre **kube-scheduler-custom-role.yml** y le agregamos el siguiente contenido:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: system:serviceaccount:kube-system:my-scheduler
  namespace: kube-system
rules:
- apiGroups:
  - storage.k8s.io
resources:
- csinodes
verbs:
- get
- list
- watch
```

Ejecutamos la creación del rol:

```
$ kubectl apply -f kube-scheduler-custom-role.yml
role.rbac.authorization.k8s.io/system:serviceaccount:kube-system:my-scheduler created
```

30.5. Creando el RoleBinding

Finalmente, tenemos que indicar el binding que se va a producir con el rol

Creamos el archivo con nombre **kube-scheduler-custom-role-binding.yml** y le agregamos el siguiente contenido:



```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-csinodes
  namespace: kube-system
subjects:
- kind: User
  name: kubernetes-admin
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: system:serviceaccount:kube-system:my-scheduler
  apiGroup: rbac.authorization.k8s.io
```

Ejecutamos la creación del binding del rol:

```
$ kubectl apply -f kube-scheduler-custom-role-binding.yml
rolebinding.rbac.authorization.k8s.io/read-csinodes created
```

30.6. Editando el kube-scheduler, activando nuestro scheduler

Ahora, toca editar la configuración del kube-scheduler y añadir cierta configuración:

Ejecutamos el siguiente comando:

```
$ kubectl edit clusterrole system:kube-scheduler
```

Agregamos al final de todo esta sección completa:

```
- apiGroups:
  - ""
    resourceNames:
      - kube-scheduler
      - my-scheduler
    resources:
      - endpoints
    verbs:
      - delete
      - get
      - patch
      - update
  - apiGroups:
    - storage.k8s.io
    resources:
      - storageclasses
    verbs:
      - watch
      - list
      - get
```

Y por último, ya si podemos ejecutar la creación de nuestro scheduler personalizado

Ejecutamos en la consola la creación:

```
$ kubectl apply -f kube-scheduler-custom.yml

serviceaccount/my-scheduler created
clusterrolebinding.rbac.authorization.k8s.io/my-scheduler-as-kube-scheduler created
deployment.apps/my-scheduler created
```

30.7. Comprobando el scheduler

Para comprobar que nuestro scheduler se ha creado correctamente, vamos a listar los Pods que se encuentren en ejecución, en el espacio de nombres kube-system

Ejecutamos el siguiente comando:

```
$ kubectl get pods --namespace kube-system
```

NAME	READY	STATUS	RESTARTS	AGE
coredns-6955765f44-bzz62	1/1	Running	0	4d2h
coredns-6955765f44-klj6b	1/1	Running	2	22d
etcd-kubemaster	1/1	Running	2	22d
kube-apiserver-kubemaster	1/1	Running	3	22d
kube-controller-manager-kubemaster	1/1	Running	9	22d
kube-proxy-bhf4v	1/1	Running	2	22d
kube-proxy-k7wgh	1/1	Running	0	47h
kube-proxy-q24bh	1/1	Running	4	22d
kube-scheduler-kubemaster	1/1	Running	9	22d
my-scheduler-789cb54bc9-szjv7	1/1	Running	0	48s
weave-net-52qlp	2/2	Running	6	22d
weave-net-gm5sg	2/2	Running	9	22d
weave-net-hs5qq	2/2	Running	0	47h

- Observamos el scheduler por defecto, con el nombre **kube-scheduler-kubemaster**
- Observamos nuestro scheduler, con el nombre **my-scheduler-789cb54bc9-szjv7**

30.8. Asignando pods a scheduler

Ahora, vamos a crear 3 Pods, para que veamos cómo se comportan los agendadores en cada caso:

- El primer pod, no tendrá indicado de forma explícita la asignación a ningún scheduler
- El segundo pod, tendrá explícitamente asignado el scheduler **default-scheduler**
- El tercer pod, tendrá explícitamente asignado el scheduler **my-scheduler**

Vamos a crear un archivo con nombre **kube-scheduler-custom-select-by-pod.yml** y le agregamos el siguiente contenido:

```

apiVersion: v1
kind: Pod
metadata:
  name: no-annotation
  labels:
    name: multischeduler-example
spec:
  containers:
    - name: pod-with-no-annotation-container
      image: k8s.gcr.io/pause:2.0
---
apiVersion: v1
kind: Pod
metadata:
  name: annotation-default-scheduler
  labels:
    name: multischeduler-example
spec:
  schedulerName: default-scheduler
  containers:
    - name: pod-with-default-annotation-container
      image: k8s.gcr.io/pause:2.0
---
apiVersion: v1
kind: Pod
metadata:
  name: annotation-second-scheduler
  labels:
    name: multischeduler-example
spec:
  schedulerName: my-scheduler
  containers:
    - name: pod-with-second-annotation-container
      image: k8s.gcr.io/pause:2.0

```

Ejecutamos la creación de los pods:

```
$ kubectl apply -f kube-scheduler-custom-select-by-pod.yml
```

```
pod/no-annotation created
pod/annotation-default-scheduler created
pod/annotation-second-scheduler created
```

Finalmente, comprobamos que tenemos en operación los 3 Pods:

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
annotation-default-scheduler	1/1	Running	0	10s
annotation-second-scheduler	1/1	Running	0	10s
no-annotation	1/1	Running	0	10s

- El Pod que no tenía anotación de scheduler, lo ha puesto en marcha el scheduler por defecto

30.9. Visualizando eventos (A nivel de Pod del scheduler)

Otra operación interesante, es visualizar los eventos que el scheduler pudiera tener, para llevar a cabo tareas con propósito de detección de errores

Vamos a visualizar los eventos a nivel del Pod del scheduler por defecto del sistema (está en el espacio de nombres kube-system):

```
$ kubectl describe pod kube-scheduler-kubemaster --namespace kube-system  
...  
Events: <none>
```

Abajo del todo, observaremos el bloque Events, si todo va bien y no hay ningún error, observamos <none>

Mostraría errores en el planificador del tipo por ejemplo, imposibilidad de asignar Pods a un nodo, etc.

30.10. Visualizando eventos (A nivel de event log de kubernetes)

Otra opción que tenemos para visualizar eventos, es mediante la ejecución del comando events, directamente con la herramienta kubectl:

```
$ kubectl get events  
LAST SEEN TYPE REASON OBJECT MESSAGE  
<unknown> Normal Scheduled pod/daemonset-monitor-sshd-disk-fzhcq Successfully assigned default/daemonset-  
monitor-sshd-disk-fzhcq to kubeminion2  
20m Normal Pulling pod/daemonset-monitor-sshd-disk-fzhcq Pulling image "li...  
...
```

El conjunto de eventos que observamos es referente al namespace que tenemos ahora mismo, que por defecto es default

Podemos visualizar los eventos de un namespace en concreto, por ejemplo del kube-system:

```
$ kubectl get events --namespace kube-system

LAST SEEN   TYPE      REASON          OBJECT            MESSAGE
42m        Normal    Killing         pod/kube-proxy-bhf4v  Stopping container kube-proxy
<unknown>  Normal    Scheduled       pod/kube-proxy-gg55k  Successfully assigned kube-system/kube-proxy-gg55k to
kubeminion2
41m        Normal    Pulled         pod/kube-proxy-gg55k  Container image "k8s.gcr.io/kube-proxy:v1.17.0" already
present on machine
...
...
```

30.11. Visualizando eventos (A nivel de log del pod del scheduler)

Y otra opción que también tenemos, es la visualización de eventos a modo del Log del propio Pod del scheduler

```
$ kubectl logs kube-scheduler-kubemaster --namespace kube-system

I0103 16:50:02.807653    1 serving.go:312] Generated self-signed cert in-memory
W0103 16:50:04.457410    1 configmap_cafile_content.go:102] unable to load initial CA bundle for: "client-ca::kube-
system::extension-apiserver-authentication::client-ca-file" due to: configmap "extension-apiserver-authentication" not
found
W0103 16:50:04.457619    1 configmap_cafile_content.go:102] unable to load initial CA bundle for: "client-ca::kube-
system::extension-apiserver-authentication::requestheader-client-ca-file" due to: configmap "extension-apiserver-
authentication" not found
...
...
```



Fondos Europeos



MINISTERIO
GOBIERNO DE ESPAÑA
DE EDUCACIÓN, FORMACIÓN PROFESIONAL
Y DEPORTE

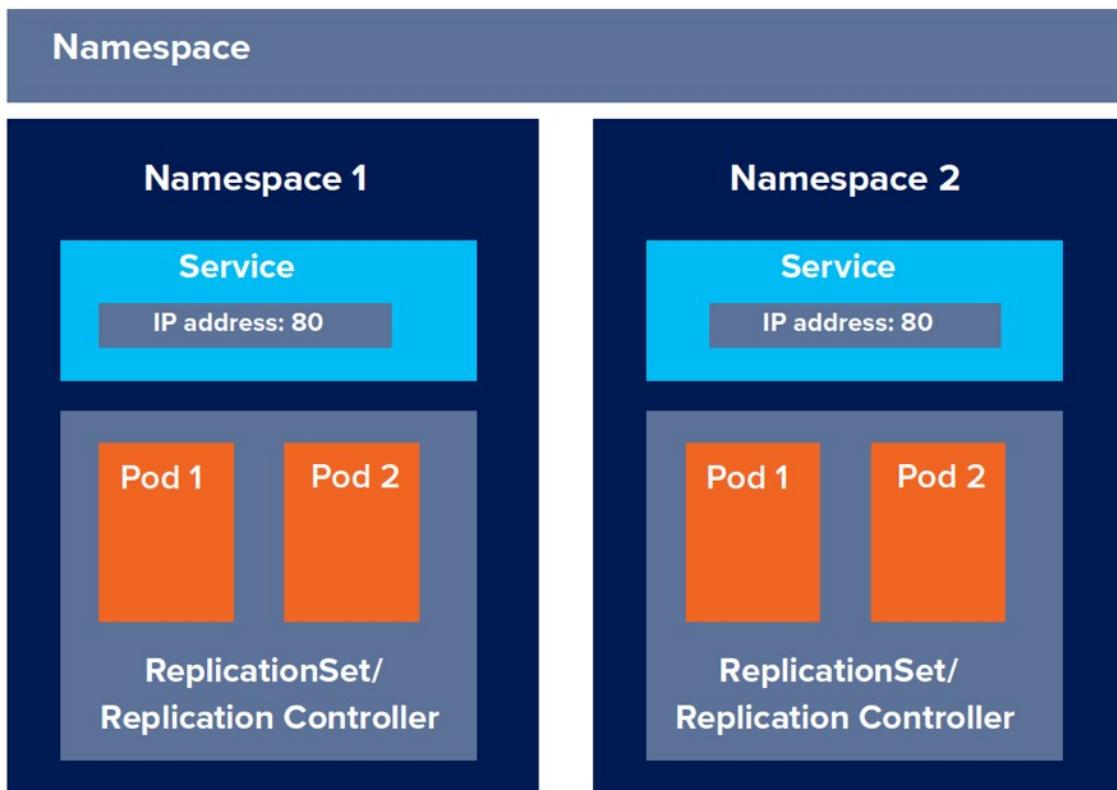
Cofinanciado por
la Unión Europea



Capítulo 31. Namespace

Kubernetes es capaz de soportar múltiples clústeres virtuales respaldados por el mismo clúster físico.

Los clústeres virtuales se denominan espacios de nombres (namespaces)



31.1. ¿Cuándo utilizar múltiples espacios de nombre?

Los espacios de nombres están pensados para utilizarse en entornos con muchos usuarios distribuidos entre múltiples equipos o proyectos.

Para clústeres con unas pocas decenas de usuarios, no deberíamos crear o pensar en espacios de nombres :)



Deberíamos empezar a usarlos solamente si necesitamos las características que estos elementos proporcionan

Los espacios de nombres proporcionan un campo de acción para los nombres

Los nombres de los recursos tienen que ser únicos dentro de cada espacio de nombres, pero no entre dichos espacios de nombres (puedo tener un mismo recurso que se llame igual, pero en diferentes espacios de nombres)

Esencialmente, consiste en una forma de dividir los recursos del clúster entre múltiples usuarios



Según Kubernetes, en futuras versiones, los objetos que se encuentren dentro de un mismo espacio de nombres tendrán las mismas políticas de control de acceso por defecto

No es necesario utilizar múltiples espacios de nombres sólo para separar recursos ligeramente diferentes, como por ejemplo versiones diferentes de la misma aplicación... Si queremos llevar a cabo esta acción, lo que **debemos de utilizar con las etiquetas para distinguir recursos dentro del mismo espacio de nombres**

31.2. Espacios de nombre por defecto

Cuando se pone en marcha kubernetes, por defecto arranca inicialmente con tres espacios de nombre:



Fondos Europeos



- **default:** El espacio de nombres por defecto para aquellos elementos que no especifican ningún espacio de nombres
- **kube-system:** El espacio de nombres para aquellos elementos creados específicamente por el sistema de kubernetes
 - kube api server, kube scheduler, etc.
- **kube-public:** El espacio de nombres kube-public se crea de forma automática y es legible por todos los usuarios (incluyendo aquellos no autenticados)
 - Este espacio de nombres se reserva principalmente para uso interno del clúster, en caso de que algunos recursos necesiten ser visibles y legibles de forma pública para todo el clúster
 - La naturaleza pública de este espacio de nombres es simplemente por convención, no es algo que tengamos que utilizar de forma indispensable

31.3. ¿Qué elementos no operan bajo un espacio de nombres?



La mayoría de los recursos de Kubernetes (Pods, Services, Replication Controllers, etc.) están en algunos espacios de nombres

Sin embargo, los recursos que representan a los propios espacios de nombres no están a su vez en espacios de nombres

Cofinanciado por la Unión Europea



De forma parecida, los recursos de bajo nivel, como los nodos y los volúmenes persistentes, tampoco están circunscritos en ningún espacio de nombres

Capítulo 32. Lab: Namespaces

Mediante este laboratorio, vamos a interactuar con los namespaces, agregando contenido e interactuando con los mismos

32.1. Listando los namespaces

Para obtener los namespaces que el clúster tiene presentes, ejecutamos el siguiente comando

```
$ kubectl get namespaces
```

NAME	STATUS	AGE
default	Active	12h
kube-node-lease	Active	12h
kube-public	Active	12h
kube-system	Active	12h



Fondos Europeos



MINISTERIO
DE EDUCACIÓN,
FORMACIÓN PROFESIONAL
Y DEPORTE

COFINANCIADO POR
LA UNIÓN EUROPEA



- Listamos todos los pods de cualquier namespace

```
$ kubectl get pods --all-namespaces
```

NAMESPACE	NAME	READY	STATUS	RESTARTS	AGE
kube-system	coredns-5644d7b6d9-b8g2p	1/1	Running	0	12h
kube-system	coredns-5644d7b6d9-bmllbw	1/1	Running	0	12h
kube-system	etcd-kubemaster	1/1	Running	0	12h
kube-system	kube-apiserver-kubemaster	1/1	Running	0	12h
kube-system	kube-controller-manager-kubemaster	1/1	Running	0	12h
kube-system	kube-proxy-24k62	1/1	Running	0	11h
kube-system	kube-proxy-j45h9	1/1	Running	0	11h
kube-system	kube-proxy-xlfrv	1/1	Running	0	12h
kube-system	kube-scheduler-kubemaster	1/1	Running	0	12h
kube-system	weave-net-6wjw8	2/2	Running	0	11h
kube-system	weave-net-9f29z	2/2	Running	0	11h
kube-system	weave-net-z94kj	2/2	Running	1	11h

- Listamos todos los servicios de cualquier namespace

```
$ kubectl get services --all-namespaces
```

NAMESPACE	NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)
default	kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP 12h
kube-system	kube-dns	ClusterIP	10.96.0.10	<none>	53/UDP, 9153/TCP 12h
kube-system	kubernetes-dashboard	ClusterIP	10.110.219.255	<none>	443/TCP 11h

- Listamos todos los replication controller de cualquier namespace

```
$ kubectl get rc --all-namespaces
```

- Listamos todos los deployments de cualquier namespace

```
$ kubectl get deployments --all-namespaces
```

NAMESPACE	NAME	READY	UP-TO-DATE	AVAILABLE	AGE
kube-system	coredns	2/2	2	2	12h

- Obtenemos información de un namespace en concreto

```
$ kubectl get namespaces kube-system
```

NAME	STATUS	AGE
kube-system	Active	12h

- Obtenemos la descripción detallada de un namespace en concreto

```
$ kubectl describe namespaces kube-system
```

Name: kube-system
Labels: <none>
Annotations: <none>
Status: Active

No resource quota.

No resource limits.

32.2. Estados de un namespace

Los namespace pueden encontrarse en dos estados

- **Active**

- Justo después de la creación, el namespace pasa a la fase de Active
- Podemos asignar contenido al espacio de nombres
- Interactuamos con normalidad con el mismo

- **Terminanting**

- El espacio de nombres está siendo eliminado, no se puede utilizar para crear nuevos elementos, como pods, rc, deployments, etc.
- El espacio de nombres enumera cada recurso conocido en ese espacio de nombres y los va eliminando uno por uno
- Una vez todos los elementos circunscritos al espacio de nombres han sido eliminados, se ejecuta una finalización de operación que elimina dicho espacio de nombres de la lista que contempla kubernetes

32.3. Creando espacios de nombres

- Mediante archivo de manifiesto

Creamos el archivo **first-custom-namespace.yml** con el siguiente contenido:

```
apiVersion: v1
kind: Namespace
metadata:
  name: my-first-namespace
```

Ejecutamos la creación

```
$ kubectl apply -f first-custom-namespace.yml
namespace/my-first-namespace created
```

- Mediante instrucción en línea

Ejecutamos la creación de un segundo namespace

```
$ kubectl create namespace second-custom-namespace
namespace/second-custom-namespace created
```

32.4. Desplegando un pod en un espacio de nombres

Una vez creados los espacios de nombres, vamos a desplegar un pod en el espacio de nombres **my-first-namespace**

- Mediante archivo de manifiesto

Creamos el archivo **first-pod-in-namespace.yml**

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod-inside-namespace
  namespace: my-first-namespace
spec:
  containers:
    - name: nginx-my-pod-inside-namespace
      image: nginx:latest
      ports:
        - containerPort: 80
```

Ejecutamos la creación

```
$ kubectl apply -f first-pod-in-namespace.yml
pod/my-pod-inside-namespace created
```

Podríamos también haber creado el Pod mediante lo que se conoce instrucción en línea, pero no lo vamos a hacer, simplemente lo indicamos a modo de información, ya que lo suyo sería que todo estuviera apuntado en un .yaml:

```
$ kubectl run --generator=run-pod/v1 nginx --namespace=my-first-namespace --image
=nginx:latest --port=8080
```

A continuación, listamos los pods dentro del namespace **my-first-namespace**

```
$ kubectl get pods --namespace my-first-namespace
NAME          READY STATUS RESTARTS AGE
nginx         1/1   Running  0       10m
```

32.5. Estableciendo preferencia del espacio de nombres

En ocasiones, nos puede resultar interesante indicar a kubernetes cual es el espacio de nombres por defecto que queremos utilizar, esto se llama indicar el contexto de operación

Le indicamos a kubernetes, que cuando ejecutemos cualquier comando de kubectl, se sitúe por defecto en el espacio de nombres **my-first-namespace**

```
$ kubectl config set-context --current --namespace=my-first-namespace
```

```
Context "kubernetes-admin@kubernetes" modified.
```

Para consultar el contexto del namespace en el que nos encontramos, ejecutamos la siguiente instrucción

```
$ kubectl config view
```

```
apiVersion: v1
clusters:
- cluster:
  certificate-authority-data: DATA+OMITTED
  server: https://192.168.15.100:6443
  name: kubernetes
contexts:
- context:
  cluster: kubernetes
  *namespace: my-first-namespace*
  user: kubernetes-admin
  name: kubernetes-admin@kubernetes
current-context: kubernetes-admin@kubernetes
kind: Config
preferences: {}
users:
- name: kubernetes-admin
  user:
    client-certificate-data: REDACTED
    client-key-data: REDACTED
```

También podemos extraer únicamente el elemento que nos interesa, ejecutamos la instrucción

```
$ kubectl config view | grep namespace:
```

32.6. Eliminando espacio de nombres

Ahora, vamos a proceder a eliminar el espacio de nombres que no tenía ningún elemento asociado

```
$ kubectl delete namespace second-custom-namespace
```

A continuación, eliminamos el namespace que tenía asociado un pod

```
$ kubectl delete namespace my-first-namespace
```

Listamos de nuevo todos los pods, y observaremos como al haber eliminado el espacio de nombres, todos los recursos que estuvieran circunscritos al namespace, también son eliminados

```
$ kubectl get pods --all-namespaces
```

32.7. Comprobando qué elementos están bajo un espacio de nombres

Para comprobar qué elementos se encuentran bajo un espacio de nombres, aparecería algo parecido a lo siguiente

```
$ kubectl api-resources --namespaced=true
```

NAME	SHORTNAMES	APIGROUP	NAMESPACED	KIND
bindings			true	
Binding				
configmaps	cm		true	
ConfigMap				
endpoints	ep		true	
Endpoints				
events	ev		true	
Event				
limitranges	limits		true	
LimitRange				
persistentvolumeclaims	pvc		true	
PersistentVolumeClaim				
pods	po		true	Pod
PodTemplate			true	
replicationcontrollers	rc		true	
ReplicationController				
resourcequotas	quota		true	
ResourceQuota				
secrets			true	
Secret				

serviceaccounts	sa		true
ServiceAccount			
services	svc		true
Service			
controllerrevisions		apps	true
ControllerRevision			
daemonsets	ds	apps	true
DaemonSet			
deployments	deploy	apps	true
Deployment			
replicasesets	rs	apps	true
ReplicaSet			
statefulsets	sts	apps	true
StatefulSet			
localsubjectaccessreviews		authorization.k8s.io	true
LocalSubjectAccessReview			
horizontalpodautoscalers	hpa	autoscaling	true
HorizontalPodAutoscaler			
cronjobs	cj	batch	true
CronJob			
jobs		batch	true
Lease		coordination.k8s.io	true
leases			
Event	ev	events.k8s.io	true
ingresses	ing	extensions	true
Ingress			
ingresses	ing	networking.k8s.io	true
Ingress			
networkpolicies	netpol	networking.k8s.io	true
NetworkPolicy			
poddisruptionbudgets	pdb	policy	true
PodDisruptionBudget			
rolebindings		rbac.authorization.k8s.io	true
RoleBinding			
roles		rbac.authorization.k8s.io	true
			Role

Por otro lado, comprobamos qué elementos no se encuentran circunscritos a un espacio de nombres, aparecería algo parecido a lo siguiente



```
$ kubectl api-resources --namespaced=false
```

NAME	SHORTNAMES	APIGROUP	
NAMESPACED	KIND		
componentstatuses	cs		false
ComponentStatus			
namespaces	ns		false
Namespace			
nodes	no		false
Node			
persistentvolumes	pv		false
PersistentVolume			
mutatingwebhookconfigurations		admissionregistration.k8s.io	false
MutatingWebhookConfiguration			
validatingwebhookconfigurations		admissionregistration.k8s.io	false
ValidatingWebhookConfiguration			
customresourcedefinitions	crd,crds	apiextensions.k8s.io	false
CustomResourceDefinition			
apiservices		apiregistration.k8s.io	false
APIService			
tokenreviews		authentication.k8s.io	false
TokenReview			
selfsubjectaccessreviews		authorization.k8s.io	false
SelfSubjectAccessReview			
selfsubjectrulesreviews		authorization.k8s.io	false
SelfSubjectRulesReview			
subjectaccessreviews		authorization.k8s.io	false
SubjectAccessReview			
certificatesigningrequests	csr	certificates.k8s.io	false
CertificateSigningRequest			
runtimeclasses		node.k8s.io	false
RuntimeClass			
podsecuritypolicies	psp	policy	false
PodSecurityPolicy			
clusterrolebindings		rbac.authorization.k8s.io	false
ClusterRoleBinding			
clusterroles		rbac.authorization.k8s.io	false
ClusterRole			
priorityclasses	pc	scheduling.k8s.io	false
PriorityClass			
csidrivers		storage.k8s.io	false
CSIDriver			
csinodes		storage.k8s.io	false
CSINode			
storageclasses	sc	storage.k8s.io	false
StorageClass			
volumeattachments		storage.k8s.io	false
VolumeAttachment			

32.8. Espacios de nombres y DNS

Cuando creamos un **Servicio**, se crea en kubernetes una entrada DNS correspondiente a dicho servicio

La entrada tiene la forma `<service-name>.<namespace-name>.svc.cluster.local`, significa que si un contenedor simplemente usa `<service-name>`, se resolverá al servicio que sea local al espacio de nombres con el que estemos trabajando por defecto

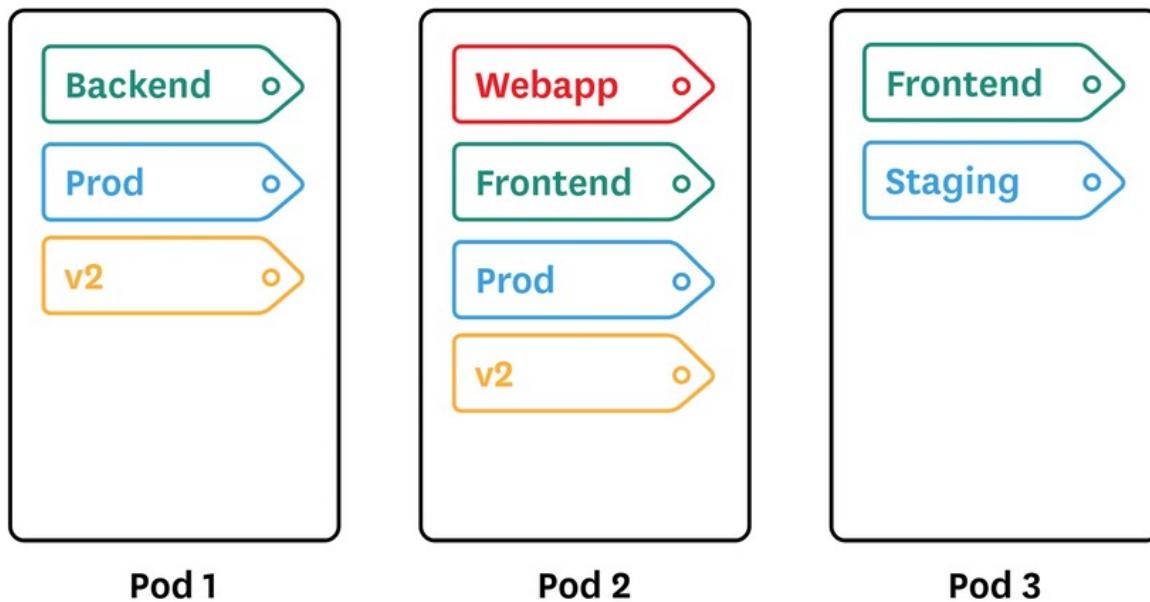
Esto resulta de utilidad para poder emplear la misma configuración entre múltiples espacios de nombres como por ejemplo si quisieramos tener espacios de nombres como "Development", "Staging", "Production", etc.

Si queremos referenciar recursos entre distintos espacios de nombres, debemos entonces utilizar el nombre cualificado completo de dominio (FQDN)

Capítulo 33. Labels

El uso de las etiquetas en Kubernetes es un mecanismo muy potente, ya que mediante ellas, el orquestador es capaz de organizar la nube de despliegue de pods que tengamos entre manos

Una etiqueta consiste en un par key-value con ciertas restricciones que debemos tener en cuenta cuando las definamos



La estructura de una etiqueta consta de opcionalmente indicar un prefijo y un nombre, separados por el carácter /

33.1. Restricciones de nombre

- Restricciones del prefijo
 - **Prefijo:**
 - Es opcional
 - No puede tener más de 253 caracteres en total
 - Está seguido del carácter /
 - Los prefijos **kubernetes.io** y **k8s.io** están reservados para el núcleo de componentes de Kubernetes y no podemos utilizarlos
- Restricciones del nombre
- **Longitud:** Como máximo deberá de tener 63 caracteres
- **Caracteres:** Comenzará y terminará con un carácter alfanumérico comprendido en el rango [a-z0-9A-Z]
 - Se permiten los guiones medios (-)
 - Se permiten los guiones bajos (_)
 - Se permiten los puntos (.)

- Se permiten caracteres alfanuméricos entre medias



Capítulo 34. Lab: Labels

Mediante este laboratorio vamos a interactuar con las etiquetas, asignando etiquetas a pods e interactuando con los mismos

34.1. Creando pods con etiquetas

Primero, vamos a crear un pod que va a contener 1 container, dicho pod tendrá asignadas dos etiquetas

Creamos el archivo **pod-with-labels.yml** e indicamos el siguiente contenido

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-labels
  labels:
    customer: C8229565
    app: frontend
spec:
  containers:
    - name: webserver
      image: nginx:latest
      ports:
        - containerPort: 80
```

Ejecutamos la creación

```
$ kubectl apply -f pod-with-labels.yml
pod/pod-with-labels created
```

34.2. Consultando las etiquetas de los pods

Ejecutamos el siguiente comando, para visualizar los pods desplegados así como la visualización de sus etiquetas

```
$ kubectl get pods --show-labels
```

34.3. Añadiendo etiquetas al pod

- Mediante archivo de manifiesto

Vamos a añadir una nueva etiqueta a nuestro pod desplegado, editamos el archivo **pod-with-**

labels.yml e indicamos el siguiente contenido

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-labels
  labels:
    customer: C8229565
    app: frontend
    version: v2
spec:
  containers:
    - name: webserver
      image: nginx:latest
      ports:
        - containerPort: 80
```

Ejecutamos la actualización de la etiqueta del pod

```
$ kubectl apply -f pod-with-labels.yml
```

```
pod/pod-with-labels configured
```

- Mediante instrucción en línea

```
$ kubectl label pods pod-with-labels compilation-date=today!
```

34.4. Filtrando pods mediante etiqueta

En ocasiones nos puede resultar interesante, el filtrado en la obtención de pods que contengan una determinada etiqueta

Vamos a filtrar pods que tengan la etiqueta **customer** con el valor **C8229565**, añadimos también el parámetro **--show-labels** para comprobar que efectivamente aparece la columna de LABELS para comprobar

```
$ kubectl get pods --selector customer=C8229565 --show-labels
```

NAME	READY	STATUS	RESTARTS	AGE	LABELS
pod-with-labels	1/1	Running	0	29m	customer=C8229565

Otra opción de filtrado que disponemos, es indicar un conjunto de valores para una etiqueta, en este caso, vamos a indicar un conjunto de valores para la etiqueta **customer**

Si cumple que el pod tiene alguno de los valores, entonces aparecerá en el listado

Vamos a indicar el valor existente, por un lado **C8229565** y por otro nos inventamos otro valor, por ejemplo **CAAA**

```
$ kubectl get pods --selector 'customer in (C8229565, CAAA)' --show-labels
```

NAME	READY	STATUS	RESTARTS	AGE	LABELS
pod-with-labels	1/1	Running	0	33m	customer=C8229565

Observamos que aparece el pod por que uno de los valores del conjunto se cumple

Por otra parte, también podríamos filtrar Pods que tuvieran de forma explícita por ejemplo 2 etiquetas con 2 valores específicos



Tiene que cumplirse la condición que los pods tienen que tener 2 etiquetas exactamente con esa key y esos values, si algo no concuerda, el pod no se visualiza

```
$ kubectl get pods -l customer=C8229565,app=frontend
```

NAME	READY	STATUS	RESTARTS	AGE
pod-with-labels	1/1	Running	0	51s

NAME	READY	STATUS	RESTARTS	AGE
pod-with-labels	1/1	Running	0	7m54s

34.5. Filtrando pods por estado de ejecución

Otro tipo de filtro que nos podría interesar utilizar, es llevar a cabo un filtro de pods mediante alguna propiedad del propio objeto desplegado en sí, en lugar de por etiquetas específicas, por ejemplo, vamos a filtrar todos los pods cuya propiedad del objeto asociado al pod **status.phase** sea igual a **Running**

Con este filtrado, obtendríamos los Pods que se encuentran en ejecución en base al valor de esa etiqueta del objeto status

```
$ kubectl get pods --field-selector status.phase=Running
```

NAME	READY	STATUS	RESTARTS	AGE	LABELS
pod-with-labels	1/1	Running	0	33m	customer=C8229565

34.6. Filtrando pods por propiedad de metadata

También podemos filtrar pods por propiedad de metadata o cualquier otra que nosotros hayamos especificado en el archivo de manifiesto, por ejemplo, filtrar pods que se encuentren desplegados en un namespace en concreto

```
$ kubectl get pods --field-selector metadata.namespace=default
```

NAME	READY	STATUS	RESTARTS	AGE	LABELS
pod-with-labels	1/1	Running	0	33m	customer=C8229565

Por otra parte, también podemos llevar a cabo una combinación de múltiples filtros, de forma que por ejemplo, podemos concatenar filtros de propiedades específicas, y de estado del objeto

```
$ kubectl get pods --field-selector metadata.namespace=default,status.phase=Running
```

NAME	READY	STATUS	RESTARTS	AGE	LABELS
pod-with-labels	1/1	Running	0	33m	customer=C8229565

Incluso podemos incluso introducir operador de negación, de forma que podríamos buscar pods, que no estuvieran en ejecución

```
$ kubectl get pods --field-selector status.phase!=Running
```

```
No resources found in default namespace.
```

- En el momento de lanzar el comando todos los pods están en ejecución Running, por eso la consulta no obtiene ningún resultado

34.7. Obteniendo logs del Pod, filtrando por etiqueta

Otra opción puede ser obtener logs de contenedores del Pod, filtrando estos por una etiqueta determinada

De esta forma, podríamos obtener traza de log unificada únicamente de Pods que cumplieran la regla del etiquetado

- Le indicamos un describe y nos quedamos por la IP interna del Pod

```
$ kubectl describe pod pod-with-labels | grep IP:
```

```
IP: 10.36.0.2
```

```
IP: 10.36.0.2
```

- Accedemos al navegador o bien ejecutamos un curl con la URL <http://10.36.0.2:80>, de esta forma se habrá generado traza de las propias peticiones http
- Por último, obtenemos traza de log pero filtrando por etiqueta
- Concretamente nos interesa la etiqueta con el key-value: **customer=C8229565**

```
$ kubectl logs -lcustomer=C8229565

...
10.32.0.1 - - [24/Sep/2019:15:22:44 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0
(X11; Linux x86_64; rv:60.0) Gecko/20100101 Firefox/60.0" "-"
2019/09/24 15:22:44 [error] 6#6: *1 open() "/usr/share/nginx/html/favicon.ico" failed
(2: No such file or directory), client: 10.32.0.1, server: localhost, request: "GET
/favicon.ico HTTP/1.1", host: "10.36.0.2"
10.32.0.1 - - [24/Sep/2019:15:22:44 +0000] "GET /favicon.ico HTTP/1.1" 404 153 "-"
"Mozilla/5.0 (X11; Linux x86_64; rv:60.0) Gecko/20100101 Firefox/60.0" "-"

...
```

También se puede combinar la visualización de log basado en una etiqueta determinada, junto con la indicación de que muestre logs de todos los contenedores si es que hubiera más de uno

```
$ kubectl logs -lcustomer=C8229565 --all-containers=true
```

34.8. Eliminando etiquetas del pod

- Mediante archivo de manifiesto

Editamos el archivo **pod-with-labels.yml** y eliminamos la etiqueta **version**

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-labels
  labels:
    customer: C8229565
    app: frontend
spec:
  containers:
    - name: webserver
      image: nginx:latest
      ports:
        - containerPort: 80
```

Ejecutamos la actualización de la etiqueta del pod

```
$ kubectl apply -f pod-with-labels.yml
pod/pod-with-labels configured
```

- Mediante instrucción en línea

Eliminamos ahora la etiqueta **app** del pod que se encuentra en ejecución **pod-with-labels**

```
$ kubectl label pods pod-with-labels app-
```

Finalmente listamos los pods, únicamente aparece el que estamos tratando, así como una única etiqueta que queda **customer**

```
$ kubectl get pods --show-labels
```

NAME	READY	STATUS	RESTARTS	AGE	LABELS
pod-with-labels	1/1	Running	0	24m	customer=C8229565

34.9. Eliminando pod mediante etiquetas

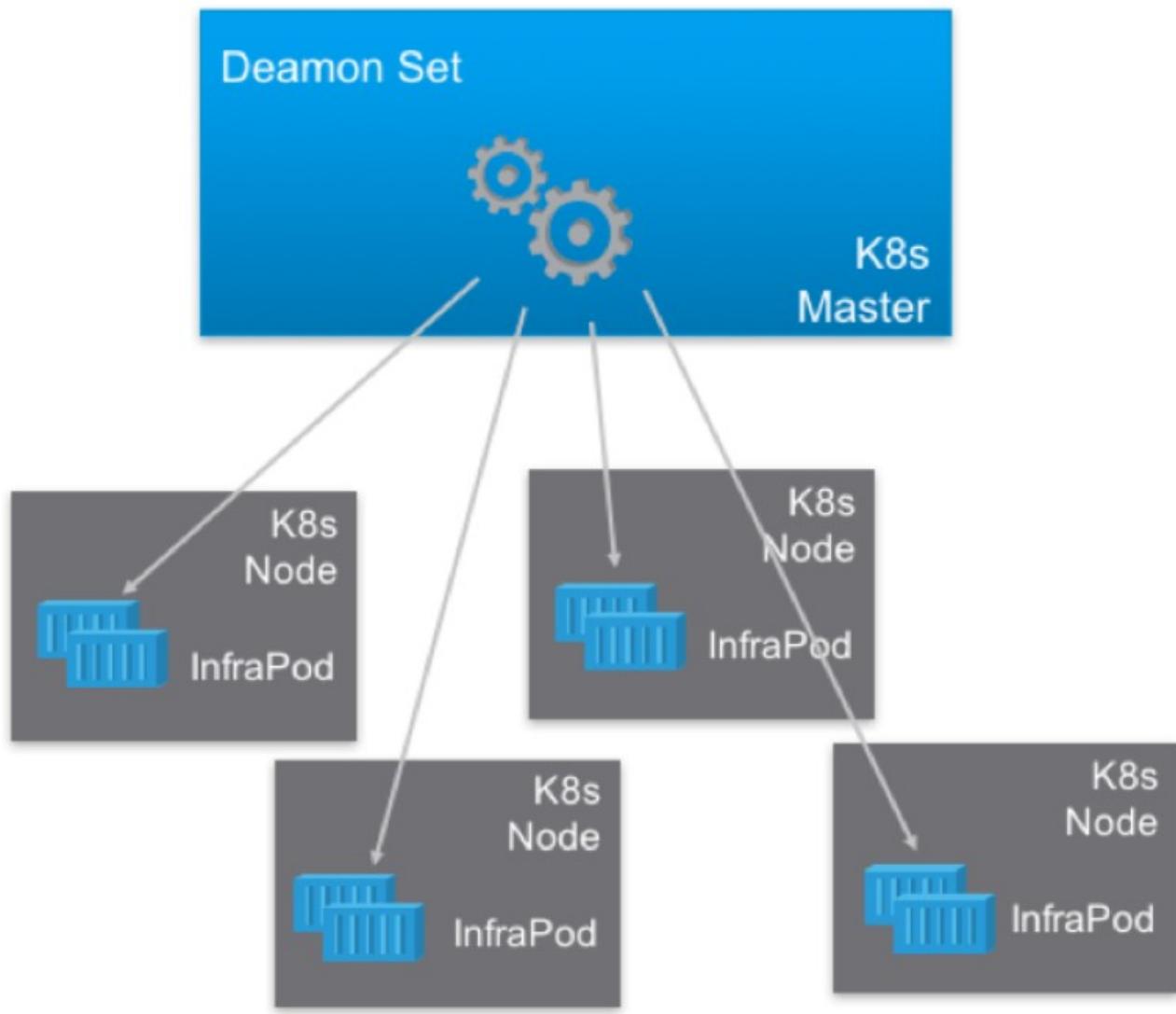
Otra opción que nos puede resultar útil, es eliminar pods en base a un conjunto de valores que tenga una etiqueta

Vamos a eliminar pods que contengan la etiqueta **customer** y cuyos valores de esa etiqueta pueden ser **C8229565** o bien **CAAA**

En nuestro caso, detectará el pod por que coincidirá uno de sus valores, concretamente el **C8229565** y este será eliminado

```
$ kubectl delete pods --selector 'customer in (C8229565, CAAA)'  
pod "pod-with-labels" deleted
```

Capítulo 35. DaemonSet



Un DaemonSet garantiza que todos (o algunos) de los nodos ejecuten una copia de un Pod.

Conforme se añade más nodos al clúster, nuevos Pods son añadidos a los mismos nodos nuevos que van entrando.

Conforme se elimina nodos del clúster, dichos Pods se destruyen.

Al eliminar un DaemonSet se limpian todos los Pods que han sido creados.

Los DaemonSets no usan un scheduler para desplegar pods.

35.1. Operativa del DaemonSet

Podemos preguntarnos, ¿Por qué no podemos asignar al scheduler programación de lanzamiento de pods en cada nodo?

Es debido a que los DaemonSets tienen un conjunto de instrucciones especiales que incluyen no sólo la ejecución de un pod en un nodo específico, sino que también contemplan casos de uso como

cuando nuevos nodos se añaden al clúster.

Automáticamente ese conjunto de instrucciones, hace también que los Pods DaemonSets se ejecuten en esos nuevos nodos que han aparecido

Si eliminamos un pod o conjunto de Pods DaemonSet, automáticamente se recrean de nuevo los Pods

Los DaemonSets ignoran cualquier mancha (taints) de los nodos que estos pudieran tener, es por esa razón, por la que observamos DaemonSets operando en el nodo máster, incluso aún teniendo este una mancha de no ejecución de Pods por parte de un scheduler **NoSchedule**



Capítulo 36. Lab: DaemonSet

Mediante este laboratorio, vamos a observar los DaemonSets que el sistema trae por defecto y vamos a realizar algunas prácticas creando nuestros propios DaemonSets

36.1. Observando los DaemonSets del namespace kube-system

Ejecutamos el siguiente comando:

```
$ kubectl get pods --namespace kube-system -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE	NOMINATED NODE
coredns-6955765f44-bzz62 <none>	1/1	Running	0	4d19h	10.46.0.4	kubeminion1	<none>
coredns-6955765f44-klj6b <none>	1/1	Running	2	23d	10.32.0.2	kubemaster	<none>
etcd-kubemaster <none>	1/1	Running	2	23d	192.168.15.100	kubemaster	<none>
kube-apiserver-kubemaster <none>	1/1	Running	3	23d	192.168.15.100	kubemaster	<none>
kube-controller-manager-kubemaster <none>	1/1	Running	9	23d	192.168.15.100	kubemaster	<none>
kube-proxy-bhf4v <none>	1/1	Running	2	23d	192.168.15.100	kubemaster	<none>
kube-proxy-k7wgh <none>	1/1	Running	0	2d16h	192.168.15.102	kubeminion2	<none>
kube-proxy-q24bh <none>	1/1	Running	4	23d	192.168.15.101	kubeminion1	<none>
kube-scheduler-kubemaster <none>	1/1	Running	9	23d	192.168.15.100	kubemaster	<none>
my-scheduler-789cb54bc9-szjv7 <none>	1/1	Running	0	16h	10.36.0.2	kubeminion2	<none>
weave-net-52qlp <none>	2/2	Running	6	23d	192.168.15.100	kubemaster	<none>
weave-net-gm5sg <none>	2/2	Running	9	23d	192.168.15.101	kubeminion1	<none>
weave-net-hs5qq <none>	2/2	Running	0	2d16h	192.168.15.102	kubeminion2	<none>

Observamos 3 pods con el prefijo **kube-proxy...**, cada uno está operando en un nodo diferente

36.2. Probando regeneración DaemonSets de sistema

Vamos a probar a eliminar todos los pods kube-proxy para ver si el sistema es capaz de volver a generarlos, situando de nuevo 1 en cada nodo, nos fijamos en el nombre de pod que tengamos cada uno para llevar a cabo la eliminación

Ejecutamos en consola:

```
$ kubectl delete pod kube-proxy-bhf4v --namespace kube-system
$ kubectl delete pod kube-proxy-k7wgh --namespace kube-system
$ kubectl delete pod kube-proxy-q24bh --namespace kube-system
```

Ahora, listamos de nuevo los pods y comprobamos si kubernetes ha sido capaz de llevar a cabo la regeneración:

```
$ kubectl get pods --namespace kube-system
```

NAME	READY	STATUS	RESTARTS	AGE
coredns-6955765f44-bzz62	1/1	Running	0	4d19h
coredns-6955765f44-klj6b	1/1	Running	2	23d
etcd-kubemaster	1/1	Running	2	23d
kube-apiserver-kubemaster	1/1	Running	3	23d
kube-controller-manager-kubemaster	1/1	Running	9	23d
kube-proxy-gg55k	1/1	Running	0	60s
kube-proxy-gnpx2	1/1	Running	0	46s
kube-proxy-jpvr8	1/1	Running	0	78s
kube-scheduler-kubemaster	1/1	Running	9	23d
my-scheduler-789cb54bc9-szjv7	1/1	Running	0	16h
weave-net-52qlp	2/2	Running	6	23d
weave-net-gm5sg	2/2	Running	9	23d
weave-net-hs5qq	2/2	Running	0	2d16h

Observamos, que efectivamente, los DaemonSet han vuelto a ser recreados

36.3. Creando un DaemonSet

Vamos a crear un DaemonSet cuyo objetivo va a ser el de monitorizar los servidores que tengan discos duros de tipo SSD

El contenedor de la imagen lo único que realiza es un while con el siguiente script

 "/bin/sh" "-c" "while true; do echo 'SSD OK'; sleep 5; done"

El fundamento de la práctica es interoperar con un DaemonSet y observar su comportamiento

Lo primero que vamos a hacer, es añadirle una etiqueta al nodo **kubeminion1** que indique el tipo de disco que tiene:

```
$ kubectl label node kubeminion1 disk=ssd
```

```
node/kubeminion1 labeled
```

Ahora, vamos a crear el archivo **daemonset-monitor-sshd-disk.yml** y le agregamos el siguiente

contenido:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: daemonset-monitor-sshd-disk
spec:
  selector:
    matchLabels:
      app: ssd-monitor
  template:
    metadata:
      labels:
        app: ssd-monitor
    spec:
      nodeSelector:
        disk: ssd
      containers:
        - name: main
          image: linuxacademycontent/ssd-monitor
```

Ejecutamos la creación del DaemonSet:

```
$ kubectl apply -f daemonset-monitor-sshd-disk.yml
daemonset.apps/daemonset-monitor-sshd-disk created
```

Al DaemonSet le hemos aplicado una política de selección de nodo, de manera que el DaemonSet sólo se ejecutará en los nodos que cumplan la etiqueta **disk:ssd**

Ahora, listamos los DaemonSet de nuestro espacio de nombres default:

```
$ kubectl get daemonsets
NAME              DESIRED  CURRENT  READY  UP-TO-DATE  AVAILABLE  NODE SELECTOR  AGE
daemonset-monitor-sshd-disk  1       1       1       1           1           disk=ssd   81s
```

También comprobamos que el Pod está ejecutándose en la máquina correcta **kubeminion1**:

```
$ kubectl get pods -o wide
NAME                  READY  STATUS  RESTARTS  AGE  IP          NODE  NOMINATED NODE
READINESS GATES
daemonset-monitor-sshd-disk-t99qq     1/1  Running  0   2m36s  10.46.0.6  kubeminion1  <none>
<none>
```

Ahora, vamos a etiquetar el nodo **kubeminion2** con la misma etiqueta que el nodo **kubeminion1**, para comprobar que en caliente, el DaemonSet es puesto en marcha también en el nodo en

cuestión:

```
$ kubectl label node kubeminion2 disk=ssd
```

```
node/kubeminion2 labeled
```

Listamos de nuevo los DaemonSets:

```
$ kubectl get daemonsets
```

NAME	READY	UP-TO-DATE	AGE	NODE SELECTOR
daemonset-monitor-sshd-disk-fzhcq	0	11s	10.36.0.5	kubeminion2 <none>
<none>	0	4m40s	10.46.0.6	kubeminion1 <none>
daemonset-monitor-sshd-disk-t99qq	0	4m40s	10.46.0.6	kubeminion1 <none>



Observamos que efectivamente, se ha creado un DaemonSet adicional en el nodo kubeminion2

Ahora, eliminamos la etiqueta disk al nodo **kubeminion1**:

```
$ kubectl label node kubeminion1 disk-
```

```
node/kubeminion1 labeled
```

Finalmente obtenemos el conjunto de Pods, y observaremos como en el nodo kubeminion1 ha desaparecido el DaemonSet, quedando sólo el Pod en ejecución en el Nodo kubeminion2:

```
$ kubectl get daemonsets
```

NAME	READY	UP-TO-DATE	AGE	NODE SELECTOR
daemonset-monitor-sshd-disk-fzhcq	0	11s	10.36.0.5	kubeminion2 <none>
<none>	0	4m40s	10.46.0.6	kubeminion1 <none>



36.4. Eliminación del DaemonSet

Si cuando hemos creado un DaemonSet, intentamos eliminar directamente los Pods, estos serán recreados de nuevo y nos dará la sensación de que no se puede borrar

Primero listamos los DaemonSets que tengamos en el sistema:

```
$ kubectl get daemonset
```

NAME	DESIRED	CURRENT	READY	UP-TO-DATE	AVAILABLE	NODE SELECTOR	AGE
daemonset-monitor-sshd-disk	1	1	1	1	1	disk=ssd	5h35m



A continuación lo eliminamos:

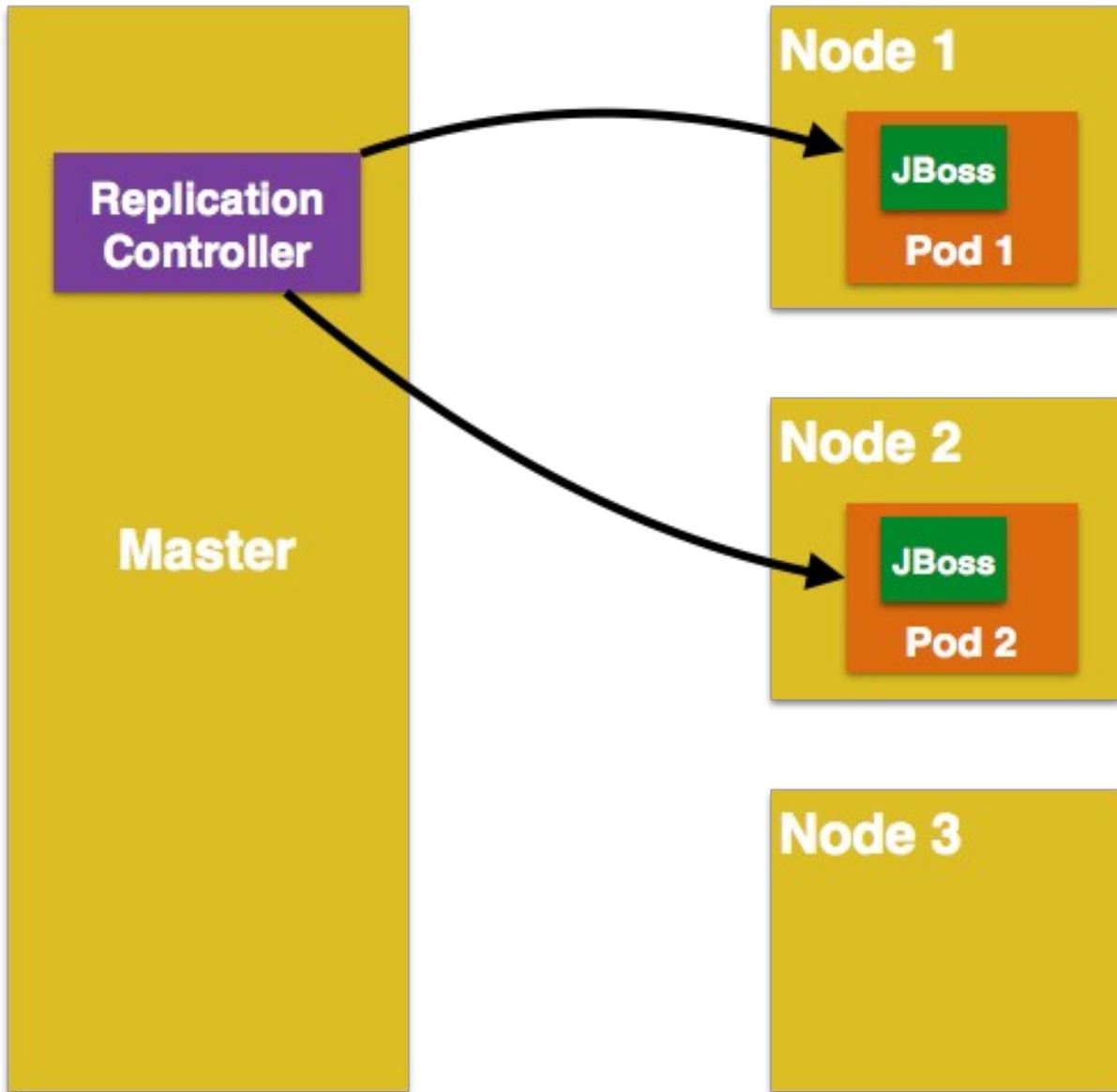
```
$ kubectl delete daemonset daemonset-monitor-sshd-disk
```

```
daemonset.apps "daemonset-monitor-sshd-disk" deleted
```



Capítulo 37. ReplicationController

Gestionar el despliegue a nivel de Pods está bien ya que tenemos un control fino de lo que estamos desplegando, pero llega un momento que se nos complica la gestión



37.1. Desventajas de uso de Pods simples

- Una vez el pod ha caído en un nodo, no se puede mover a otro nodo si este sufre algún problema
- No podemos escalar, ya que no disponemos de control de replicación, debiendo manualmente generar nuevos Pods

Para solventar estos inconvenientes, vamos a hacer uso del **Replication Controller**

37.2. Ventajas de uso

- El controlador de replicación es una unidad superior al Pod

- Permite gestionar numero de pods del mismo tipo (replicas)
- Si hay más Pods de los ordenados, baja el número
- Si hay menos Pods de los ordenados, los aumenta hasta nivelar el número de replicas que han sido ordenadas
- Los Pods son reemplazados de forma automática en caso de fallo
- Es aconsejable utilizar el Replication Controller aún estando desplegando únicamente un Pod
- El funcionamiento es parecido a un supervisor de procesos, en lugar de supervisar procesos individuales en un sólo nodo, ReplicationController es capaz de supervisar múltiples Pods en múltiples Nodos
- A menudo se abrevia como "rc" o "rcs", hay polémica con eso :D

37.3. ¿Cuándo debemos de usarlo?

No requerimos de orquestación en las actualizaciones o sencillamente son sistemas que no vamos a actualizar nunca o que raramente se daría el caso

Pero si queremos de control de replicas

Entonces si sería un buen candidato a utilizar

37.4. Etiquetas en ReplicationController

El uso de las etiquetas (labels) cuando utilizamos la unidad de despliegue ReplicationController resulta de uso fundamental

Las etiquetas van a permitir indicar a la unidad ReplicationController cuáles son los Pods que me pertenecen para gestionarlos

El secreto está en etiquetar los Pods de despliegue e indicar un selector concreto en la especificación de la unidad de despliegue

Capítulo 38. Lab: ReplicationController

38.1. Creación del primer ReplicationController

Queremos un despliegue de un servidor web nginx, compuesto de 3 réplicas en el clúster

Creamos el archivo de manifiesto **replication-controller-with-1-container.yml** e indicamos el siguiente contenido

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: replication-controller-with-1-container
spec:
  replicas: 3
  selector:
    customer: B8745
  template:
    metadata:
      labels:
        customer: B8745
    spec:
      containers:
        - name: nginx-container
          image: nginx
          ports:
            - containerPort: 80
```

Ejecutamos la instrucción de creación

```
$ kubectl apply -f replication-controller-with-1-container.yml
replicationcontroller/replication-controller-with-1-container created
```

38.2. Análisis de estructura mínima

Como observamos, el ReplicationController ha introducido nuevos elementos que pasamos a describir a continuación:

- **kind**
 - ReplicationController, indicamos la nueva unidad a utilizar
- **spec → replicas**
 - Indicamos el número de Pods que tendremos en el estado final a desplegar
- **spec → selector**

- Indicamos un selector de etiqueta, con formato key-value, esto le vale a Kubernetes en la unidad de replicación, para encontrar Pods que esta unidad tenga que gestionar

- **spec → template → metadata**

- Especificamos las etiquetas que a modo de metadata tendrán los Pods cuando se creen
- La sección del template realmente crea una plantilla, de modo que los siguientes Pods que sea necesario crear, cumplirán todos la misma estructura

- **spec → template → spec**

- Especificamos en la plantilla los contenedores que tendrá dentro el Pod, su imagen, puertos, etc.

38.3. Listando los Pods creados por el ReplicationController

Comprobamos que efectivamente se han creado 3 Pods, ya que indicamos en la metadata del .yml **replicas: 3**

```
$ kubectl get pods

NAME           READY   STATUS    RESTARTS   AGE
replication-controller-with-1-container-jw2rd  1/1     Running   0          26s
replication-controller-with-1-container-qc8z5   1/1     Running   0          26s
replication-controller-with-1-container-rbk9v   1/1     Running   0          26s
```

38.4. Listando las unidades de replicación

Ahora, pasamos a listar las unidades ReplicationController que tengamos en el sistema

El sistema debería de indicarnos que efectivamente tenemos 1 controlador de replicación, compuesto por 3 Pods, y que los 3 Pods están listos

```
$ kubectl get rc

NAME        DESIRED   CURRENT   READY   AGE
replication-controller-with-1-container  3         3         3       2m17s
```

38.5. Describiendo la unidad de replicación

Ahora, vamos a realizar un describe al ReplicationController

```
$ kubectl describe rc replication-controller-with-1-container

Name: replication-controller-with-1-container
Namespace: default
Selector: customer=B8745
Labels: customer=B8745
Annotations: kubectl.kubernetes.io/last-applied-configuration:
  {"apiVersion":"v1","kind":"ReplicationController","metadata":{ "annotations":{},"name":"replication-controller-with-1-container","namespace...}}
Replicas: 3 current / 3 desired
Pods Status: 3 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels: customer=B8745
  Containers:
    nginx-container:
      Image: nginx
      Port: 80/TCP
      Host Port: 0/TCP
      Environment: <none>
      Mounts: <none>
      Volumes: <none>
  Events:
  Type Reason Age From Message
  ---- ---- - - -
  Normal SuccessfulCreate 5m14s replication-controller Created pod: replication-controller-with-1-container-jw2rd
  Normal SuccessfulCreate 5m14s replication-controller Created pod: replication-controller-with-1-container-qc8z5
  Normal SuccessfulCreate 5m14s replication-controller Created pod: replication-controller-with-1-container-rbk9v
```

Observamos como el describe del rc nos indica interesantes datos:

- **Name**
 - Nombre del rc
- **Namespace**
 - Espacio de nombres en el que se encuentra la unidad rc
- **Selector**
 - Selector con formato key-value que utilizará el rc para detectar sus Pods, de forma que los identificará como suyos
- **Labels**
 - Etiquetas que la unidad rc tiene asignada como metadata propia
- **Replicas**
 - Indica el número de réplicas actualmente operativas / Réplicas ordenadas a disponer
- **Pods Status**

- Indica el estado de los Pods que la unidad de replicación controla, en nuestro caso nos indica que los 3 Pods se encuentran operando normalmente

- **Pod Template**

- Indica la estructura que tendrán los nuevos Pods que la unidad rc vaya a crear

- **Events**

- Histórico de eventos de la unidad de replicación
- A la unidad de replicación no le interesa mostrar el detalle de cada Pod (Si se le ha hecho pull a la imagen, en qué Nodo se ha alojado, etc)
- A la unidad de replicación le interesa mostrar si el Pod ha sido creado, su nombre, desde cuando y si está operando normalmente

38.6. Escalando el número de réplicas

Vamos a dar una orden a la unidad rc para que aumente el número de réplicas de la unidad, pasando de 3 a 5

- Mediante archivo de manifiesto

Modificamos el archivo de manifiesto **replication-controller-with-1-container.yml** e indicamos el siguiente contenido

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: replication-controller-with-1-container
spec:
  replicas: 5
  selector:
    customer: B8745
  template:
    metadata:
      labels:
        customer: B8745
    spec:
      containers:
        - name: nginx-container
          image: nginx
          ports:
            - containerPort: 80
```

Ejecutamos la instrucción de actualización

```
$ kubectl apply -f replication-controller-with-1-container.yml
replicationcontroller/replication-controller-with-1-container configured
```

Comprobamos que hemos aumentado el número de réplicas de 3 a 5

```
$ kubectl get rc
```

NAME	DESIRED	CURRENT	READY	AGE
replication-controller-with-1-container	5	5	5	29m

- Mediante comando el línea

Ahora, vamos a ajustar de 5 a 10 réplicas la unidad del rc

```
$ kubectl scale --replicas=10 rs/replication-controller-with-1-container
```

replicationcontroller/replication-controller-with-1-container scaled

Comprobamos que hemos aumentado el número de réplicas de 5 a 10

```
$ kubectl get rc
```

NAME	DESIRED	CURRENT	READY	AGE
replication-controller-with-1-container	10	10	10	31m

38.7. Simulando caída de servidor

Vamos a crear una situación de emergencia, en la que uno de los nodos minions ha dejado de estar operativo, en caso de estar utilizando una máquina virtual con virtualbox, desconectamos la interfaz de red del minion2

Dejamos transcurrir unos 30 segundos y obtenemos el estado de los nodos del sistema

```
$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
kubemaster	Ready	master	44h	v1.16.0
kubeminion1	Ready	<none>	43h	v1.16.0
kubeminion2	NotReady	<none>	43h	v1.16.0

Llega un momento que el rc detecta que han caído ciertas réplicas, en este caso, el rc había alojado 5 Pods en cada máquina

Consultamos al rc el estado de las réplicas y ahora nos informa que de 10 ordenadas, se disponen de 5 en funcionamiento

```
$ kubectl get rc
```

NAME	DESIRED	CURRENT	READY	AGE
replication-controller-with-1-container	10	10	5	40m

Dejamos que transcurran unos 5 minutos, que es el tiempo por defecto que el Kube Scheduler tiene para ordenar de nuevo a la unidad rc un redespliegue de los Pods que han dejado de responder

Listamos de nuevo las unidades rc y observamos que se han recompuesto las 5 unidades que antes habían dejado de responder

```
$ kubectl get rc
```

NAME	DESIRED	CURRENT	READY	AGE
replication-controller-with-1-container	10	10	10	46m

Listamos de nuevo el conjunto de Pods, de manera que observaremos que algunos tienen orden de terminación, pero esta no es finalmente ejecutada por que el Kube Scheduler no ha establecido aún comunicación con el kubelet del Nodo afectado para poder ejecutar la orden de eliminación

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
replication-controller-with-1-container-5lwqv	1/1	Running	0	10m
replication-controller-with-1-container-5p4xn	1/1	Running	0	10m
replication-controller-with-1-container-chrlz	1/1	Running	0	24m
replication-controller-with-1-container-f8629	1/1	Terminating	0	26m
replication-controller-with-1-container-jnp8g	1/1	Terminating	0	24m
replication-controller-with-1-container-jw2rd	1/1	Terminating	0	55m
replication-controller-with-1-container-kp9mt	1/1	Terminating	0	24m
replication-controller-with-1-container-lsrql	1/1	Running	0	24m
replication-controller-with-1-container-p998x	1/1	Running	0	10m
replication-controller-with-1-container-qc8z5	1/1	Running	0	55m
replication-controller-with-1-container-qvhhg	1/1	Running	0	26m
replication-controller-with-1-container-rbk9v	1/1	Running	0	55m
replication-controller-with-1-container-sdt2n	1/1	Running	0	10m
replication-controller-with-1-container-w8ljm	1/1	Running	0	10m
replication-controller-with-1-container-xhssn	1/1	Terminating	0	24m

A continuación, reactivamos la red del minion2, de forma que al reactivarse serán eliminados los Pods en fase de terminación

Consultamos por último el listado de Pods del sistema y observamos las 10 réplicas en funcionamiento

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
replication-controller-with-1-container-5lwqv	1/1	Running	0	12m
replication-controller-with-1-container-5p4xn	1/1	Running	0	12m
replication-controller-with-1-container-chrlz	1/1	Running	0	26m
replication-controller-with-1-container-lsrql	1/1	Running	0	26m
replication-controller-with-1-container-p998x	1/1	Running	0	12m
replication-controller-with-1-container-qc8z5	1/1	Running	0	57m
replication-controller-with-1-container-qvhhg	1/1	Running	0	28m
replication-controller-with-1-container-rbk9v	1/1	Running	0	57m
replication-controller-with-1-container-sdt2n	1/1	Running	0	12m
replication-controller-with-1-container-w8ljm	1/1	Running	0	12m



La filosofía del clúster es que todo esté lo más balanceado posible, pero si no es necesario y el nodo en cuestión puede soportar perfectamente la carga de proceso de los 10 Pods no serán movidos aunque aparezcan nuevos nodos en el clúster

38.8. Obteniendo información detallada de las unidades rc

Podemos especificar un modificador cuando listamos las unidades rc de forma que podemos visualizar incluso los nombres de los contenedores de la unidad rc, la imagen y selector de etiquetas

```
$ kubectl get rc -o wide
```

NAME	DESIRDED	CURRENT	READY	AGE	CONTAINERS
IMAGES					
SELECTOR					
replication-controller-with-1-container	10	10	10	69m	nginx- container nginx customer=B8745

38.9. Eliminando las unidades rc

Por último, eliminamos las unidades rc deseadas

```
$ kubectl delete rc/replication-controller-with-1-container
```

```
replicationcontroller "replication-controller-with-1-container" deleted
```

Comprobamos que han desaparecido tanto la unidad de rc, como los Pods

```
$ kubectl get rc
```

No resources found in **default** namespace.

```
$ kubectl get pods
```

No resources found in **default** namespace.



Capítulo 39. ReplicaSet

La unidad ReplicaSet ha sido una nueva unidad de control de despliegue que ha aparecido recientemente, de forma que las unidades Deployment hacen uso de ella por defecto, en lugar de utilizar la unidad de replicación ReplicationController

39.1. ¿Cómo funciona la nueva unidad?

Un elemento ReplicaSet (rs) se define con una serie de campos, incluyendo un selector que especifica cómo se identifican los Pods que la unidad controla

Por otro lado, la unidad rs especifica cuántos Pods debe de mantener operativos, así como una plantilla de Pods para poder obtener nuevos Pods con una serie de características predefinidas (estructura de las nuevas réplicas)

La unidad rs opera su propósito creando y eliminando Pods según sea necesario hasta alcanzar el número ordenado

Cuando el rs necesita crear unos Pods, tira de la plantilla definida

39.2. Ventajas de uso

- El rs es una unidad superior al Pod
- Permite gestionar numero de pods del mismo tipo (replicas)
- Si hay más Pods de los ordenados, baja el número
- Si hay menos Pods de los ordenados, los aumenta hasta nivelar el número de replicas que han sido ordenadas
- Los Pods son reemplazados de forma automática en caso de fallo
- Es aconsejable utilizar el rs aún estando desplegando únicamente un Pod
- El funcionamiento es parecido a un supervisor de procesos, en lugar de supervisar procesos individuales en un sólo nodo, rs es capaz de supervisar múltiples Pods en múltiples Nodos
- A menudo se abrevia como "rs"

39.3. ¿Cuándo debemos de usarlo?

No requerimos de orquestación en las actualizaciones o sencillamente son sistemas que no vamos a actualizar nunca o que raramente se daría el caso

Pero si queremos de control de replicas

Entonces si sería un buen candidato a utilizar

39.4. Etiquetas en ReplicationController

El nexo de detección entre un rs y sus Pods es mediante un campo de metadatos por etiquetas, donde se especifica la propiedad del Pod

Todos los Pods adquiridos por la unidad rs tienen su propia información de identificación en el rs

Mediante ese nexo de detección, basado en etiquetas a modo de metadatos, el rs conoce el estado de los Pods que el mantiene y planifica en consecuencia

Un rs identifica nuevos Pods para adquirir utilizando su selector

Si hay un Pod que no tiene referencia de nexo al rs, y de repente la adquiere, la unidad rs lo hará suyo de forma inmediata para llevar el control del mismo



Fondos Europeos



Cofinanciado por
la Unión Europea



Capítulo 40. Lab: ReplicaSet

Mediante este laboratorio emplearemos diferentes casuísticas de uso con la unidad ReplicaSet

40.1. Creación del primer ReplicaSet

Queremos un despliegue de un servidor web nginx, compuesto de 3 réplicas en el clúster, con un etiquetado de que se trata de la parte frontend del sistema

Creamos el archivo de manifiesto **replicaset-with-1-container.yml** e indicamos el siguiente contenido

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: replicaset-with-1-container
  labels:
    app: frontend
spec:
  replicas: 3
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
        - name: nginx-container
          image: nginx
          ports:
            - containerPort: 80
```

Ejecutamos la instrucción de creación

```
$ kubectl apply -f replicaset-with-1-container.yml
replicaset.apps/replicaset-with-1-container created
```

40.2. Análisis de estructura mínima

Como observamos, el ReplicaSet ha introducido nuevos elementos que pasamos a describir a continuación:

- **apiVersion**

- apps/v1, se ha actualizado la API y ya no tiramos directamente de v1

- **kind**

- ReplicaSet, indicamos la nueva unidad a utilizar

- **spec → replicas**

- Indicamos el número de Pods que tendremos en el estado final a desplegar

- **spec → selector → matchLabels**

- Indicamos un selector de etiqueta (ahora de 3 niveles), con formato key-value, esto le vale a Kubernetes en la unidad de replicación, para encontrar Pods que esta unidad tenga que gestionar

- **spec → template → metadata**

- Especificamos las etiquetas que a modo de metadata tendrán los Pods cuando se creen
- La sección del template realmente crea una plantilla, de modo que los siguientes Pods que sea necesario crear, cumplirán todos la misma estructura

- **spec → template → spec**

- Especificamos en la plantilla los contenedores que tendrá dentro el Pod, su imagen, puertos, etc.

40.3. Listando los Pods creados por el ReplicaSet

Comprobamos que efectivamente se han creado 3 Pods, ya que indicamos en la metadata del .yml **replicas: 3**

```
$ kubectl get pods

NAME                  READY STATUS  RESTARTS AGE
replicaset-with-1-container-fw5kj  1/1   Running  0       3m51s
replicaset-with-1-container-jfjzc  1/1   Running  0       3m51s
replicaset-with-1-container-n5d2b  1/1   Running  0       3m51s
```

40.4. Listando las unidades de replicación

Ahora, pasamos a listar las unidades ReplicaSet que tengamos en el sistema

El sistema debería de indicarnos que efectivamente tenemos 1 controlador de replicación, compuesto por 3 Pods, y que los 3 Pods están listos

```
$ kubectl get rs

NAME      DESIRED  CURRENT  READY  AGE
replicaset-with-1-container  3        3        3      4m46s
```

40.5. Describiendo la unidad de replicación

Ahora, vamos a realizar un describe al ReplicaSet

```
$ kubectl describe rs replicaset-with-1-container
```

```
Name: replicaset-with-1-container
Namespace: default
Selector: app=frontend
Labels: app=frontend
Annotations: kubectl.kubernetes.io/last-applied-configuration:
    {"apiVersion":"apps/v1","kind":"ReplicaSet","metadata":{"annotations":{},"name":"replicaset-with-1-container","namespace":"default"},"spec...
Replicas: 3 current / 3 desired
Pods Status: 3 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels: app=frontend
  Containers:
    nginx-container:
      Image: nginx
      Port: 80/TCP
      Host Port: 0/TCP
      Environment: <none>
      Mounts: <none>
      Volumes: <none>
  Events:
    Type Reason Age From Message
    ---- ---- - - -
    Normal SuccessfulCreate 6m11s replicaset-controller Created pod: replicaset-with-1-container-n5d2b
    Normal SuccessfulCreate 6m10s replicaset-controller Created pod: replicaset-with-1-container-jfjzc
    Normal SuccessfulCreate 6m10s replicaset-controller Created pod: replicaset-with-1-container-fw5kj
```

Observamos como el describe del rc nos indica interesantes datos:

- **Name**
 - Nombre del rs
- **Namespace**
 - Espacio de nombres en el que se encuentra la unidad rs
- **Selector**
 - Selector con formato key-value que utilizará el rs para detectar sus Pods, de forma que los identificará como suyos
- **Labels**
 - Etiquetas que la unidad rs tiene asignada como metadata propia

- **Replicas**

- Indica el número de réplicas actualmente operativas / Réplicas ordenadas a disponer

- **Pods Status**

- Indica el estado de los Pods que la unidad de replicación controla, en nuestro caso nos indica que los 3 Pods se encuentran operando normalmente

- **Pod Template**

- Indica la estructura que tendrán los nuevos Pods que la unidad rc vaya a crear

- **Events**

- Histórico de eventos de la unidad de replicación
- A la unidad de replicación no le interesa mostrar el detalle de cada Pod (Si se le ha hecho pull a la imagen, en qué Nodo se ha alojado, etc)
- A la unidad de replicación le interesa mostrar si el Pod ha sido creado, su nombre, desde cuando y si está operando normalmente

40.6. Escalando el número de réplicas

Vamos a dar una orden a la unidad rc para que aumente el número de réplicas de la unidad, pasando de 3 a 5

- Mediante archivo de manifiesto

Modificamos el archivo de manifiesto **replicaset-with-1-container.yml** e indicamos el siguiente contenido



Fondos Europeos



MINISTERIO DE EDUCACIÓN, FORMACIÓN PROFESIONAL Y DEPORTE
GOBIERNO DE ESPAÑA

Cofinanciado por
la Unión Europea



```

apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: replicaset-with-1-container
  labels:
    app: frontend
spec:
  replicas: 5
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
        - name: nginx-container
          image: nginx
          ports:
            - containerPort: 80

```

Ejecutamos la instrucción de actualización

```

$ kubectl apply -f replicaset-with-1-container.yml
replicaset.apps/replicaset-with-1-container configured

```

Comprobamos que hemos aumentado el número de réplicas de 3 a 5

```

$ kubectl get rs
NAME           DESIRED  CURRENT  READY  AGE
replicaset-with-1-container  5       5       5     15m

```

- Mediante comando el línea

Ahora, vamos a ajustar de 5 a 10 réplicas la unidad del rc

```

$ kubectl scale --replicas=10 rs/replicaset-with-1-container
replicaset.apps/replicaset-with-1-container scaled

```

Comprobamos que hemos aumentado el número de réplicas de 5 a 10

```
$ kubectl get rs
```

NAME	DESIRED	CURRENT	READY	AGE
replicaset-with-1-container	10	10	10	17m

40.7. Simulando caída de servidor

Vamos a crear una situación de emergencia, en la que uno de los nodos minions ha dejado de estar operativo, en caso de estar utilizando una máquina virtual con virtualbox, desconectamos la interfaz de red del minion2

Dejamos transcurrir unos 30 segundos y obtenemos el estado de los nodos del sistema

```
$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
kubemaster	Ready	master	46h	v1.16.0
kubeminion1	Ready	<none>	45h	v1.16.0
kubeminion2	NotReady	<none>	45h	v1.16.0

Llega un momento que el rs detecta que han caído ciertas réplicas, en este caso, el rs había alojado 5 Pods en cada máquina

Consultamos al rs el estado de las réplicas y ahora nos informa que de 10 ordenadas, se disponen de 5 en funcionamiento

```
$ kubectl get rs
```

NAME	DESIRED	CURRENT	READY	AGE
replicaset-with-1-container	10	10	5	19m

Dejamos que transcurran unos 5 minutos, que es el tiempo por defecto que el Kube Scheduler tiene para ordenar de nuevo a la unidad rs un redespliegue de los Pods que han dejado de responder

Ejecutamos el comando de monitoreo de Pods en tiempo real hasta que algunos (5) muestren el estado "Terminating"

También observamos como el sistema se recomponen de forma automática y aparecen nuevos contenedores en fase de creación y posterior arranque

```
$ kubectl get pods --watch
```

NAME	READY	STATUS	RESTARTS	AGE
replicaset-with-1-container-fw5kj	1/1	Running	0	20m
replicaset-with-1-container-jfjzc	1/1	Running	0	20m
replicaset-with-1-container-k6rj8	1/1	Running	0	3m29s
replicaset-with-1-container-lj8xg	1/1	Running	0	3m29s
replicaset-with-1-container-lptvh	1/1	Running	0	6m21s
replicaset-with-1-container-lw7n6	1/1	Running	0	6m21s
replicaset-with-1-container-n5d2b	1/1	Running	0	20m
replicaset-with-1-container-npmtk	1/1	Running	0	3m29s
replicaset-with-1-container-vz8nx	1/1	Running	0	3m29s
replicaset-with-1-container-wvm5n	1/1	Running	0	3m29s
replicaset-with-1-container-lw7n6	1/1	Terminating	0	10m
replicaset-with-1-container-n5d2b	1/1	Terminating	0	24m
replicaset-with-1-container-lj8xg	1/1	Terminating	0	7m8s
replicaset-with-1-container-lptvh	1/1	Terminating	0	10m
replicaset-with-1-container-npmtk	1/1	Terminating	0	7m8s
replicaset-with-1-container-kr9kl	0/1	Pending	0	0s
replicaset-with-1-container-kr9kl	0/1	Pending	0	0s
replicaset-with-1-container-4q958	0/1	Pending	0	1s
replicaset-with-1-container-4q958	0/1	Pending	0	1s
replicaset-with-1-container-svs99	0/1	Pending	0	0s
replicaset-with-1-container-n5nqd	0/1	Pending	0	0s
replicaset-with-1-container-kr9kl	0/1	ContainerCreating	0	1s
replicaset-with-1-container-svs99	0/1	Pending	0	0s
replicaset-with-1-container-97hn	0/1	Pending	0	0s
replicaset-with-1-container-n5nqd	0/1	Pending	0	0s
replicaset-with-1-container-97hn	0/1	Pending	0	0s
replicaset-with-1-container-4q958	0/1	ContainerCreating	0	1s
replicaset-with-1-container-svs99	0/1	ContainerCreating	0	0s
replicaset-with-1-container-n5nqd	0/1	ContainerCreating	0	1s
replicaset-with-1-container-97hn	0/1	ContainerCreating	0	1s
replicaset-with-1-container-kr9kl	1/1	Running	0	18s
replicaset-with-1-container-svs99	1/1	Running	0	18s
replicaset-with-1-container-4q958	1/1	Running	0	22s
replicaset-with-1-container-n5nqd	1/1	Running	0	23s
replicaset-with-1-container-97hn	1/1	Running	0	25s

Listamos de nuevo las unidades rs y observamos que se han recompuesto las 5 unidades que antes habían dejado de responder

```
$ kubectl get rs
```

NAME	DESIRED	CURRENT	READY	AGE
replicaset-with-1-container	10	10	10	28m

Listamos de nuevo el conjunto de Pods, de manera que observaremos que algunos tienen orden de

terminación, pero esta no es finalmente ejecutada por que el Kube Scheduler no ha establecido aún comunicación con el kubelet del Nodo afectado para poder ejecutar la orden de eliminación

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
replicaset-with-1-container-4q958	1/1	Running	0	5m15s
replicaset-with-1-container-97hnh	1/1	Running	0	5m14s
replicaset-with-1-container-fw5kj	1/1	Running	0	29m
replicaset-with-1-container-jfjzc	1/1	Running	0	29m
replicaset-with-1-container-k6rj8	1/1	Running	0	12m
replicaset-with-1-container-kr9kl	1/1	Running	0	5m15s
replicaset-with-1-container-lj8xg	1/1	Terminating	0	12m
replicaset-with-1-container-lptvh	1/1	Terminating	0	15m
replicaset-with-1-container-lw7n6	1/1	Terminating	0	15m
replicaset-with-1-container-n5d2b	1/1	Terminating	0	29m
replicaset-with-1-container-n5nqd	1/1	Running	0	5m14s
replicaset-with-1-container-npmtk	1/1	Terminating	0	12m
replicaset-with-1-container-svs99	1/1	Running	0	5m14s
replicaset-with-1-container-vz8nx	1/1	Running	0	12m
replicaset-with-1-container-wvm5n	1/1	Running	0	12m



A continuación, reactivamos la red del minion2, de forma que al reactivarse serán eliminados los Pods en fase de terminación

Consultamos por último el listado de Pods del sistema y observamos las 10 réplicas en funcionamiento

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
replicaset-with-1-container-4q958	1/1	Running	0	7m39s
replicaset-with-1-container-97hnh	1/1	Running	0	7m38s
replicaset-with-1-container-fw5kj	1/1	Running	0	31m
replicaset-with-1-container-jfjzc	1/1	Running	0	31m
replicaset-with-1-container-k6rj8	1/1	Running	0	14m
replicaset-with-1-container-kr9kl	1/1	Running	0	7m39s
replicaset-with-1-container-n5nqd	1/1	Running	0	7m38s
replicaset-with-1-container-svs99	1/1	Running	0	7m38s
replicaset-with-1-container-vz8nx	1/1	Running	0	14m
replicaset-with-1-container-wvm5n	1/1	Running	0	14m



La filosofía del clúster es que todo esté lo más balanceado posible, pero si no es necesario y el nodo en cuestión puede soportar perfectamente la carga de proceso de los 10 Pods no serán movidos aunque aparezcan nuevos nodos en el clúster



40.8. Añadiendo manualmente un Pod con nexo al ReplicaSet

Ahora vamos a crear de forma aislada un Pod, pero vamos a añadir metadata de etiquetas, para ver si la unidad de replicación ReplicaSet es capaz de detectarlo y hacerlo suyo

Primero monitoreamos el sistema con los Pods que están desplegados, de forma que tengamos visión en tiempo real de los Pods que se van creando y destruyendo

```
$ kubectl get pods --watch
```

NAME	READY	STATUS	RESTARTS	AGE
replicaset-with-1-container-46ht6	1/1	Running	0	4m1s
replicaset-with-1-container-5vs8z	1/1	Running	0	4m
replicaset-with-1-container-d4ljp	1/1	Running	0	4m
replicaset-with-1-container-gvj4c	1/1	Running	0	3m59s
replicaset-with-1-container-ns2cs	1/1	Running	0	4m1s
replicaset-with-1-container-q979j	1/1	Running	0	3m59s
replicaset-with-1-container-t4qfx	1/1	Running	0	3m59s
replicaset-with-1-container-thrss	1/1	Running	0	4m1s
replicaset-with-1-container-twj9h	1/1	Running	0	4m2s
replicaset-with-1-container-xk69j	1/1	Running	0	4m

Creamos en otra consola el archivo de manifiesto **single-pod-for-replicaset.yml** e indicamos el siguiente contenido

```
apiVersion: v1
kind: Pod
metadata:
  name: single-pod-for-replicaset
  labels:
    app: frontend
spec:
  containers:
    - name: nginx-container
      image: nginx
      ports:
        - containerPort: 80
```

Ejecutamos la orden de creación

```
$ kubectl apply -f single-pod-for-replicaset.yml
pod/single-pod-for-replicaset created
```

Vamos a la consola donde estábamos monitoreando las secuencias de los Pods y vamos a observar

como el nuevo Pod automáticamente recibe la orden de destrucción, ya que el rs que está activo toma el control, detecta que ya hay 10 réplicas activas y no hay necesidad de incorporar la 11 por que no se le ha ordenado, por lo tanto la destruye de forma inmediata

NAME	READY	STATUS	RESTARTS	AGE
replicaset-with-1-container-46ht6	1/1	Running	0	4m1s
replicaset-with-1-container-5vs8z	1/1	Running	0	4m
replicaset-with-1-container-d4ljp	1/1	Running	0	4m
replicaset-with-1-container-gvj4c	1/1	Running	0	3m59s
replicaset-with-1-container-ns2cs	1/1	Running	0	4m1s
replicaset-with-1-container-q979j	1/1	Running	0	3m59s
replicaset-with-1-container-t4qfx	1/1	Running	0	3m59s
replicaset-with-1-container-thrss	1/1	Running	0	4m1s
replicaset-with-1-container-twj9h	1/1	Running	0	4m2s
replicaset-with-1-container-xk69j	1/1	Running	0	4m
single-pod-for-replicaset	0/1	Pending	0	0s
single-pod-for-replicaset	0/1	Pending	0	1s
single-pod-for-replicaset	0/1	Pending	0	1s
single-pod-for-replicaset	0/1	Terminating	0	1s
single-pod-for-replicaset	0/1	Terminating	0	1s
single-pod-for-replicaset	0/1	Terminating	0	1s
single-pod-for-replicaset	0/1	Terminating	0	4s
single-pod-for-replicaset	0/1	Terminating	0	5s
single-pod-for-replicaset	0/1	Terminating	0	5s



Fondos Europeos

MINISTERIO DE EDUCACIÓN, FORMACIÓN PROFESIONAL Y DEPORTE
GOBIERNO DE ESPAÑA

Cofinanciado por la Unión Europea

40.9. Obteniendo información detallada de las unidades rs

Podemos especificar un modificador cuando listamos las unidades rs de forma que podemos visualizar incluso los nombres de los contenedores de la unidad rs, la imagen y selector de etiquetas

```
$ kubectl get rs -o wide
```

NAME	DESIRDED	CURRENT	READY	AGE	CONTAINERS
IMAGES SELECTOR					
replicaset-with-1-container	10	10	10	37m	nginx-container
nginx	app=frontend				

40.10. Obteniendo metadata en formato .yml de un pod desplegado por el rs

Otra opción interesante es la obtención de metadata a modo de ingeniería inversa del propio Pod

Un caso de uso podría ser, disponemos de un Pod que lo ha creado un ReplicaSet y queremos obtener información del dueño del Pod

Listamos todos los Pods del sistema

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
replicaset-with-1-container-2kl7x	1/1	Running	0	5m42s
replicaset-with-1-container-2mrl9	1/1	Running	0	5m42s
replicaset-with-1-container-4pkpn	1/1	Running	0	5m42s
replicaset-with-1-container-9295d	1/1	Running	0	5m42s
replicaset-with-1-container-d8msk	1/1	Running	0	5m42s
replicaset-with-1-container-qfsqq	1/1	Running	0	5m42s
replicaset-with-1-container-qv659	1/1	Running	0	5m42s
replicaset-with-1-container-rqqsm	1/1	Running	0	5m42s
replicaset-with-1-container-wqr5x	1/1	Running	0	5m42s
replicaset-with-1-container-wxxbf	1/1	Running	0	5m42s

A continuación ejecutamos el siguiente comando contra uno de los Pods

```
$ kubectl get pods replicaset-with-1-container-2kl7x -o yaml
```



```
...  
name: replicaset-with-1-container-2kl7x  
namespace: default  
ownerReferences:  
- apiVersion: apps/v1  
blockOwnerDeletion: true  
controller: true  
kind: ReplicaSet  
name: replicaset-with-1-container  
uid: d60d70b2-5056-4eea-ac08-7e164505e983  
resourceVersion: "139818"  
selfLink: /api/v1/namespaces/default/pods/replicaset-with-1-container-2kl7x  
uid: 905c2485-733e-4e52-b206-64f221d0358b  
spec:  
  containers:  
  ...
```

Si nos fijamos en la sección **ownerReferences**, ahí aparecerán datos específicos de quien tiene la potestad del Pod

40.11. Eliminando las unidades rs con Pods

Por último, eliminamos las unidades rs deseadas

```
$ kubectl delete rs/replicaset-with-1-container  
replicaset.apps "replicaset-with-1-container" deleted
```

Comprobamos que han desaparecido tanto la unidad de rs, como los Pods

```
$ kubectl get rs  
No resources found in default namespace.
```

```
$ kubectl get pods  
No resources found in default namespace.
```

40.12. Eliminando únicamente la unidad rs sin eliminar los Pods

Eliminamos el rs de esta forma

```
$ kubectl delete rs replicaset-with-1-container --cascade=false
```

Comprobamos que ha desaparecido la unidad de rs

```
$ kubectl get rs  
No resources found in default namespace.
```

Y comprobamos que los Pods siguen operativos aún habiendo desaparecido la unidad rs

```
$ kubectl get pods
```

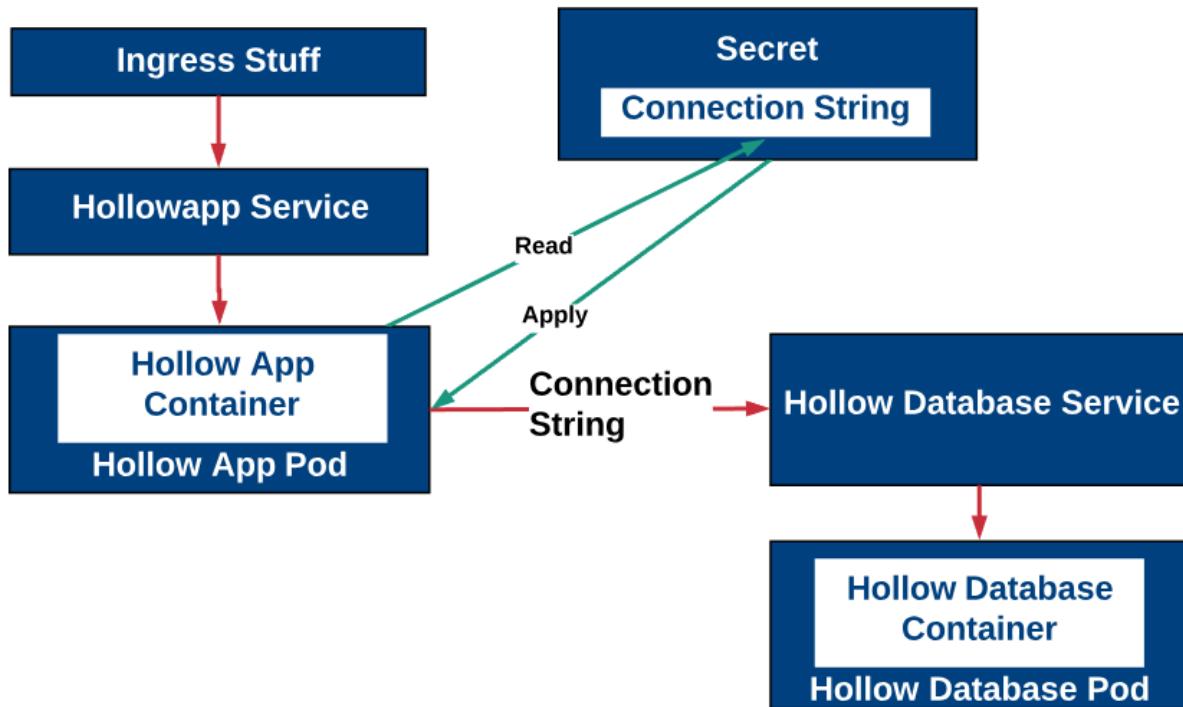
NAME	READY	STATUS	RESTARTS	AGE
replicaset-with-1-container-2kl7x	1/1	Running	0	21m
replicaset-with-1-container-2mrl9	1/1	Running	0	21m
replicaset-with-1-container-4pkpn	1/1	Running	0	21m
replicaset-with-1-container-9295d	1/1	Running	0	21m
replicaset-with-1-container-d8msk	1/1	Running	0	21m
replicaset-with-1-container-qfsqq	1/1	Running	0	21m
replicaset-with-1-container-qv659	1/1	Running	0	21m
replicaset-with-1-container-rqqsm	1/1	Running	0	21m
replicaset-with-1-container-wqr5x	1/1	Running	0	21m
replicaset-with-1-container-wxxbf	1/1	Running	0	21m

Si de nuevo volvemos a crear la unidad rs, esta acapará de nuevo los Pods y podremos volver a eliminar la unidad rs arrastrando el borrado de los Pods, sin incluir el --cascade=false



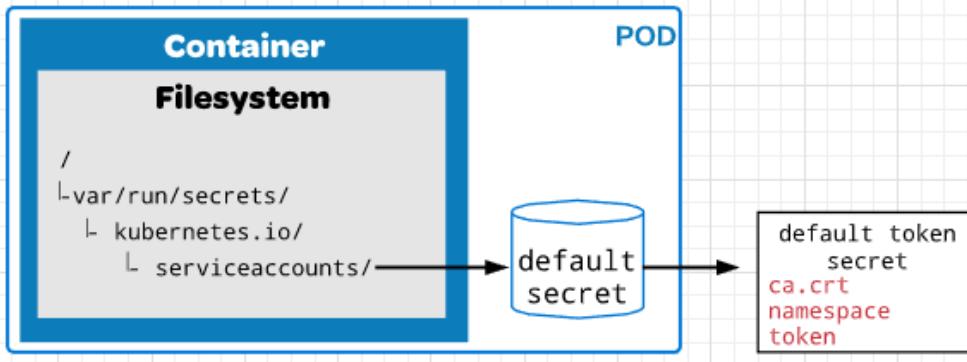
Capítulo 41. Secrets

Los secretos en kubernetes son elementos donde podemos almacenar información sensible al usuario, como contraseñas, tokens, claves ssh, etc.



Secrets

Secrets allow you to expose entries as files in a volume. Keeping this data secure is paramount to cluster security.



41.1. ¿Para qué valen?

Si ese tipo de información la introducimos mediante secretos, nos aportará seguridad y flexibilidad en lo que a un contenedor se refiere

Ya que los valores de los secrets únicamente podrán ser accedidos a nivel interno por los contenedores

Dispondremos de más control sobre elementos de configuración que queramos gestionar y reduciremos el riesgo de exposición accidental de información sensible

En definitiva, los secretos se utilizan para proteger datos confidenciales a los que puede acceder el Pod

Los datos nunca se escriben en el disco, por que se almacenan en un sistema de archivos en memoria (tmpfs)

Debido a que los secretos se pueden crear independientemente de los Pods, hay menos riesgo de que el secreto se exponga durante el ciclo de vida del Pod.

En esencia, los secrets son mapas de conjunto en formato clave - valor

41.2. ¿Quién puede utilizar secrets?

Los usuarios del clúster pueden crear secretos y el sistema kubernetes en sí también lo hace para establecer configuraciones a nivel interno entre sus Pods de gestión y control

Para que un Pod pueda utilizar un secreto, este debe de hacerle referencia al mismo

Los secretos pueden utilizarse en un Pod de 2 formas:

- **Mediante Volumen**
 - Montado un volumen entre uno o más contenedores
- **Utilizando kubelet**
 - Cuando inicia la secuencia de descarga para crear los Pods que a su vez tendrán dentro contenedores

Capítulo 42. Lab: Secrets

Mediante esta laboratorio, vamos a experientar las distintas operatorias que podemos llegar a realizar con los secrets

42.1. Creando nuestros propios secretos con kubectl (inline)

Imaginenos que necesitamos dos credenciales de acceso como usuario y contraseña que las tenemos almacenadas en 2 archivos

Ejecutamos en la consola estas dos instrucciones

```
$ echo -n 'root' > ./username.txt  
$ echo -n 'my-perfect-password' > ./password.txt
```

Si necesitamos especificar caracteres especiales (\$, ! o *) en los secrets, debemos de utilizar caracteres de escape especiales como el \\

Por ejemplo, imaginemos que queremos poner como valor de la clave la siguiente secuencia de caracteres "S!B*d\$zDsb"

Debemos entonces de ejecutar el siguiente comando para coger los valores por ejemplo in line

```
kubectl create secret generic dev-db-secret --from-literal=password=S\!B\\*d\$zDsb
```

Y ahora creamos el secreto

```
$ kubectl create secret generic db-access --from-file=./username.txt --from-file=.  
/password.txt  
  
secret/db-access created
```

- Indicamos como flag de tipo **generic**, para obtener el secreto desde archivos, directorios o incluso en forma de literal
- Como nombre del secret indicamos **db-access**
- Indicamos las dos rutas de archivos con los modificadores **--from-file**
 - Si quisiéramos indicar un literal sin archivo, indicamos **--from-literal**

42.2. Listando secretos

Para listar los secretos, ejecutamos el siguiente comando

```
$ kubectl get secrets
```

NAME	TYPE	DATA	AGE
db-access	Opaque	2	4m15s

42.3. Describiendo los secretos

Para realizar una descripción del secreto, podemos hacer uso del comando describe

```
$ kubectl describe secrets/db-access
```

Name: db-access

Namespace: default

Labels: <none>

Annotations: <none>

Type: Opaque

Data

====

password.txt: 19 bytes

username.txt: 4 bytes

Por defecto, observamos que tanto los comandos kubectl get y describe, no enseñan los valores contenidos en los secretos por defecto

Esto es así para proteger el secreto ante una exposición accidental del mismo en una consulta de terminal, etc.

42.4. Creando secretos mediante archivo de manifiesto (.yml)

Ahora, vamos a crear otro secreto con 2 valores, pero esta vez mediante archivo de manifiesto, para dejar constancia del mismo

Por defecto, tenemos que introducir la información en el secreto codificado en base64

Primero vamos a proceder a obtener el login (admin) de usuario en base64

```
echo -n 'admin' | base64  
YWRtaW4=
```

A continuación, vamos a obtener la clave del usuario (1f2d1e2e67df) también en formato base64

```
echo -n '1f2d1e2e67df' | base64  
MWYyZDFIMmU2N2Rm
```

Creamos el archivo **database-secret.yml** y agregamos el siguiente contenido

```
apiVersion: v1  
kind: Secret  
metadata:  
  name: mysecret  
type: Opaque  
data:  
  username: YWRtaW4=  
  password: MWYyZDFIMmU2N2Rm
```

Procedemos a crear el secreto

```
$ kubectl apply -f database-secret.yml  
  
secret/mysecret created
```

42.5. Obteniendo descripción detallada del secreto

Si hacemos uso de la ingeniería inversa que nos proporciona el modificador `-o yaml`, podremos obtener la estructura del .yaml a partir del secreto que se encuentra en funcionamiento

```
$ kubectl get secret mysecret -o yaml  
  
apiVersion: v1  
data:  
  password: MWYyZDFIMmU2N2Rm  
  username: YWRtaW4=  
kind: Secret  
metadata:  
  annotations:  
    kubectl.kubernetes.io/last-applied-configuration: |  
      {"apiVersion":"v1","data":{"password":"MWYyZDFIMmU2N2Rm","username":"YWRtaW4="}}  
    , "kind": "Secret", "metadata": {"annotations": {}, "name": "mysecret", "namespace": "default"}  
    , "type": "Opaque"}  
  creationTimestamp: "2019-09-25T18:27:34Z"  
  name: mysecret  
  namespace: default  
  resourceVersion: "145361"  
  selfLink: /api/v1/namespaces/default/secrets/mysecret  
  uid: a997e77b-e160-44f9-a89c-56014251a4a4  
type: Opaque
```

42.6. Creando secretos que no estén en base64

En ocasiones también nos puede ocurrir que el contenido de los secretos por alguna razón no podamos introducirlos en base64, debiendo introducir estos en formato de texto plano tal y como tenemos los valores

Para conseguir ese efecto, tenemos que introducir en el .yml la instrucción **stringData**

Vamos a crear otro secreto, pero en este caso, con los valores en plano, creamos el archivo **database-secret-with-string-data.yml**

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret-text-plain
type: Opaque
stringData:
  user: Pakito
  password: verySecurePassword
```

Ejecutamos la creación del secreto

```
$ kubectl apply -f database-secret-with-string-data.yml
secret/mysecret-text-plain created
```

Ahora, procedemos a obtener la estructura yml del secreto **mysecret-text-plain**

```
$ kubectl get secret mysecret-text-plain -o yaml
```

```
apiVersion: v1
data:
  password: dmVyeVNlY3VyZVBhc3N3b3Jk
  user: UGFraXRv
kind: Secret
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"v1","kind":"Secret","metadata":{"annotations":{},"name":"mysecret-text-plain","namespace":"default"},"stringData":{"password":"verySecurePassword","user":"Pakito"},"type":"Opaque"}
    creationTimestamp: "2019-09-25T18:38:59Z"
  name: mysecret-text-plain
  namespace: default
  resourceVersion: "146349"
  selfLink: /api/v1/namespaces/default/secrets/mysecret-text-plain
  uid: bec89cde-3143-4e5d-9191-2ca859a239cf
type: Opaque
```

Observamos que a diferencia del caso anterior, aquí si aparecen los valores en texto plano y no codificados en base64

42.7. Prioridad a la hora de definir secretos

Imaginemos que estamos definiendo un secreto con la misma key tanto en base64 como en texto plano, si se da esta casuística, kubernetes dará prioridad al texto plano (stringData)

Vamos a crear otro secreto, pero en este caso, repitiendo la key, primero codificada en base64 y a la vez en texto plano, creamos el archivo **database-secret-with-mixing-base64-string-data.yml**

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret-mixing-base64-text-plain
type: Opaque
data:
  user: UGFraXRv
  stringData:
    user: PakitoPakito
```

Ejecutamos la creación del secreto

```
$ kubectl apply -f database-secret-with-mixing-base64-string-data.yml  
secret/mysecret-mixing-base64-text-plain created
```

Ahora, procedemos a obtener la estructura yml del secreto **mysecret-mixing-base64-text-plain**

```
$ kubectl get secret mysecret-mixing-base64-text-plain -o yaml
```

```
apiVersion: v1  
data:  
  user: UGFraXRvUGFraXRv  
kind: Secret  
metadata:  
  annotations:  
    kubectl.kubernetes.io/last-applied-configuration: |  
      {"apiVersion":"v1", "data":{"user":"UGFraXRv"}, "kind":"Secret", "metadata":{  
        "annotations":{}, "name":"mysecret-mixing-base64-text-plain", "namespace":"default"},  
        "stringData":{"user":"PakitoPakito"}, "type":"Opaque"}  
    creationTimestamp: "2019-09-25T18:52:56Z"  
  name: mysecret-mixing-base64-text-plain  
  namespace: default  
  resourceVersion: "147560"  
  selfLink: /api/v1/namespaces/default/secrets/mysecret-mixing-base64-text-plain  
  uid: 67892fef-224b-4d53-9eeb-b84a497c14f5  
  type: Opaque
```

Observamos que el valor de la key "user" aparece en base64 y es bastante más extensa que el valor que indicamos en el .yml para el data → user, lo cual demuestra que ha tenido prioridad el stringData, adicionalmente de forma automática, kubernetes se encarga de codificar el texto a base64

42.8. Decodificando un secreto que esté en base64

Una vez hayamos realizado un describe, podemos directamente en la consola llevar a cabo la operación inversa para obtener el texto en claro

```
$ echo 'UGFraXRvUGFraXRv' | base64 --decode  
PakitoPakito
```

42.9. Editando un secreto

Otra opción interesante, es poder llevar a cabo la edición de un secreto que ya está creado, lo idea es realizar la tarea mediante el archivo de manifiesto .yml pero también podríamos hacer con sintaxis inline a través de la herramienta kubectl

Vamos a editar el secreto mysecret-mixing-base64-text-plain

```
$ kubectl edit secrets mysecret-mixing-base64-text-plain
```

Nos aparecerá el editor vi, localizamos la entrada data con la key user, cambiamos el valor por este otro 'YWRtaW4=' (admin)

Guardamos los cambios y la consola informará de la edición

```
secret/mysecret-mixing-base64-text-plain edited
```

Realizamos de nuevo una operación de descripción del secreto, y observaremos que el valor de la key ha cambiado por el que hemos editado

```
$ kubectl get secret mysecret-mixing-base64-text-plain -o yaml
```

```
apiVersion: v1
data:
  user: YWRtaW4=
kind: Secret
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"v1","data":{"user":"UGFraXRv"},"kind":"Secret","metadata":{ "annotations":{},"name":"mysecret-mixing-base64-text-plain","namespace":"default"}, "stringData":{"user":"PakitoPakito"},"type":"Opaque"}
  creationTimestamp: "2019-09-25T18:52:56Z"
  name: mysecret-mixing-base64-text-plain
  namespace: default
  resourceVersion: "148515"
  selfLink: /api/v1/namespaces/default/secrets/mysecret-mixing-base64-text-plain
  uid: 67892fef-224b-4d53-9eeb-b84a497c14f5
  type: Opaque
```

42.10. Utilizando un secreto dentro de un Pod

Múltiples Pods pueden referenciar al mismo secreto

Para utilizar el secreto, vamos a definir un volumen en la propia especificación del Pod y vamos a indicar un punto de montaje, adicionalmente por seguridad, indicaremos que el volumen montado dentro del Pod es de sólo lectura

Reglas de montaje

- Cada secreto que queramos utilizar dentro de un Pod necesita ser referenciado a través del bloque **spec.volumes**

- Si hay múltiples contendores en el Pod, cada contenedor necesitará su propio bloque de montaje de volumen **volumeMount**, pero todos compartirán el bloque **spec.volumes**

Vamos a crear un Pod que utilice el secreto previamente creado **mysecret-mixing-base64-text-plain**

Creamos un archivo con nombre **pod-using-secret.yml** e indicamos el siguiente contenido

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-using-secret
spec:
  containers:
    - name: redis-container
      image: redis:latest
      volumeMounts:
        - name: secret-from-kubernetes
          mountPath: "/my-secret-path"
          readOnly: true
  volumes:
    - name: secret-from-kubernetes
      secret:
        secretName: mysecret-mixing-base64-text-plain
```

Ejecutamos la creación del Pod

```
$ kubectl apply -f pod-using-secret.yml
pod/pod-using-secret created
```

Accedemos dentro del Pod

```
$ kubectl exec -it pod-using-secret bash
```

Una vez dentro del contenedor, nos dirigimos a la ruta donde está el volumen montado **/my-secret-path**, listamos el contenido y observaremos que tenemos la key "user"

```
root@pod-using-secret:/my-secret-path# ls -la
total 0
drwxrwxrwt 3 root root 100 Sep 25 20:28 .
drwxr-xr-x 1 root root 51 Sep 25 20:28 ..
drwxr-xr-x 2 root root 60 Sep 25 20:28 ..2019_09_25_20_28_40.598601627
lrwxrwxrwx 1 root root 31 Sep 25 20:28 ..data -> ..2019_09_25_20_28_40.598601627
lrwxrwxrwx 1 root root 11 Sep 25 20:28 user -> ..data/user
```

Y mostramos el valor de la key "user", por ejemplo mediante el comando cat, observamos que en consola obtenemos el valor el claro que era "admin", según la última edición que le hicimos al Pod

```
adminroot@pod-using-secret:/my-secret-path# ls -la
total 0
drwxrwxrwt 3 root root 100 Sep 25 20:28 .
drwxr-xr-x 1 root root 51 Sep 25 20:28 ..
drwxr-xr-x 2 root root 60 Sep 25 20:28 ..2019_09_25_20_28_40.598601627
lrwxrwxrwx 1 root root 31 Sep 25 20:28 ..data -> ..2019_09_25_20_28_40.598601627
lrwxrwxrwx 1 root root 11 Sep 25 20:28 user -> ..data/user
root@pod-using-secret:/my-secret-path# cat user
*admin*root@pod-using-secret:/my-secret-path#
```

42.11. Utilizando un secreto dentro de un Pod (especificando secretos y rutas de montado de claves)

Otro caso que se nos puede dar, es que queramos montar sólo una de las claves del secreto, y además que la ruta de montaje sea personalizada

Vamos a utilizar nuestro secreto previamente creado **db-access** que tenía un par de elementos, username y password

Vamos a crear un archivo con nombre **pod-using-secret-with-custom-paths.yml** e indicamos el siguiente contenido

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-using-secret-with-custom-paths
spec:
  containers:
    - name: redis-using-secret-with-custom-path
      image: redis
      volumeMounts:
        - name: volume-for-db-credentials
          mountPath: "/my-beautifoul-path/data"
          readOnly: true
  volumes:
    - name: volume-for-db-credentials
      secret:
        secretName: db-access
        items:
          - key: username.txt
            path: credentials/my-login-username
```

Recordamos, que cuando montamos el secreto db-access fue mediante archivos, con lo que las keys de los valores de los archivos, son los nombres de los propios archivos



Tenemos que hacer referencia en este caso a la clave de nombre de usuario mediante username.txt

Ejecutamos la orden de creación del Pod

```
$ kubectl apply -f pod-using-secret-with-custom-paths.yml  
pod/pod-using-secret-with-custom-paths created
```

Nos conectamos dentro del container

```
$ kubectl exec -it pod-using-secret-with-custom-paths bash
```

Una vez dentro del container, accedemos a la ruta **/my-beautifoul-path/data/credentials** y dentro de esta ejecutamos un comando cat para obtener el valor del secreto con la key **my-login-username**

```
total 4  
drwxr-xr-x 2 root root 60 Sep 25 20:56 .  
drwxr-xr-x 3 root root 60 Sep 25 20:56 ..  
-rw-r--r-- 1 root root 4 Sep 25 20:56 my-login-username  
root@pod-using-secret-with-custom-paths:/my-beautifoul-path/data/credentials# pwd  
/my-beautifoul-path/data/credentials  
root@pod-using-secret-with-custom-paths:/my-beautifoul-path/data/credentials# cat my-  
login-username  
*root*root@pod-using-secret-with-custom-paths:/my-beautifoul-path/data/credentials#
```

Observamos como obtenemos el value de **root**, cuyo valor fue originalmente el que asignamos al archivo username.txt

42.12. Especificando permisos de los archivos de montaje

Cuando montamos archivos de un secreto a un volumen, por defecto si no especificamos nada, los archivos se montan con el permiso **0644** por defecto

Podemos especificar un modo por defecto de montaje de los secretos en el volumen, incluso a nivel de cada clave si fuera necesario

La notación JSON (si utilizamos) no permite indicar notación Octal, con lo que el valor que usaremos para indicar por ejemplo el permiso 0777 será el 511 entero



Si utilizamos para realizar el despliegue notación yml no habrá ningún problema y podremos utilizar notación Octal para poder utilizar una notación algo más natural

Creamos el archivo con nombre **pod-secret-filepermission.yml** e indicamos el siguiente contenido

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-secret-filepermission
spec:
  containers:
    - name: container-with-nginx
      image: nginx
      volumeMounts:
        - name: my-beautiful-volume
          mountPath: "/my-secrets-password-path"
  volumes:
    - name: my-beautiful-volume
      secret:
        secretName: db-access
        defaultMode: 511
```



Ejecutamos la orden de creación del Pod

```
$ kubectl apply -f pod-secret-filepermission.yml
pod/pod-secret-filepermission created
```

Nos conectamos dentro del container

```
$ kubectl exec -it pod-secret-permission bash
```

Una vez dentro del container, accedemos a la ruta **/my-secrets-password-path** y observaremos dos keys, la username y la password

```
drwxrwxrwt 3 root root 120 Sep 25 21:21 .
drwxr-xr-x 1 root root 60 Sep 25 21:22 ..
drwxr-xr-x 2 root root 80 Sep 25 21:21 ..2019_09_25_21_21_57.175726636
lrwxrwxrwx 1 root root 31 Sep 25 21:21 ..data -> ..2019_09_25_21_21_57.175726636
lrwxrwxrwx 1 root root 19 Sep 25 21:21 password.txt -> ..data/password.txt
lrwxrwxrwx 1 root root 19 Sep 25 21:21 username.txt -> ..data/username.txt
root@pod-secret-filepermission:/my-secrets-password-path#
```

Si le hacemos un cat a cualquier de las claves obtendremos el valor en claro

42.13. Actualización de los secretos

Cuando un secreto está siendo consumido por un volumen que lo ha montado, las claves son proyectadas hacia el volumen con los valores que tengan cuando se actualicen

El agente Kubelet periódicamente procede a realizar comprobaciones para refrescar los valores de las key si estos hubieran cambiado

El proceso de refresco de los valores de las claves en un secreto creado no es instantáneo ya que kubernetes crea una caché precisamente para conseguir eficiencia en las lecturas a disco de los volúmenes

42.14. Utilizando secretos como variables de entorno

En este caso, vamos a utilizar secretos como variables de entorno en un Pod

Utilizaremos el secreto anteriormente creado **db-access**

La login de usuario se encuentra en la key **username.txt**

La contraseña de usuario se encuentra en la key **password.txt**

Indicamos una política de reinicio Never

En este caso no vamos a montar volúmenes

Vamos a crear un contenedor con un sistema nginx que tenga inyectadas 2 variables de entorno, una para el usuario y otra para la contraseña

Creamos el archivo con nombre **pod-secret-with-env-variables.yml** e indicamos el siguiente contenido

```

apiVersion: v1
kind: Pod
metadata:
  name: pod-secret-with-env-variables
spec:
  containers:
    - name: nginx-container
      image: nginx
    env:
      - name: SECRET_USERNAME
        valueFrom:
          secretKeyRef:
            name: db-access
            key: username.txt
      - name: SECRET_PASSWORD
        valueFrom:
          secretKeyRef:
            name: db-access
            key: password.txt
  restartPolicy: Never

```



Fondos Europeos



Ejecutamos la orden de creación del Pod

```
$ kubectl apply -f pod-secret-with-env-variables.yml
pod/pod-secret-with-env-variables created
```

Nos conectamos dentro del contenedor

```
$ kubectl exec -it pod-secret-with-env-variables bash
root@pod-secret-with-env-variables:/#
```



Cofinanciado por
la Unión Europea



Ejecutamos instrucciones echo para visualizar el valor de los secretos

```
root@pod-secret-with-env-variables:/# echo $SECRET_USERNAME
root
root@pod-secret-with-env-variables:/#

root@pod-secret-with-env-variables:/# echo $SECRET_PASSWORD
my-perfect-password
root@pod-secret-with-env-variables:/#
```

42.15. Limitaciones de los secretos/seuridad

- Los secretos deben de ser creados antes de que se cree cualquier Pod que dependa de dicho

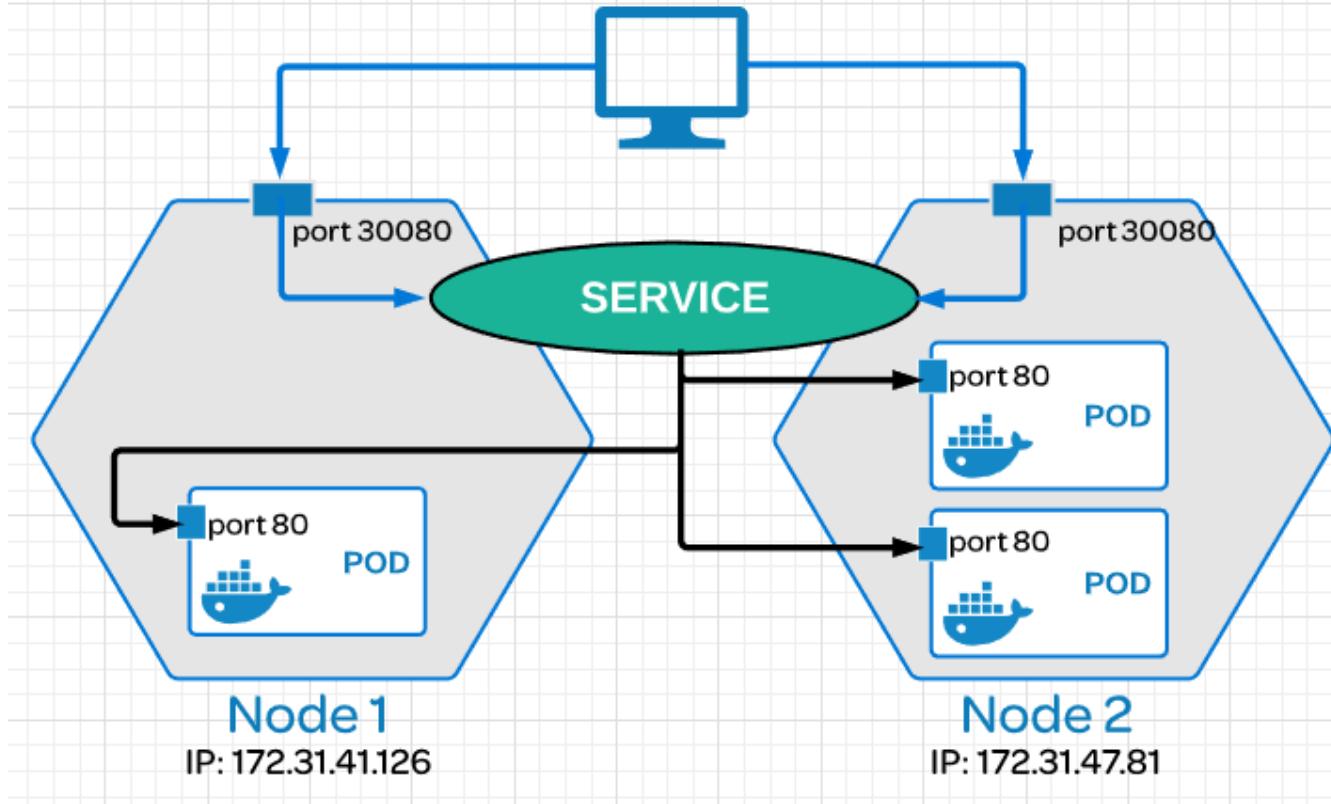
secreto

- Los secretos son objetos que residen en un namespace concreto, sólo puede ser referenciado por Pods que se encuentren en el mismo namespace
- Existe una limitación de 1MB de tamaño por cada unidad secret que se crea, como alternativa se pueden crear múltiples secretos pequeños
- Si referenciamos secretos mediante la técnica de variables de entorno con **secretKeyRef** y esas claves no existen en el secreto, el Pod no arrancará
- Kubernetes sólo envía el secreto a un nodo cuyo Pods lo necesiten usar
- Kubernetes almacena los secretos en una memoria temporal **tmpfs**, por lo que los secretos no son almacenados localmente en disco
- Una vez el Pod del que depende el secreto ha sido destruido, el agente Kubelet procederá a realizar la destrucción del secreto en el nodo en cuestión

Capítulo 43. Services

Kubernetes Service

A service allows you to dynamically access a group of replica pods.

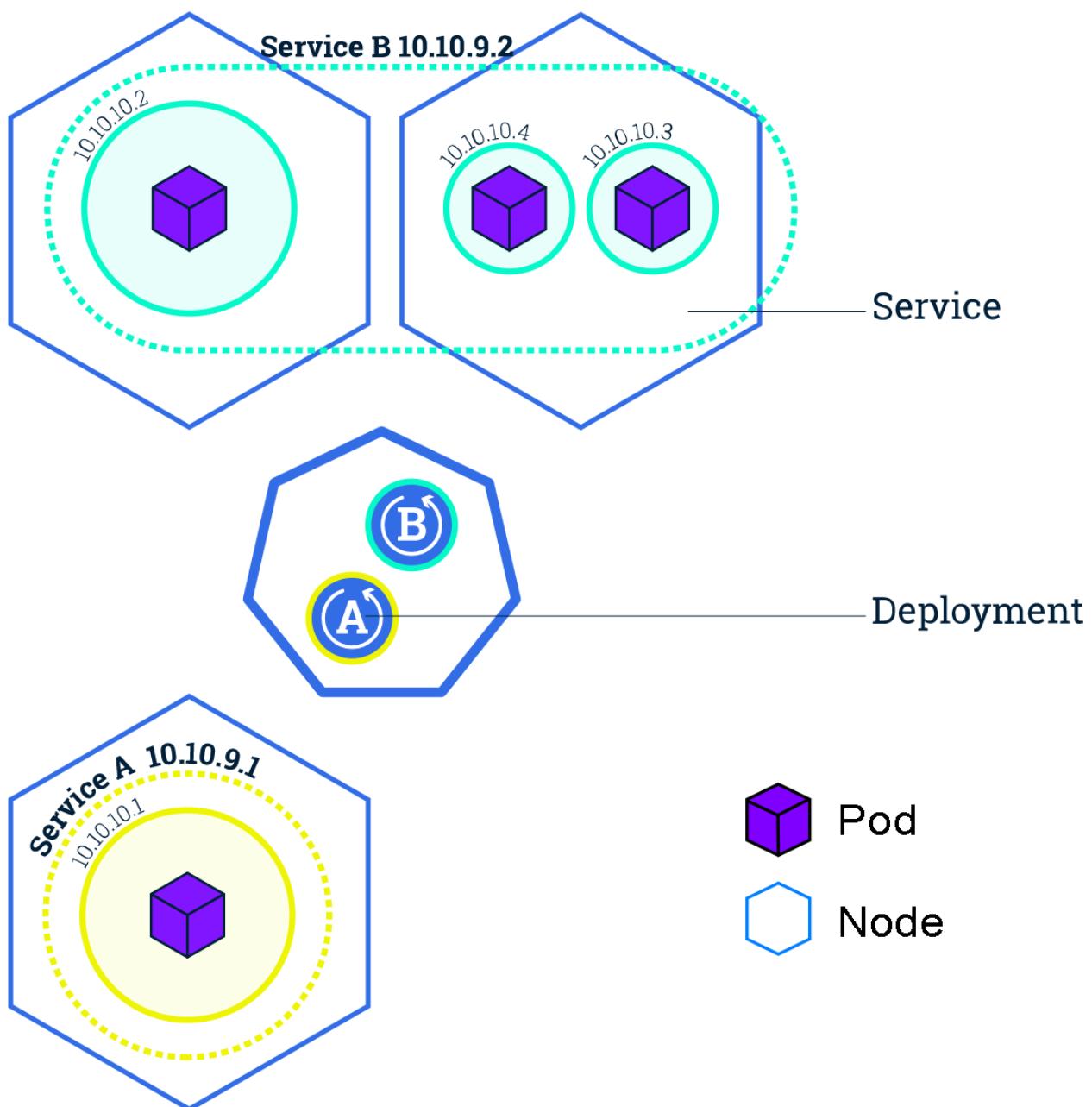


- Es la forma de exponer los pods.
- La forma de poder acceder a la aplicación puede ser
 - Desde dentro del cluster
 - Desde fuera del cluster
- Services se encarga de esto.
- Se trata de objetos Rest del API de k8s
- Los pods creados desde el ReplicationController no pueden acceder directamente, y aunque pudieramos, solo apuntaríamos a un pod, y al caerse deberíamos apuntar a otro Pod.
- Con un servicio, podemos indicar un solo endpoint que haga de enrutador a nuestros pods, balanceando las peticiones y redirigiendo en caso de fallo.
- las direcciones del servicio no cambian nunca.
- Los pods acceden a través del nombre del servicio a los pods asignados al mismo
- Un servicio posee una ip, un nombre dns y un puerto
- El puerto es a nivel de todo el cluster, y es la forma de conectar desde el exterior.
- Al crear el servicio, se crea un endpoint para cada pod conectado, y se mantiene actualizado según el estado deseado.

Cuando hablamos de un servicio, podemos imaginarnos como una entidad que envuelve al propio pod, de forma que proporcionará adicionalmente una IP de acceso que será fija, para poder acceder a dicho Pod

Realmente un servicio a bajo nivel es una agrupación lógica del par IP/Puerto, pero no hay realmente una interfaz de red a bajo nivel que responda ese par, si intentamos hacerle por ejemplo un ping, posiblemente no funcionará :)

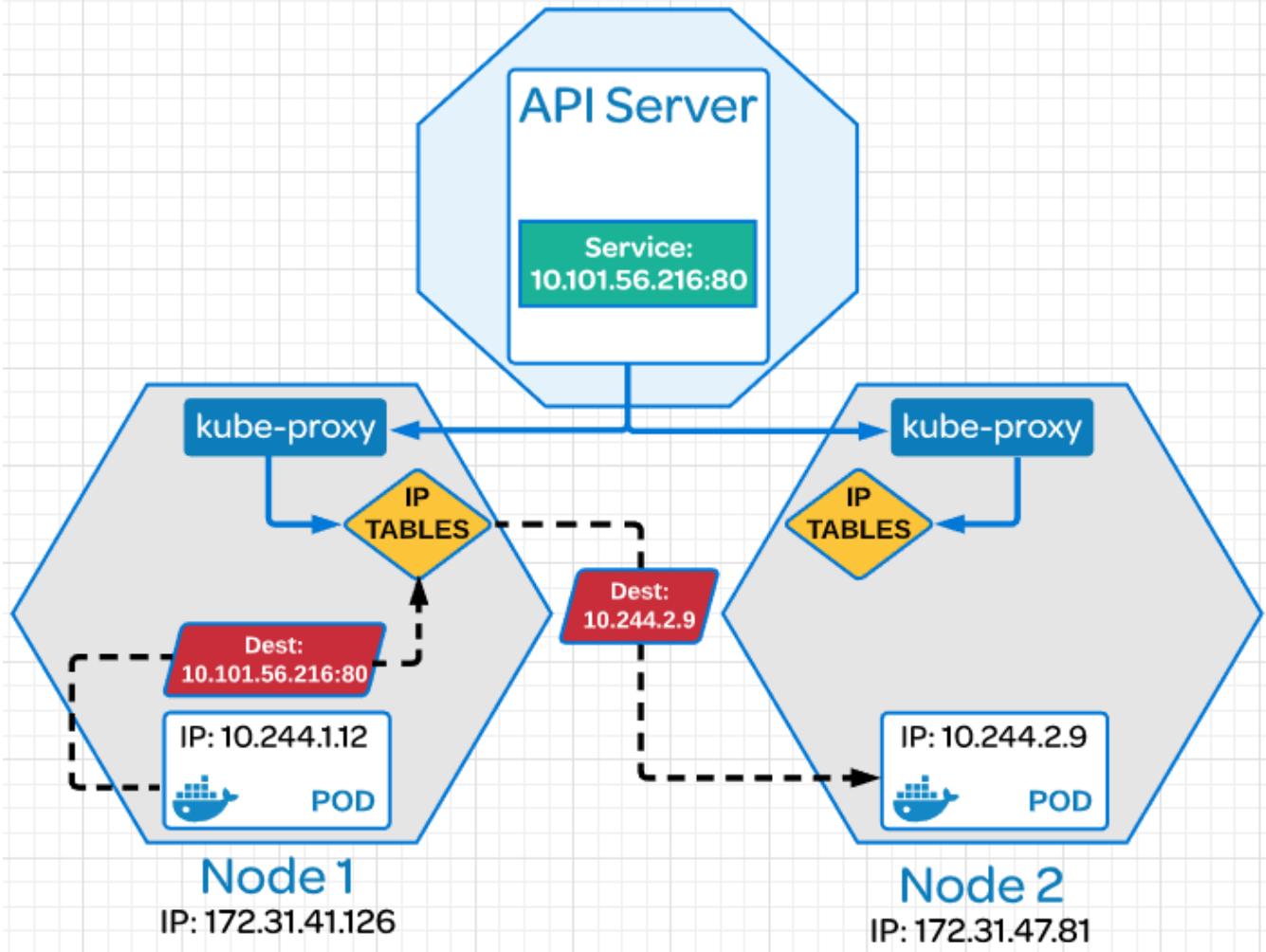
Cuando se lleven a cabo maniobras de escalado de réplicas, imaginemos que ahora, escalamos a 3 réplicas, lo que era antes un pod atendido por el servicio, ahora pasan a ser por ejemplo 3, el servicio actuará en consonancia, de forma que será capáz de detectar las nuevas réplicas que van apareciendo en el clúster y haciéndolas suyas



43.1. Arquitectura del servicio

kube-proxy

kube-proxy handles the traffic associated with a service by creating iptables rules.



El componente del sistema **kube-proxy** maneja todo el tráfico asociado con el servicio

Los servicios en sí mismos consisten en un par IP y Puerto

La IP asociada al servicio es en esencia una IP virtual de red, lo que significa que no tiene una NIC (Network Interface Controller), tarjeta de red física asociada

Cuando el servicio se crea en el componente del sistema **kube-api-server**, la dirección IP virtual es asignada al servicio de forma inmediata, acto seguido, el **kube-api-server** notifica al componente **kube-proxy** para que estos mantengan las tablas de enrutamiento de red actualizadas

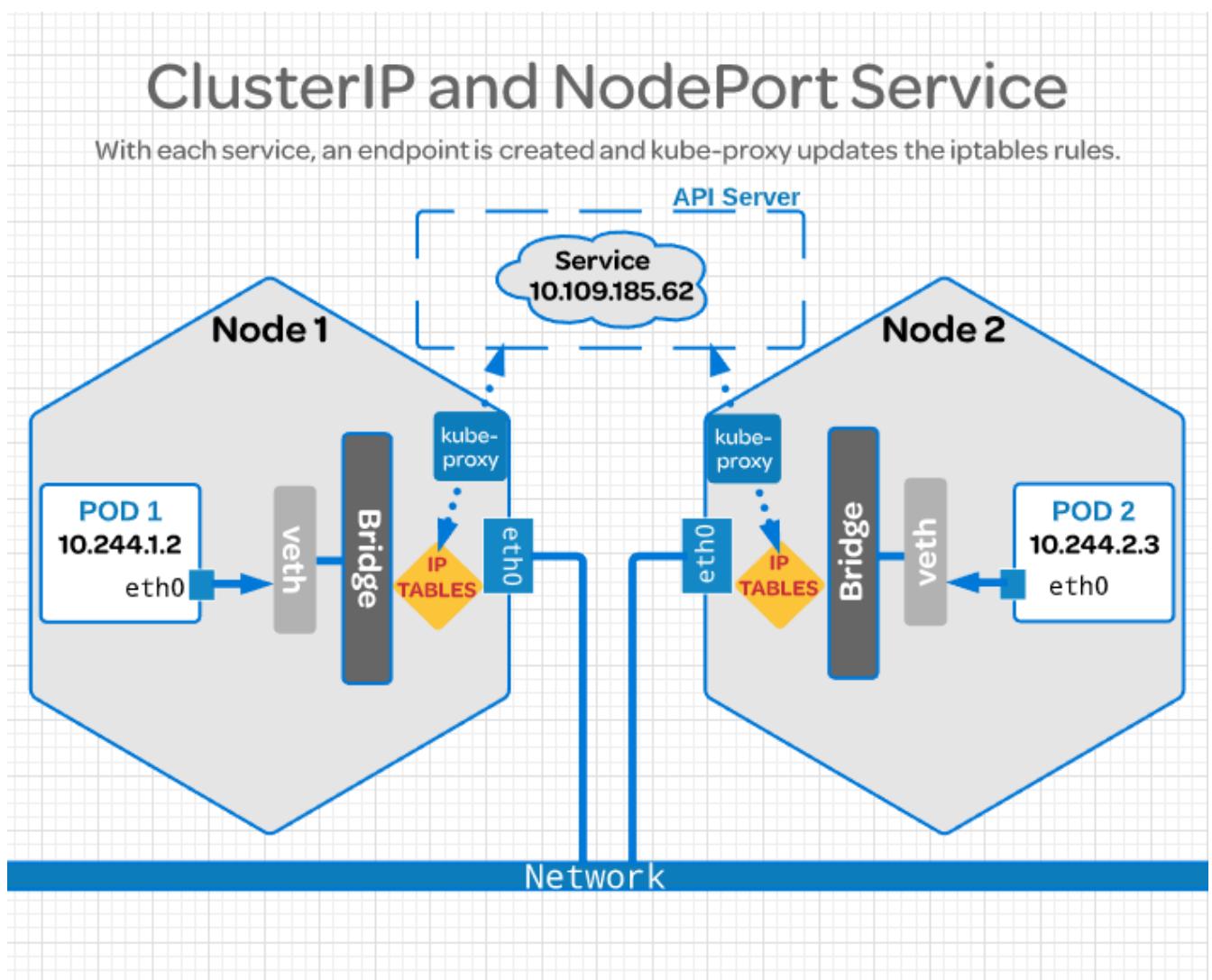
El componente **kube-proxy** se asegura de que el servicio es interceptable, asegurándose de que los paquetes de red llegan a su destino correcto

Cuando las tablas de enrutamiento son creadas, cuando un Pod intenta alcanzar otro Pod mediante la IP del servicio, las tablas de enrutamiento mantenidas por los **kube-proxy** entran en funcionamiento y la petición de red es resuelta y dirigida finalmente hacia la IP interna correcta

del Pod solicitado

43.2. Tipos de Servicios

Kubernetes tiene básicamente 3 tipos de servicios:



- **ClusterIP**

- Consiste en la exposición del servicio a nivel de IP interna dentro del clúster
- El servicio sólo es alcanzable desde dentro del clúster
- Es el valor por defecto del servicio si no indicamos lo contrario

- **NodePort**

- Expone el servicio en cada IP de cada nodo del clúster en un puerto estático fijo (abre un puerto específico en cada uno de los nodos que componen el clúster)
- Se crea a nivel interno automáticamente un servicio de ClusterIP hacia el cual se enruta el Servicio NodePort
- El servicio puede ser accedido mediante la IP lógica de cada nodo del clúster
- Podemos ponernos en contacto con el servicio desde fuera del clúster de la siguiente forma <NODE_IP>:<NODE_PORT>



- <NODE_IP>

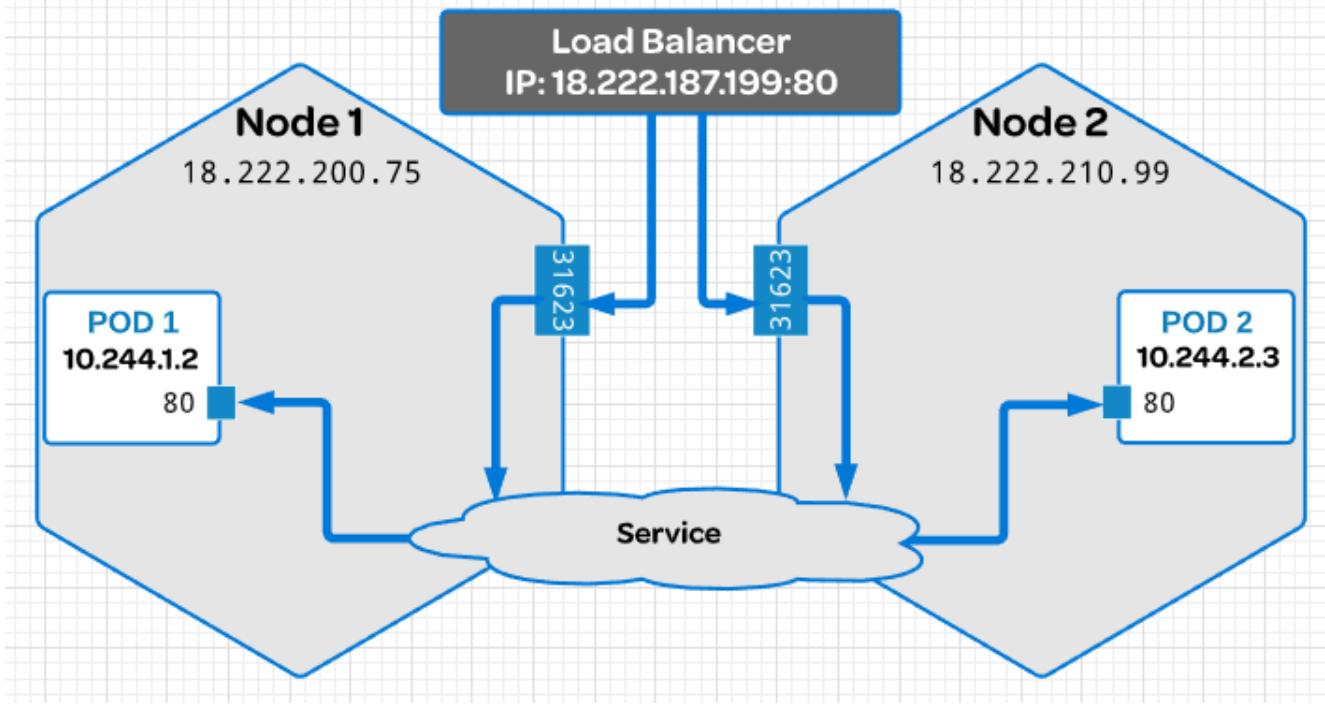
- La ip fija de la tarjeta de red lógica del nodo del clúster de kubernetes (eth0)

- <NODE_PORT>

- El puerto de exposición que hemos indicado al crear el servicio de tipo NodePort

Load Balancers

The load balancer redirects traffic to all the nodes and their node ports. The clients trying to access your application only talk to the load balancer's IP address.



- LoadBalancer

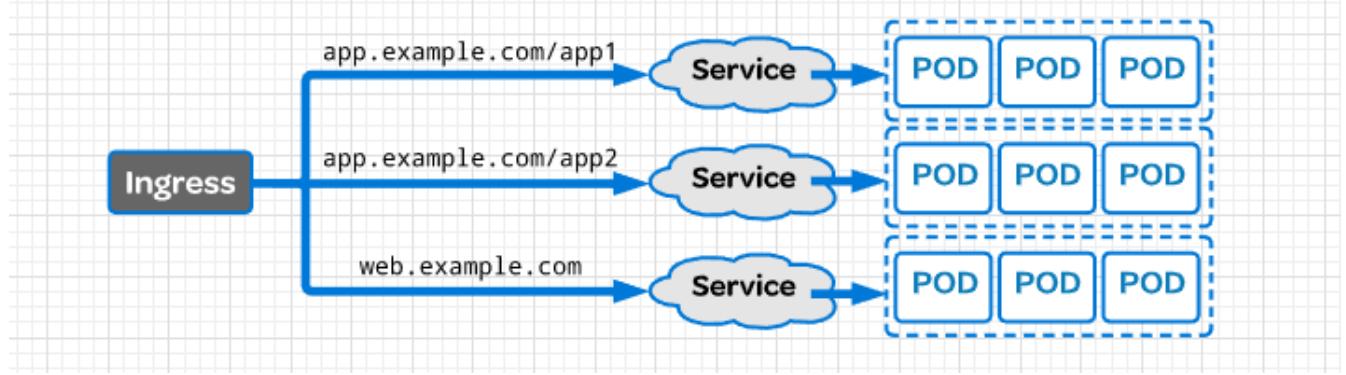
- Expone el Servicio externamente utilizando el balanceador de carga de un proveedor de la nube
- Los servicios NodePort y ClusterIP, a los que se enrutan las rutas externas del balanceador de carga, se crean automáticamente
- Podemos decir, que este tipo de servicio actúa como una extensión del servicio de tipo NodePort
- El balanceador de carga distribuye el tráfico hacia todos los nodos y hacia los puertos de conexión
- Los clientes que intentan acceder a la aplicación, únicamente tienen que comunicarse con la IP del balanceador de carga

Podemos encontrar más información al respecto en la web de la documentación oficial de kubernetes:

43.3. Ingress

Ingress

The load balancer redirects traffic to all the nodes and their node ports. The clients trying to access your application only talk to the load balancer's IP address.



Ingress no está considerado un tipo de servicio, pero actúa como puerta de entrada al clúster

Nos permite consolidar las reglas de enrutamiento en un sólo recurso, puede exponer múltiples servicios bajo la misma dirección IP

El equilibrador de carga redirige el tráfico a todos los nodos y sus puertos de nodo

Los clientes que intentan acceder a su aplicación solo hablan con la dirección IP del equilibrador de carga

El recurso de kubernetes de tipo ingress opera en la capa de aplicación, provee capacidades que los servicios no pueden hacer

De forma conjunta, cuando se crea un recurso de tipo ingress, se crea de forma paralela un controlador ingress

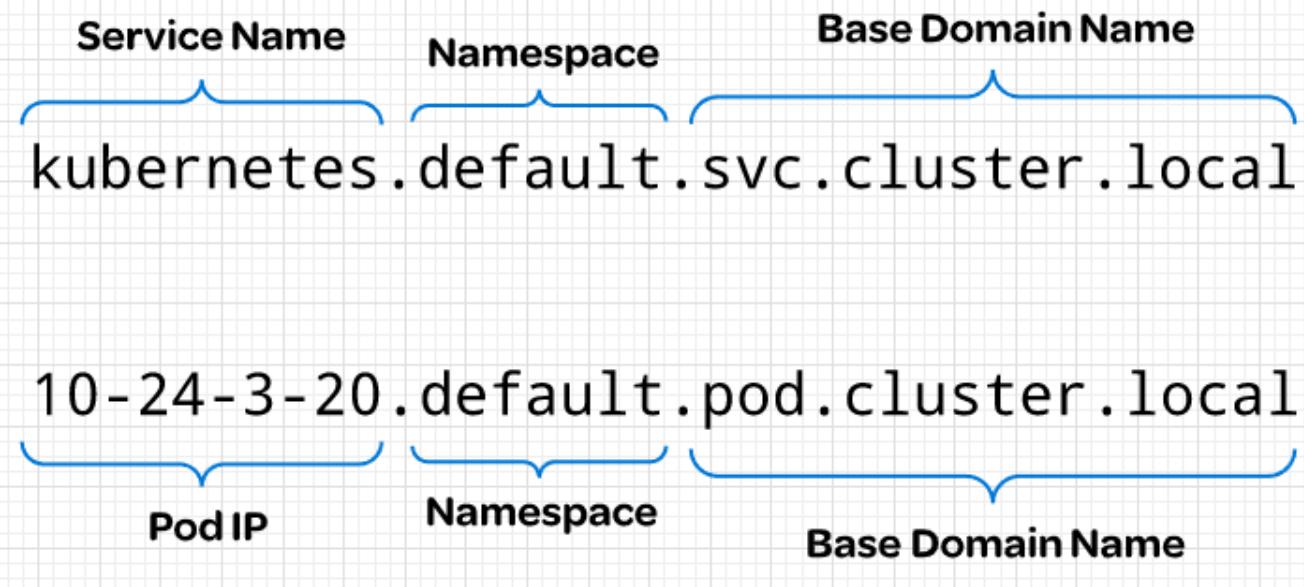
43.4. Labels

- La parte más interesante son las etiquetas.
- El servicio usa las etiquetas para saber que endpoints tiene que balancear.
- Estas etiquetas son dinámicas, lo que permite actualizar a que pods estamos dirigiendo las peticiones.

43.5. Service Discovery DNS

DNS

Every service defined in the cluster is assigned a DNS name. A pod's DNS search list will include the pod's own namespace and the cluster's default domain.



- Se implementa como DNS alojado en el cluster
- Basado en kubelet, para que mapee los nombres dentro de todo el cluster.
- Cada pod posee variables de entorno, y se agregan al crear el pod.
- A cada servicio definido en el clúster se le asigna un nombre DNS.
- La lista de búsqueda de DNS de un pod incluirá el propio espacio de nombres del pod y el dominio predeterminado del clúster.
- Kubernetes soporta entradas DNS a través de TLS o en plano
- Fácilmente integrado en el componente etcd



Si agregamos variables después de crear el pod, este no las poseerá ya que se agregan solo en creación

Capítulo 44. Lab: Services

44.1. Creando el deployment para operaciones

Vamos a crear una unidad deployment que nos permita llevar a cabo operaciones de escalado de los servicios que vamos a desplegar

Creamos el archivo **deployment-for-services.yml** e indicamos el siguiente contenido:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deployment-for-services
  labels:
    department: engineering
spec:
  replicas: 2
  selector:
    matchLabels:
      app: frontend-backend
  template:
    metadata:
      labels:
        app: frontend-backend
    spec:
      containers:
        - name: nginx
          image: nginx:alpine
          ports:
            - containerPort: 80
        - name: tomcat
          image: tomcat:alpine
          ports:
            - containerPort: 8080
```

Ejecutamos la creación:

```
$ kubectl apply -f deployment-for-services.yml
deployment/deployment-for-services created
```

44.2. Creando un servicio de tipo ClusterIP (DHCP)

Un servicio de tipo **ClusterIP** es el servicio por defecto en kubernetes, proporciona un servicio de red, en el ámbito interno del clúster, mediante el cual, otras aplicaciones dentro del clúster pueden acceder al mismo

Con este tipo de servicio, no hay acceso externo

Vamos a dejar que kubernetes asigne una IP del rango que el tiene operando

Creamos el archivo **service-clusterip-dhcp.yml** e indicamos el siguiente contenido:

```
apiVersion: v1
kind: Service
metadata:
  name: service-clusterip-dhcp
spec:
  selector:
    app: frontend-backend
  type: ClusterIP
  ports:
    - name: http
      protocol: TCP
      port: 8181
      targetPort: 80
```

Ejecutamos la creación

```
$ kubectl apply -f service-clusterip-dhcp.yml
```

```
service/service-clusterip-dhcp created
```

- **kind**

- Indicamos como tipo de Objeto kubernetes Service

- **metadata.name**

- Indicamos como nombre del objeto el mismo que el archivo en formato yml

- **spec.selector**

- Indicamos el par **key:value** de los pods al cual hará caso el servicio para hacerlos suyos
 - En nuestro caso, el servicio únicamente atenderá pods que tengan una etiqueta con el par **app:frontend**

- **spec.ports.protocol**

- Protocolo de operación del servicio, puede ser TCP o UDP, en nuestro caso, TCP

- **spec.ports.port**

- Puerto de exposición del servicio, el puerto que utilizaremos para acceder al servicio dentro del clúster

- **spec.ports.targetPort**

- Puerto a nivel interno de operación de los contenedores de los Pods

Ahora, listamos los servicios que están operando en el sistema:

```
$ kubectl get svc
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	18d
service-clusterip-dhcp	ClusterIP	10.96.21.11	<none>	8181/TCP	22s

Observamos que se ha generado un nuevo servicio, con el subrango **10.96.21.11**

Realizamos un curl al servicio desde cualquier nodo y observamos como tenemos respuesta del software nginx

```
$ curl 10.96.21.11:8181
```

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
body {
    width: 35em;
...

```



44.3. Creando un servicio de tipo NodePort (DHCP)

Un servicio de tipo NodePort es la forma más básica de obtener tráfico externo directamente a nuestro servicio

NodePort, como indica el propio nombre, abre un puerto específico en cada uno de los nodos que componen el clúster, de forma que todo el tráfico que se envía a ese puerto, se reenvía al servicio

Creamos el archivo **service-nodeport-dhcp.yml** e indicamos el siguiente contenido:

```
apiVersion: v1
kind: Service
metadata:
  name: service-nodeport-dhcp
spec:
  selector:
    app: frontend-backend
  type: NodePort
  ports:
    - name: http
      protocol: TCP
      port: 8282
      nodePort: 30050
      targetPort: 80
```



Ejecutamos la creación:

```
$ kubectl apply -f service-nodeport-dhcp.yml
```

```
service/service-nodeport-dhcp created
```

- Un servicio **NodePort** tiene dos diferencias clave con un servicio **ClusterIP**
 - La palabra clave type: NodePort que indicamos para tal fin
 - El puerto adicional que indicamos como nodePort, especifica qué puerto abrir en los nodos

Si no especificamos puerto mediante la entrada **nodePort**, el sistema elegirá un puerto aleatorio



Con el uso de este tipo de servicios, es conveniente que la mayoría de las veces kubernetes elija el puerto, ya que podemos llegar a tener bastantes servicios y la gestión manual de puertos disponibles puede llegar a suponernos un esfuerzo de gestión extra como operadores

Sólo podemos tener un servicio por puerto (El socket ya estaría ocupado si intentamos levantar otro en el mismo puerto)



Sólo podemos usar puertos de exposición de alto rango, comprendidos entre 30000 - 32767

Si la dirección IP del nodo cambiara por alguna razón, es nuestra responsabilidad de los sistemas que estén apuntando a dicho par IP:PORT, llevar a cabo el ajuste pertinente

Consultamos los servicios que tenemos disponibles en el sistema en este momento y observamos nuestro servicio de tipo NodePort:

```
$ kubectl get svc
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	19d
service-clusterip-dhcp	ClusterIP	10.96.21.11	<none>	8181/TCP	16h
service-nodeport-dhcp	NodePort	10.96.4.126	<none>	8282:30050/TCP	5s

Ahora, si desde cualquier nodo realizamos un curl mediante el par, IP:PORT interno obtenemos resultado:

```
$ curl 10.96.4.126:8282

<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
body {
```

Y si desde cualquier interfaz lógica de cualquier nodo, hacemos un curl al par, IP lógica de la máquina y puerto de apertura de nodo, también obtendremos resultado, pero ahora, desde una interfaz propia de la máquina:

```
$ curl 192.168.15.101:30050

<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
body {
```

44.4. Creando un servicio de tipo ClusterIP (IP personalizada + binding a interfaz externa)

También tenemos la posibilidad de crear un servicio, indicando nosotros la IP de rango interno de clúster que queremos asignar, así como el binding concreto con cierta interfaz de red que deseemos

Creamos el archivo **service-clusterip-ip-custom-binding-external-ip.yml** e indicamos el siguiente contenido:

```

apiVersion: v1
kind: Service
metadata:
  name: service-clusterip-ip-custom-binding-external-ip
spec:
  selector:
    app: frontend-backend
  clusterIP: 10.96.5.100
  ports:
    - nodePort: 0
      name: http
      protocol: TCP
      port: 8383
      targetPort: 80
  externalIPs:
    - 192.168.15.101

```

- Indicamos de tipo **clusterIP**
- Indicamos una IP fija dentro del rango de asignación de IP interna que tiene nuestro clúster **10.96.5.100**
- Indicamos un binding con una interfaz de red externa, mediante el elemento **externalIPs: 192.168.15.101**
- Indicamos a modo de "truco", una entrada nodePort:0, esto hará la magia :)
- Sólo podremos acceder al servicio, si indicamos el par **192.168.15.101:8383** (estamos cerrando el tráfico a una interfaz concreta del clúster)
- También como es lógico, podremos acceder a través de la IP de red interna del servicio (CLUSTER-IP) **10.96.5.100:8383**

Ejecutamos la creación:

```
$ kubectl apply -f service-clusterip-ip-custom-binding-external-ip.yml
service/service-clusterip-ip-custom-binding-external-ip created
```

Consultamos los servicios que tiene el sistema actualmente:



Fondos Europeos



MINISTERIO
DE EDUCACIÓN,
FORMACIÓN PROFESIONAL
Y DEPORTE

Gobierno de España
Cofinanciado por
la Unión Europea



```
$ kubectl get svc
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP
IP	PORT(S)	AGE	
kubernetes	ClusterIP	10.96.0.1	<none>
443/TCP	19d		
service-clusterip-dhcp	ClusterIP	10.96.21.11	<none>
8181/TCP	16h		
service-clusterip-ip-custom-binding-external-ip	ClusterIP	10.96.5.100	192.168.15.101
.15.101	8383/TCP	5s	
service-nodeport-dhcp	NodePort	10.96.4.126	<none>
8282:30050/TCP	23m		

44.5. Listando los endpoints

Por cada servicio del clúster, tenemos aparejados una serie de endpoints finales, o direcciones IP hacia las que los servicios enrutarán el tráfico.

Un EndPoint es un objeto del kube-api-server, es creado automáticamente por el servicio.

El EndPoint mantiene una caché de las direcciones IP de los Pods que gestiona el servicio

Para comprobar los endpoints que tenemos en el sistema, indicamos el siguiente comando:

```
$ kubectl get endpoints
```

NAME	ENDPOINTS	AGE
kubernetes	192.168.15.100:6443	21d
service-clusterip-dhcp	10.46.0.5:80	3d
service-clusterip-ip-custom-binding-external-ip	10.46.0.5:80	2d7h
service-nodeport-dhcp	10.46.0.5:80	2d8h

Si un servicio tuviera como scope más de un Pod, aparecerían tantas IP's como el servicio atendiera, en nuestro caso, observamos una única IP por servicio, lo cual significa que esos servicios ahora mismo sólo están atendiendo un único pod



Se observa además el propio servicio con nombre kubernetes, que es el responsable de abrir la interfaz segura por el puerto 6443 con el nodo maestro

44.6. Listando las reglas de iptables

También podemos observar las reglas de IP Tables para los servicios que esté usando kubernetes.

Ejecutamos el siguiente comando en la consola:

```
$ sudo iptables-save | grep KUBE
```

...

```
-A KUBE-SERVICES -d 192.168.15.101/32 -p tcp -m comment --comment "default/service-clusterip-ip-custom-binding-external-ip:http external IP" -m tcp --dport 8383 -m addrtype --dst-type LOCAL -j KUBE-SVC-S3OBE4U62ETCHHJG
```

...

Entre otras, podemos observar algunas reglas de destino de paquetes como la que nos aparece

44.7. Creando servicio NodePort mediante exposición en línea (Sin YAML)

Existe la opción (poco recomendada), de poder realizar una exposición del servicio mediante la línea de comandos, sin creación de manifiesto, esto puede servirnos para realizar alguna prueba puntual, pero no para tomarlo como normal, ya que no dejamos constancia mediante ningún tipo de manifiesto de la operación que estamos llevando a cabo :)

Vamos a exponer un servicio, para el deployment que tenemos de este laboratorio, escribimos en la consola el siguiente comando:

```
$ kubectl expose deploy deployment-for-services --name=service-nodeport-withouth-yaml --port=8484 --target-port=80  
--type=NodePort  
service/service-nodeport-withouth-yaml exposed
```

- port: Expone el puerto al host anfitrión
- target-port: Indica el puerto de enlace interno a los contenedores que hay dentro de los pods (los contenedores emiten en el puerto 8080 en este caso)

A continuación, describimos el servicio para observar su contenido:

```
$ kubectl describe service service-nodeport-withouth-yaml
```

```
Name:           service-nodeport-withouth-yaml
Namespace:      default
Labels:         department=engineering
Annotations:    <none>
Selector:       app=frontend-backend
Type:          NodePort
IP:            10.96.196.89
Port:          <unset> 8484/TCP
TargetPort:     80/TCP
NodePort:       <unset> 31021/TCP
Endpoints:     10.46.0.5:80
Session Affinity: None
External Traffic Policy: Cluster
Events:        <none>
```

Las opciones más importantes: * **Type**: NodePort, publicación del puerto a nivel del cluster * **IP**: IP Virtual del servicio * **NodePort**: Puerto del cluster utilizado

Ahora, vamos a hacer un curl al servicio para ver si funciona de forma correcta:

```
$ curl -I kubeminion1:31021
```

```
HTTP/1.1 200 OK
Server: nginx/1.17.6
Content-Type: text/html
Content-Length: 612
Connection: keep-alive
ETag: "5dd406e1-264"
Accept-Ranges: bytes
```

Podemos observar que responde el servicio y que cualquier nodo del clúster responderá a la URL

```
$ curl -I kubeminion2:31021
```

```
HTTP/1.1 200 OK
Server: nginx/1.17.6
Content-Type: text/html
Content-Length: 612
Connection: keep-alive
ETag: "5dd406e1-264"
Accept-Ranges: bytes
```

```
$ curl -I kubemaster:31021
```

```
HTTP/1.1 200 OK
Server: nginx/1.17.6
Content-Type: text/html
Content-Length: 612
Connection: keep-alive
ETag: "5dd406e1-264"
Accept-Ranges: bytes
```

44.8. Eliminando los servicios

Para eliminar cualquier servicio podemos indicar la siguiente instrucción:

Procedemos a eliminar el servicio que hemos creado de forma online:

```
$ kubectl delete service service-nodeport-withouth-yaml
service "service-nodeport-withouth-yaml" deleted
```

Únidamente tenemos que poner al final el nombre del servicio que deseamos eliminar

44.9. Creando un servicio de tipo LoadBalancer

Primero, vamos a listar los servicios que tenemos actualmente en el sistema, y vamos a observar el detalle que a excepción de uno, ninguno tiene la columna rellena de "EXTERNAL-IP":

```
$ kubectl get services
NAME           TYPE      CLUSTER-IP   EXTERNAL-IP   PORT(S)      AGE
kubernetes     ClusterIP  10.96.0.1    <none>        443/TCP     21d
service-clusterip-dhcp   ClusterIP  10.96.21.11   <none>        8181/TCP    3d1h
service-clusterip-ip-custom-binding-external-ip ClusterIP  10.96.5.100  192.168.15.101  8383/TCP    2d8h
service-nodeport-dhcp       NodePort   10.96.4.126   <none>        8282:30050/TCP 2d8h
service-nodeport-withouth-yaml   NodePort   10.96.196.89  <none>        8484:31021/TCP 9m8s
```

Cuando observamos servicios que la columna de EXTERNAL-IP está en <none>, significa que esos servicios no son accesibles desde el exterior del clúster, únicamente son accesibles a nivel interno

En este punto, cada petición que reciba el servicio, la dirigirá contra un pod que encuentre disponible, pero no tenemos capacidad total de balanceo en cuanto al servicio

Para conseguir esa magia de tiempo de caída 0 del servicio si un nodo cae, introducimos un balanceador de carga que apunte a un servicio

Si el clúster está desplegado en un proveedor de cloud, el balanceador de carga puede ser aprovisionado automáticamente, creando un tipo de servicio de balanceador de carga, en lugar de

crear un servicio de tipo NodePort

Creamos el archivo **service-load-balancer.yml** y le agregamos el siguiente contenido:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-loadbalancer
spec:
  type: LoadBalancer
  ports:
    - port: 8585
      targetPort: 80
  selector:
    app: frontend-backend
```

- Especificamos el tipo LoadBalancer
- No tenemos que especificar el NodePort, por que eso lo hará kubernetes por nosotros
- Puerto de operación del balanceador 8585
- Puerto de operación interno de los Pods 80
- Etiqueta selector para hacerle caso a esos Pods app: frontend-backend

Ejecutamos la creación del balanceador:

```
$ kubectl apply -f service-load-balancer.yml
service/nginx-loadbalancer created
```

También podríamos crearlo de forma inline, aunque no es nada recomendable, por que no dejaríamos constancia:

```
$ kubectl expose deployment deployment-for-services --port 8585 --target-port 80 --type LoadBalancer
```

Ahora, visualizamos de nuevo los servicios:

```
$ kubectl get services
NAME           TYPE      CLUSTER-IP   EXTERNAL-IP   PORT(S)      AGE
kubernetes     ClusterIP  10.96.0.1   <none>        443/TCP     21d
nginx-loadbalancer   LoadBalancer  10.96.251.183 <pending>    8585:30858/TCP 3m
service-clusterip-dhcp   ClusterIP  10.96.21.11  <none>        8181/TCP     3d2h
service-clusterip-ip-custom-binding-external-ip ClusterIP  10.96.5.100  192.168.15.101 8383/TCP   2d9h
service-nodeport-dhcp     NodePort   10.96.4.126  <none>        8282:30050/TCP 2d9h
```

Observamos que el servicio de balanceador de carga que tenemos, está en estado <pending>, esto es debido a que aún no hemos configurado ese dato

En el caso de estar operando el clúster en modalidad **PaaS**, con el proveedor de nube, podríamos adquirir una dirección IP, y de forma automática en unos segundos el servicio realizaría el binding en cuestión de forma automática



En nuestro caso, al estar operando el clúster en modalidad **IaaS**, será nuestra responsabilidad indicar una IP externa específica para llevar a cabo dicho binding

Modificamos el archivo **service-load-balancer.yml** y añadimos el siguiente contenido:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-loadbalancer
spec:
  type: LoadBalancer
  ports:
    - port: 8585
      targetPort: 80
  selector:
    app: frontend-backend
  externalIPs:
    - 192.168.15.100
```



Ejecutamos la actualización del balanceador:

```
$ kubectl apply -f service-load-balancer.yml
service/nginx-loadbalancer configured
```



Y ahora, comprobamos si el binding de Ip externa se ha producido correctamente:

```
$ kubectl get services
NAME           TYPE      CLUSTER-IP   EXTERNAL-IP     PORT(S)      AGE
kubernetes     ClusterIP  10.96.0.1    <none>        443/TCP     21d
nginx-loadbalancer   LoadBalancer 10.96.251.183  192.168.15.100  8585:30858/TCP  13m
service-clusterip-dhcp   ClusterIP  10.96.21.11   <none>        8181/TCP     3d2h
service-clusterip-ip-custom-binding-external-ip ClusterIP  10.96.5.100   192.168.15.101  8383/TCP     2d9h
service-nodeport-dhcp       NodePort   10.96.4.126   <none>        8282:30050/TCP  2d9h
```



Comprobamos que la petición se realiza correctamente a través del puerto 8080 y de la IP 192.168.15.100:

```
$ curl -I 192.168.15.100:8585
```

HTTP/1.1 200 OK

Server: nginx/1.17.6

Content-Type: text/html

Content-Length: 612

Connection: keep-alive

ETag: "5dd406e1-264"

Accept-Ranges: bytes



Tenemos que tener en cuenta, que esta forma de generación de un balanceador de carga "On-Premise", en caso de que la IP deje de estar operativa por alguna razón (192.168.15.100), por que el nodo se caiga o algo por el estilo, el balanceador de carga dejaría de operar

44.10. Política de enturado del balanceador de carga

El balanceador de carga, permite indicarle 2 políticas de enrutado:

- **Cluster**

- Es la política predeterminada
- El clúster oculta la IP de origen del cliente y puede causar un segundo salto a otro nodo, deberíamos de tener en este caso una buena distribución de carga general

- **Local**

- El clúster conserva la IP de origen del cliente y evita un segundo salto para los servicios de tipo LoadBalancer y de tipo NodePort, pero con este ajuste corremos el riesgo de propagar tráfico de una forma potencialmente desequilibrada

Vamos a crear el archivo **service-load-balancer-external-traffic-policy.yml** y le agregamos el siguiente contenido:

```
apiVersion: v1
kind: Service
metadata:
  name: service-load-balancer-external-traffic-policy
spec:
  type: LoadBalancer
  ports:
    - port: 8686
      targetPort: 80
  selector:
    app: frontend-backend
  externalIPs:
    - 192.168.15.100
  externalTrafficPolicy: Local
```

Ejecutamos la creación del balanceador:

```
$ kubectl apply -f service-load-balancer-external-traffic-policy.yml  
service/service-load-balancer-external-traffic-policy created
```

44.11. Comprobando el funcionamiento DNS

Podemos observar las entradas DNS de los pods del sistema, los que están operando en el espacio de nombres especial kube-system

Ejecutamos el siguiente comando en consola:

```
$ kubectl get pods --namespace kube-system
```

NAME	READY	STATUS	RESTARTS	AGE
coredns-6955765f44-bzz62	1/1	Running	0	3d6h
coredns-6955765f44-klj6b	1/1	Running	2	21d
etcd-kubemaster	1/1	Running	2	21d
kube-apiserver-kubemaster	1/1	Running	3	21d
kube-controller-manager-kubemaster	1/1	Running	9	21d
kube-proxy-bhf4v	1/1	Running	2	21d
kube-proxy-k7wgh	1/1	Running	0	28h
kube-proxy-q24bh	1/1	Running	4	21d
kube-scheduler-kubemaster	1/1	Running	9	21d
weave-net-52qlp	2/2	Running	6	21d
weave-net-gm5sg	2/2	Running	9	21d
weave-net-hs5qq	2/2	Running	0	28h

Observamos que para el caso por ejemplo, de coredns, hay 2 Pods que se les ha asignado una entrada de nombre específica, podemos ver el deployment que regula los pods coredns ejecutando la siguiente instrucción:

```
$ kubectl get deployment coredns --namespace kube-system
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
coredns	2/2	2	2	21d

También podemos observar el servicio específico kube-dns que tiene operando kubernetes con la IP interna de clúster que tiene asignada, también en el espacio de nombres específico kube-system:

```
$ kubectl get services --namespace kube-system
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kube-dns	ClusterIP	10.96.0.10	<none>	53/UDP,53/TCP,9153/TCP	21d

Vamos a crear un nuevo Pod, creamos el archivo **pod-service-dns.yml** y le agregamos el siguiente contenido:

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-service-dns
  namespace: default
spec:
  containers:
  - image: busybox:1.28.4
    command:
      - sleep
      - "3600"
    imagePullPolicy: IfNotPresent
    name: busybox
  restartPolicy: Always
```

Ejecutamos la creación del Pod:

```
$ kubectl apply -f pod-service-dns.yml
pod/pod-service-dns created
```

Comprobamos la creación del nuevo pod:

```
$ kubectl get pods -o wide
NAME                  READY   STATUS    RESTARTS   AGE     IP          NODE      NOMINATED NODE
READINESS GATES
deployment-for-services-6fdbcb84645-5n79n   2/2   Running   0       37h    10.46.0.5   kubeminion1   <none>
<none>
deployment-for-services-6fdbcb84645-p55gj   2/2   Running   0       3d13h   10.46.0.3   kubeminion1   <none>
<none>
pod-service-dns           1/1   Running   0       16m    10.36.0.0   kubeminion2   <none>
<none>
```

Ahora, vamos a observar el contenido del archivo **resolv.conf** del contenedor que se encuentra en el pod **pod-service-dns**:

```
$ kubectl exec -t pod-service-dns -- cat /etc/resolv.conf
nameserver 10.96.0.10
search default.svc.cluster.local svc.cluster.local cluster.local
options ndots:5
```

Comprobamos un lookup del nombre DNS del servicio nativo de kubernetes:

```
$ kubectl exec -it pod-service-dns -- nslookup kubernetes

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 kubernetes.default.svc.cluster.local
```

Lanzamos ahora un lookup de resolución de nombre DNS, pero con la tripleta que identifica el nombre del pod:

```
$ kubectl exec -ti pod-service-dns -- nslookup 10-36-0-0.default.pod.cluster.local

Server: 10.96.0.10
Address 1: 10.96.0.10 kube-dns.kube-system.svc.cluster.local

Name: 10-36-0-0.default.pod.cluster.local
Address 1: 10.36.0.0 pod-service-dns
```

También podemos resolver cualquier otro pod mediante resolución DNS, ahora, vamos a resolver la DNS de otro pod que no es el pod-service-dns, pero invocando el comando lookup a él:

```
$ kubectl exec -ti pod-service-dns -- nslookup 10-46-0-5.default.pod.cluster.local

Server: 10.96.0.10
Address 1: 10.96.0.10 kube-dns.kube-system.svc.cluster.local

Name: 10-46-0-5.default.pod.cluster.local
Address 1: 10.46.0.5 10-46-0-5.service-load-balancer-external-traffic-policy.default.svc.cluster.local
```

44.12. Ajustando DNS específica

Vamos ahora a crear un Pod, pero especificando la configuración que deseamos en cuanto a las entradas DNS del mismo

En cuanto a la política DNS de un Pod, podemos especificar las siguientes:

- **Default**
 - El Pod hereda la configuración de resolución de nombres del nodo en el que se ejecutan los pods
- **ClusterFirst**
 - Cualquier consulta DNS que no coincida con el sufijo de dominio de clúster configurado, como "www.kubernetes.io", se reenvía al servidor de nombres ascendente heredado del nodo.
 - Los administradores de clúster pueden tener servidores DNS adicionales y de dominio

auxiliar configurados

- **ClusterFirstWithHostNet**

- Para los pods que se ejecutan con hostNetwork, se debe establecer explícitamente su política de DNS

- **None**

- Permite que un Pod ignore la configuración de DNS del entorno de Kubernetes.
- Se supone que todas las configuraciones de DNS deben proporcionarse utilizando el campo dnsConfig en la especificación de pod

Creamos un archivo con el nombre **pod-with-custom-dns.yml** y le agregamos el siguiente contenido:

```
apiVersion: v1
kind: Pod
metadata:
  namespace: default
  name: pod-with-custom-dns
spec:
  containers:
    - name: test
      image: nginx:alpine
  dnsPolicy: "None"
  dnsConfig:
    nameservers:
      - 8.8.8.8
    searches:
      - my-pod.svc.cluster.local
      - my-dns-pod.dns.search.suffix
    options:
      - name: ndots
        value: "2"
      - name: edns0
```

- **dnsPolicy**

- Especificación de la política de DNS

- **dnsConfig.nameservers**

- Especificamos manualmente la lista de servidores DNS que queremos que utilice el Pod

- **dnsConfig.searches**

- Especificamos manualmente los nombres DNS que responderán ante búsquedas
- Se trata de una lista de dominios de búsqueda de DNS para la búsqueda de nombres de host en el Pod.
- Esta propiedad es opcional, cuando se especifica, la lista proporcionada se fusionará con los nombres de dominio de búsqueda base generados a partir de la política de DNS elegida.

- Se eliminan los nombres de dominio duplicados.
- Kubernetes permite como máximo 6 dominios de búsqueda.

- **options**

- Se trata de una lista opcional de objetos donde cada objeto puede tener una propiedad de nombre (obligatorio) y una propiedad de valor (opcional).
- El contenido de esta propiedad se fusionará con las opciones generadas a partir de la política de DNS especificada.
- Se eliminan las entradas duplicadas.

Ejecutamos la creación del pod:

```
$ kubectl apply -f pod-with-custom-dns.yml  
pod/pod-with-custom-dns created
```

Le echamos un vistazo al contenido interno del archivo **/etc/resolv.conf** del propio pod:

```
$ kubectl exec -it pod-with-custom-dns -- cat /etc/resolv.conf  
nameserver 8.8.8.8  
search my-pod.svc.cluster.local my-dns-pod.dns.search.suffix  
options ndots:2 edns0
```

44.13. Creando servicio para acceder a pod con DNS específica de servicio (ClusterIP)

Vamos a crear el siguiente archivo **service-clusterip-for-custom-dns.yml** e indicamos el siguiente contenido:

```
apiVersion: v1  
kind: Service  
metadata:  
  name: service-clusterip-for-custom-dns  
spec:  
  selector:  
    app: frontend-backend  
  type: ClusterIP  
  ports:  
    - name: http  
      protocol: TCP  
      port: 8787  
      targetPort: 80
```

Ejecutamos la creación del servicio:

```
$ kubectl apply -f service-clusterip-for-custom-dns.yml
```

```
service/service-clusterip-for-custom-dns created
```

A continuación, listamos los Pods que tenemos operando en este momento:

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
deployment-for-services-6fdbcb84645-5n79n	2/2	Running	0	39h
deployment-for-services-6fdbcb84645-p55gj	2/2	Running	0	3d15h
pod-service-dns	1/1	Running	2	136m
pod-with-custom-dns	1/1	Running	0	62m

El servicio cubre, por que así lo hemos definido, los Pods con el prefijo **deployment-for...**

Ahora, vamos a arrancar un nuevo Pod al vuelo, para una vez dentro, intentar lanzar un comando de resolución de dns contra el servicio, para poder comunicarnos con los Pods del servicio, pero esta vez, mediante la entrada DNS del propio servicio

Arrancamos la instancia del container:

```
$ kubectl run --generator=run-pod/v1 --rm test-pod -it --image ubuntu bash
```

Instalamos la utilidad que necesitamos:

```
root@test-pod:/# apt-get update && apt-get install host && apt-get install curl -y
```



La utilidad host permite realizar búsquedas en DNSs

Llevamos a cabo la búsqueda DNS y la consola nos informa de la resolución y dirección IP del propio servicio:

```
root@test-pod:/# host service-clusterip-for-custom-dns
```

```
service-clusterip-for-custom-dns.default.svc.cluster.local has address 10.96.206.89
```

Finalmente, realizamos una petición al servicio, a través del puerto que hemos configurado (8787), para comprobar que la petición es devuelva, mediante únicamente el nombre del servicio:

```
$ curl -I service-clusterip-for-custom-dns:8787
```

```
HTTP/1.1 200 OK
Server: nginx/1.17.6
Date: Tue, 07 Jan 2020 11:22:55 GMT
Content-Type: text/html
Content-Length: 612
Last-Modified: Tue, 19 Nov 2019 15:14:41 GMT
Connection: keep-alive
ETag: "5dd406e1-264"
Accept-Ranges: bytes
```

44.14. Uso avanzado

- La verdadera potencia de los servicios se encuentra en las etiquetas.
- Podemos crear aplicaciones en replication controllers distintos con mismas etiquetas y distintas versiones.
- Con indicar al servicio que etiqueta debe tener apuntará a unas, a otras o a todas.
- En otros casos prácticos, podemos apuntar primero a la versión 1.0, luego a la versión 1.1. En caso de que la versión 1.1 de problemas, lo único que debemos hacer es cambiar la etiqueta a la versión 1.0 ya que no estamos destruyendo contenedores o pods, solo redireccionando el tráfico, pudiendo volver a la versión anterior en un tiempo record.

Capítulo 45. Ingress Controller: Nginx

Cuando tenemos un entorno On-Premise con Kubernetes y queremos poner en marcha reglas ingress en nuestra infraestructura, necesitamos utilizar lo que se conoce como un controlador ingress.

Dicho controlador, tendrá la capacidad de enrutar el tráfico en base a la URL que nosotros indiquemos por ejemplo en el navegador.

Debemos de diferenciar los dos conceptos clave:

- **Recurso o regla Ingress**

- Se trata de una mera definición de reglas que indican como se enrutan los servicios, URL o rutas de acceso.

- **Controlador Ingress**

- Se trata de un contenedor que enruta las peticiones hacia los servicios correspondientes en base a la definición del recurso ingress.
- Es una plataforma, como por ejemplo un sistema nginx que hace de proxy inverso



Cuando desplegamos kubernetes con reglas ingress en un entorno cloud, el controlador de ingress así como los平衡adores de carga, son proporcionados por el proveedor de cloud en cuestión, como AWS, Azure, GCE, etc.

Kubernetes no incorpora ningún controlador ingress, todo lo que proporciona es una API propia para leer la definición del recurso ingress.

Eso nos puede parecer poca cosa, pero al final lo que se ha conseguido es que haya una gran cantidad de controladores ingress.

En general cualquier contenedor que ejecute un software capaz de actuar como un proxy inverso (p. ej. nginx) puede actuar de controlador ingress.

¿La ventaja? La configuración del recurso ingress forma parte de la definición de tu aplicación Kubernetes.

45.1. Lab: Ingress Controller (Nginx)

Mediante esta laboratorio, vamos a llevar a cabo la implementación del controlador de ingress nginx.

45.1.1. Creando el espacio de nombres (Namespace)

Los componentes del controlador de reglas de ingress estarán operando bajo un namespace que vamos a crear para tal propósito.

Creamos el archivo **nginx-ingress-controller-namespace.yml** y le añadimos el siguiente contenido:

```
apiVersion: v1
kind: Namespace
metadata:
  name: nginx-ingress
```

Ejecutamos el manifiesto:

```
$ kubectl apply -f nginx-ingress-controller-namespace.yml
namespace/nginx-ingress created
```

45.1.2. Creando la cuenta de servicio (ServiceAccount)

Vamos a crear la cuenta de servicio, servirá para que el proxy inverso nginx se pueda comunicar con los diferentes servicios del clúster.

Creamos el archivo **nginx-ingress-controller-service-account.yml** y le añadimos el siguiente contenido:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: nginx-ingress
  namespace: nginx-ingress
```

Ejecutamos el manifiesto:

```
$ kubectl apply -f nginx-ingress-controller-service-account.yml
serviceaccount/nginx-ingress created
```

45.1.3. Creando el cluster role y realizando el binding (ClusterRole y ClusterRoleBinding)

Creamos el rol a nivel de clúster y su binding correspondiente, para que el controlador ingress pueda operar a nivel de todo el clúster.

Creamos el archivo **nginx-ingress-controller-rbac.yml** y le añadimos el siguiente contenido:

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: nginx-ingress
rules:
- apiGroups:
  - ""
    resources:
    - services
    - endpoints
  verbs:
    - get
    - list
    - watch
- apiGroups:
  - ""
    resources:
    - secrets
  verbs:
    - get
    - list
    - watch
- apiGroups:
  - ""
    resources:
    - configmaps
  verbs:
    - get
    - list
    - watch
    - update
    - create
- apiGroups:
  - ""
    resources:
    - pods
  verbs:
    - list
    - watch
- apiGroups:
  - ""
    resources:
```



Fondos Europeos



MINISTERIO
DE EDUCACIÓN, FORMACIÓN PROFESIONAL
Y DEPORTE

Gobierno de España
Cofinanciado por
la Unión Europea



```

- events
verbs:
- create
- patch
- list
- apiGroups:
- networking.k8s.io
resources:
- ingresses
verbs:
- list
- watch
- get
- apiGroups:
- networking.k8s.io
resources:
- ingresses/status
verbs:
- update
- apiGroups:
- k8s.nginx.org
resources:
- virtualservers
- virtualserverroutes
- globalconfigurations
- transportservers
- policies
verbs:
- list
- watch
- get
- apiGroups:
- k8s.nginx.org
resources:
- virtualservers/status
- virtualserverroutes/status
verbs:
- update
- apiGroups:
- networking.k8s.io
resources:
- ingressclasses
verbs:
- get
---  

kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: nginx-ingress
subjects:
- kind: ServiceAccount

```

```
name: nginx-ingress
namespace: nginx-ingress
roleRef:
  kind: ClusterRole
  name: nginx-ingress
  apiGroup: rbac.authorization.k8s.io
```

Ejecutamos el manifiesto:

```
$ kubectl apply -f nginx-ingress-controller-rbac.yml
clusterrole.rbac.authorization.k8s.io/nginx-ingress created
clusterrolebinding.rbac.authorization.k8s.io/nginx-ingress created
```

45.1.4. Creando el cluster role y realizando el binding para aplicaciones protegidas (SSL)

Creamos el rol a nivel de clúster y su binding correspondiente, para que el controlador ingress pueda operar a nivel de todo el clúster con aplicaciones que vayan aseguradas (SSL).

Creamos el archivo **nginx-ingress-controller-rbac-secure.yml** y le añadimos el siguiente contenido:



Fondos Europeos



MINISTERIO
DE EDUCACIÓN, FORMACIÓN PROFESIONAL
Y DEPORTE

GOBIERNO
DE ESPAÑA
Cofinanciado por
la Unión Europea



```

kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: nginx-ingress-app-protect
rules:
- apiGroups:
  - appprotect.f5.com
resources:
- appolicies
- aplogconfs
verbs:
- "get"
- "watch"
- "list"
---
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: nginx-ingress-app-protect
subjects:
- kind: ServiceAccount
  name: nginx-ingress
  namespace: nginx-ingress
roleRef:
  kind: ClusterRole
  name: nginx-ingress-app-protect
  apiGroup: rbac.authorization.k8s.io

```

Ejecutamos el manifiesto:

```

$ kubectl apply -f nginx-ingress-controller-rbac-secure.yml

clusterrole.rbac.authorization.k8s.io/nginx-ingress-app-protect created
clusterrolebinding.rbac.authorization.k8s.io/nginx-ingress-app-protect created

```

45.1.5. Creando el secreto con el certificado TLS y la clave privada para el nginx (Secret)

Vamos a crear un objeto secret en kubernetes, de forma que contenga el certificado digital del servidor nginx así como su correspondiente clave privada.

Creamos el archivo **nginx-server-secret.yml** y le añadimos el siguiente contenido:

```

apiVersion: v1
kind: Secret
metadata:
  name: default-server-secret
  namespace: nginx-ingress
type: Opaque
data:
  tls.crt:
LS0tLS1CRUdjTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUN2akNDQWFZQ0NRREFPRj0THNhWFhEQU5CZ2txaGtpRzl3MEJBUXNGQURBaE1SOHdIUVIEFR
k8KUjBsT1dFbHVaM0psYzNORGiyNTBjbTlZyKdWeU1CNFhEVEU0TURreE1qRTRNRE16TzvWERUSxpNRGt4TVRFNApNRE16Tzvd0IURWZNQjBHQTR
dUa2RKVGxoSmtZHlaWE56UTi5dWRISnZiR3hsY2pDQ0FTSXdEUVIKCktWklodmNOQVFQkJRQRnZ0VQQRDQ0FRb0NnZ0VCQuwvN2hIUEtFWd
3QIBrMTNpWkt5eTlyQ08KR2xZUXYyK2EzUDF0azIrS3YwVGF5aGRCbDRrcnNuCTzzZm8vWuk1Y2Vhbkw4WGM3U1pyQkVRym9EN2REbWs1Qgo4eF
Wlg0Rm5UZ0VPaStIM2ptTFFxRIBSY1kzVnPazFFeUZBL0jnWljVbkNHZUtGeERSN0tQdGhyCmtqSXVuektURXUyaDU4Tlp0S21ScUjhDewcTNRYzhZ
2FnbmovUWRjc0YYTJnMjB1K1YzDdoZ3krZksKwK4vVUkxQUQ0YzZyM1Ima1ZWUmVhd1lxQvp1WXN2V0RkbW1GNWRwdEMzN011cDBPRUxVTE
IwSAo1TmdPc25NWFJNV1hYVlpINWRxT3ROSRTs3fhZ25Tz1jqQVpQn2MwQjFQU2FqYzJNGZRVXpNQ0F3RUFBEFOCkJna3Foa2IH0Xc
FQWpLb2tRdGRPcEsrTzhbWVpc3lySmjdSJyCvFVY2ZOUitjb0hZVUoKdGhrYnhITFMzR3VBTWI5dm15VExPY2xxeC9aYzJPbIewMEJCLzTb0s
UIVrRwtWcitTTFA3NTdUWgozZWI4dmdPdEduMS9ienM3bzNBaS9kclkrCui5Q2k1S3IPc3FHTG1US2xFaUtOYkcyR1ZyTWxjS0ZYQU80YT3C
TQwV1U3cG9mcGltU1ZmaXFsdKv5YmN3N0NYODF6cFeRuyt1eHRYK2VBZ3V0Nhh3VI5d2IyVXYkelhuZk9HbWhWNThDd1dIQnNa0kxN;
FMSKZUUkk3dkhvQXprTWIzbjAxQjQyWjNrN3RXNQpJUDFmTlpIOFUvOWxiUHNoT21FRFZkdjF5ztvRVJxbStGSis2R0oxeFJGcGZnPT0KLS0t
SVEIGSUNBVEmtLS0tLQo=
  tls.key:
LS0tLS1CRUdjTiBSU0EgUFJjVkfURSBRLRVktLS0tLQpNSUIFcEFjQkFBs0NBuUVBdi91RWM4b1jkMHUvZVJTHNFK1RYZUpckxMMnNjNjGFW.
IRiNHEvCljOcktGMEdYaVN1eE9ycXgrajlnamx4NFJdnhkenRKbXNFukj1Z1B0ME9hVGtIekhvb3FVWmcwZGxmz1dkT0EKUTZMNTdIT10Qz
ZUVVRVUQ4R0jsRINjSvo0b1hFTkhbzsyR3VTTWk2Zk1wTVM3YUhudzFtMapxWkdvRWEzWFNyZEj6eGc2chkcUNIUDICMXl3VmRyYURiUz
4cGszOVFqVUFQaHpxdmRoK1JWC1ZGNGJCaw9CbtVpeTIZTW1hWVhsMm0wTGZzeTzuUTRRdFFzdEdNVWozcGjtdlfmazJBNnljeGRFeFpkZ
MllxcHFDZETCRThCay90elFVTIKcU56cHpoOUJUTXjdjREFRQUjBb0ICQVFDZkIhbXowOhhRVmorNwpLznZJUXQwQ0YzR2MxNld6eDhVnml
UNIL3BzWE9LZIRxT1h1SENyUlp5TnUvZ2IvUUQ4bUFOCmxOMjRZTW10TWRjODg5TEzoTkp3QU5OODjDeTczckM5bzVvUDlkazAvYzRIBjAz
9r1FvYksKMjhMNk0rdHUzUmFqNjd6Vmc2d2szaEhrUopXSzBwv1YrSjdrUKRWYmhDYUZhNk5nMUZNRWxhTlozVdhhUUtqGqpDUDNDe
XWVFIK3NYaTVGM01pYVNBZ1BkQuk3WEh1dXFET1lvMU5PL0joSGt1aVg2QnRtCnorNTZud2pZMy8yUytSRmNBc3JMTnIwMDJZi9oY0Ira
WTY3TWdOTGQ5VV9RU3BDRkYrVm4KM0cyUnhybnhBb0dCQU40U3M0ZV1PU2huMvpQQjdhTUzsY0k2RHR2S2ErTGZTTFy2pOZj1SEp
2RiVWVtBwSekR0WkdlcXZxaHFISy9iTjIyeWjhOU1WMDIRQ0JFTk5jNmtWajjTVhpUWkjVbEx4QzYrCk93Z0wyZhhKendWelU0VC84ajdHall
FvRHfYRG5BYWkyaW5oZU1JWWZHRXFGKzjyQW9HQkFOMVAK0tZl0ls3RWRzRsklQnzbjUis3RmpyeXjpY05iWctQVzUvOXFHawXnY2gr
peDN3QVpGdwpaZC96ZFB2aTBkWEppc1BSzjRMazg5b2pCumpiRmRmc2I5UmjYbyt3TFU4NUhRU2NGMnN5aUFPaTVBRHdVU0FkCm45Y
bit3ZFhtaTZ0OHRpSFRkK3RoVdHkaVpBb0dCQUt6Wis1bG9OOTBtYif4Vvh5YuWkmjFSU9tMGjjcdnsTmVCaWNFSmlzaehYa2xpSVxZ3h
kIKZENFaHFsV01aV0xPb2i2NTNyZgo3aFIMsxM1Zutka3o0aFRVdnpldm9TMHVXcm9Cv2xOVHIGanIrSwHkZnZUc0hOpGdsu3FkbXgySkjh
Q4NmNLcInyNkQrZG8wS05FZzFsL0FvR0FIMkFvdHVFbFNqLzBmRzgrV3hHc1RFV1JqciRNuZrsUjhRWXQKeXjdFA4aDzxtGxKUTRCWGxQ
Ib2pZT2pCQTViYjhibXNVU1BV09NNNEoaf4QnIhbmr2eAphYkJDrkFwY0IvbEg4d1R0alVZYIN5T294ZGt5OEp0ek90ajjhS0Fizhd6NIarWD
VFo5MWpYL3RMcf3TmRKS2tDZ1ICbyt0UzB5Tzj2SWFmk2UwSkN5TGHZvDQ5cTN3Zis2QWVqWGX2WDJ1VnRYejN5QTZnbXo5aCsKcdN1k
UFRR0xhUkcrYINNcjR5dERYbE5ZSndUeThXczNKY3d1StdqZVp2b0ZpbmNvVIVIMwphdmxoTUVCRGYxSjltsDB5cDBwWUNaS2ROdHnvZEZ
tsVvhCS3gyZr6cFE9PQotLS0tLUVORCBSU0EgUFJjVkfURSBRLRVktLS0tLQo=

```

Ejecutamos el manifiesto:

```
$ kubectl apply -f nginx-server-secret.yml

secret/default-server-secret created
```

45.1.6. Creando el objeto de configuración para nginx (ConfigMap)

Si se diera el caso de que tuviéramos que cambiar la configuración del propio nginx, vamos a hacerlo de forma que no tengamos que nosotros regenerar el archivo de configuración del propio nginx.

En tal caso, utilizaremos un objeto de configuración, mediante el cual, si introdujéramos a posteriori cualquier key - value, la maquinaria de kubernetes se encargaría de regenerar dicho archivo.

Creamos el archivo **nginx-configmap.yml** y le añadimos el siguiente contenido:

```
kind: ConfigMap  
apiVersion: v1  
metadata:  
  name: nginx-config  
  namespace: nginx-ingress  
data:  
  client-max-body-size: "0"
```

Customizamos la configuración del propio Nginx mediante el objeto ConfigMap, indicamos el parámetro client-max-body-size a 0, para no limitar el tamaño de las imágenes Docker que subiremos a Nexus.



Si no ajustamos dicho parámetro, por defecto el tamaño máximo de una imagen Docker sería 1m y esto nos daría problemas a la hora de subir

Ejecutamos el manifiesto:

```
$ kubectl apply -f nginx-configmap.yml  
  
configmap/nginx-config created
```

45.1.7. Creando la IngressClass

Mediante la creación de un objeto de tipo IngressClass, estamos creando un punto de referencia de implementación.

Es como indicar por así decirlo "el driver" que vamos a utilizar, y con la creación de este objeto, lo que estamos haciendo es dándolo de alta en el clúster, de manera que cuando posteriormente se vayan a crear las reglas de ingress, estas, especifiquen el "driver" o la ingressClass que vamos a utilizar.

Creamos el archivo **nginx-ingressclass.yml** y le añadimos el siguiente contenido:

```
apiVersion: networking.k8s.io/v1  
kind: IngressClass  
metadata:  
  name: nginx  
spec:  
  controller: nginx.org/ingress-controller
```

Ejecutamos el manifiesto:

```
$ kubectl apply -f nginx-ingressclass.yml  
ingressclass.networking.k8s.io/nginx created
```

Los recursos **VirtualServer** y **VirtualServerRoute** son una nueva configuración de balanceo de carga, introducida en la versión 1.5 de kubernetes como alternativa al recurso Ingress.



Los recursos habilitan casos de uso no compatibles con el recurso Ingress, como la división del tráfico y el enrutamiento avanzado basado en contenido.

Los recursos se implementan como objetos de kubernetes de tipo **CustomResourceDefinition**



45.1.8. Creando el VirtualServer

Creamos el archivo **nginx-vs-definition.yml** y le añadimos el siguiente contenido:

```
apiVersion: apiextensions.k8s.io/v1  
kind: CustomResourceDefinition  
metadata:  
  name: virtualservers.k8s.nginx.org  
spec:  
  group: k8s.nginx.org  
  scope: Namespaced  
  names:  
    kind: VirtualServer  
    plural: virtualservers  
    singular: virtualserver  
    shortNames:  
      - vs  
  preserveUnknownFields: false  
versions:  
- name: v1  
  served: true  
  storage: true  
  subresources:  
    status: {}  
  additionalPrinterColumns:  
- name: State  
  type: string  
  description: Current state of the VirtualServer. If the resource has a valid status,  
    it means it has been validated and accepted by the Ingress Controller.  
  jsonPath: .status.state  
- name: Host  
  type: string  
  jsonPath: .spec.host  
- name: IP  
  type: string  
  jsonPath: .status.externalEndpoints[*].ip  
- name: Ports  
  type: string  
  jsonPath: .status.externalEndpoints[*].ports  
- name: Age  
  type: date  
  jsonPath: .metadata.creationTimestamp  
schema:  
openAPIV3Schema:
```

Fondos Europeos



Cofinanciado por
la Unión Europea



```

description: VirtualServer defines the VirtualServer resource.
type: object
properties:
  apiVersion:
    description: 'APIVersion defines the versioned schema of this representation
      of an object. Servers should convert recognized schemas to the latest
      internal value, and may reject unrecognized values. More info:
https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#resources'
    type: string
  kind:
    description: 'Kind is a string value representing the REST resource this
      object represents. Servers may infer this from the endpoint the client
      submits requests to. Cannot be updated. In CamelCase. More info:
https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#types-kinds'
    type: string
  metadata:
    type: object
  spec:
    description: VirtualServerSpec is the spec of the VirtualServer resource.
    type: object
    properties:
      host:
        type: string
      http-snippets:
        type: string
      ingressClassName:
        type: string
      policies:
        type: array
        items:
          description: PolicyReference references a policy by name and an optional
            namespace.
          type: object
          properties:
            name:
              type: string
            namespace:
              type: string
      routes:
        type: array
        items:
          description: Route defines a route.
          type: object
          properties:
            action:
              description: Action defines an action.
              type: object
              properties:
                pass:
                  type: string
                proxy:
                  description: ActionProxy defines a proxy in an Action.
                  type: object
                  properties:
                    requestHeaders:
                      description: ProxyRequestHeaders defines the request headers
                        manipulation in an ActionProxy.
                      type: object
                      properties:
                        pass:
                          type: boolean
                        set:
                          type: array
                        items:
                          description: Header defines an HTTP Header.
                          type: object
                          properties:

```



```

name:
  type: string
value:
  type: string
responseHeaders:
  description: ProxyRequestHeaders defines the response
  headers manipulation in an ActionProxy.
  type: object
properties:
  add:
    type: array
    items:
      description: Header defines an HTTP Header with
      an optional Always field to use with the add_header
      NGINX directive.
      type: object
      properties:
        always:
          type: boolean
name:
  type: string
value:
  type: string
hide:
  type: array
  items:
    type: string
ignore:
  type: array
  items:
    type: string
pass:
  type: array
  items:
    type: string
rewritePath:
  type: string
upstream:
  type: string
redirect:
  description: ActionRedirect defines a redirect in an Action.
  type: object
properties:
  code:
    type: integer
url:
  type: string
return:
  description: ActionReturn defines a return in an Action.
  type: object
properties:
  body:
    type: string
  code:
    type: integer
  type:
    type: string
errorPages:
  type: array
  items:
    description: ErrorPage defines an ErrorPage in a Route.
    type: object
    properties:
      codes:
        type: array
        items:
          type: integer

```



Fondos Europeos



MINISTERIO
GOBIERNO DE ESPAÑA
DE EDUCACIÓN, FORMACIÓN PROFESIONAL
Y DEPORTE

Cofinanciado por
la Unión Europea



```

redirect:
  description: ErrorPageRedirect defines a redirect for an
    ErrorPage.
  type: object
  properties:
    code:
      type: integer
    url:
      type: string
return:
  description: ErrorPageReturn defines a return for an ErrorPage.
  type: object
  properties:
    body:
      type: string
    code:
      type: integer
  headers:
    type: array
  items:
    description: Header defines an HTTP Header.
    type: object
    properties:
      name:
        type: string
      value:
        type: string
  type:
    type: string
location-snippets:
  type: string
matches:
  type: array
  items:
    description: Match defines a match.
    type: object
    properties:
      action:
        description: Action defines an action.
        type: object
        properties:
          pass:
            type: string
      proxy:
        description: ActionProxy defines a proxy in an Action.
        type: object
        properties:
          requestHeaders:
            description: ProxyRequestHeaders defines the request
              headers manipulation in an ActionProxy.
            type: object
            properties:
              pass:
                type: boolean
              set:
                type: array
              items:
                description: Header defines an HTTP Header.
                type: object
                properties:
                  name:
                    type: string
                  value:
                    type: string
  responseHeaders:
    description: ProxyRequestHeaders defines the response
      headers manipulation in an ActionProxy.

```



Fondos Europeos



MINISTERIO
DE EDUCACIÓN, FORMACIÓN PROFESIONAL
Y DEPORTE

Cofinanciado por
la Unión Europea



```

type: object
properties:
  add:
    type: array
    items:
      description: Header defines an HTTP Header
      with an optional Always field to use with
      the add_header NGINX directive.
    type: object
    properties:
      always:
        type: boolean
      name:
        type: string
      value:
        type: string
  hide:
    type: array
    items:
      type: string
  ignore:
    type: array
    items:
      type: string
  pass:
    type: array
    items:
      type: string
  rewritePath:
    type: string
  upstream:
    type: string
  redirect:
    description: ActionRedirect defines a redirect in an
    Action.
    type: object
    properties:
      code:
        type: integer
      url:
        type: string
  return:
    description: ActionReturn defines a return in an Action.
    type: object
    properties:
      body:
        type: string
      code:
        type: integer
      type:
        type: string
  conditions:
    type: array
    items:
      description: Condition defines a condition in a MatchRule.
      type: object
      properties:
        argument:
          type: string
        cookie:
          type: string
        header:
          type: string
        value:
          type: string
        variable:
          type: string

```

```

splits:
  type: array
  items:
    description: Split defines a split.
    type: object
    properties:
      action:
        description: Action defines an action.
        type: object
        properties:
          pass:
            type: string
          proxy:
            description: ActionProxy defines a proxy in an
              Action.
            type: object
            properties:
              requestHeaders:
                description: ProxyRequestHeaders defines the
                  request headers manipulation in an ActionProxy.
                type: object
                properties:
                  pass:
                    type: boolean
                  set:
                    type: array
                    items:
                      description: Header defines an HTTP
                        Header.
                      type: object
                      properties:
                        name:
                          type: string
                        value:
                          type: string
              responseHeaders:
                description: ProxyRequestHeaders defines the
                  response headers manipulation in an ActionProxy.
                type: object
                properties:
                  add:
                    type: array
                    items:
                      description: Header defines an HTTP
                        Header with an optional Always field
                        to use with the add_header NGINX directive.
                    type: object
                    properties:
                      always:
                        type: boolean
                      name:
                        type: string
                      value:
                        type: string
            hide:
              type: array
              items:
                type: string
            ignore:
              type: array
              items:
                type: string
            pass:
              type: array
              items:
                type: string
            rewritePath:

```



Fondos Europeos



Cofinanciado por
la Unión Europea



```

    type: string
    upstream:
      type: string
  redirect:
    description: ActionRedirect defines a redirect
    in an Action.
    type: object
    properties:
      code:
        type: integer
      url:
        type: string
  return:
    description: ActionReturn defines a return in
    an Action.
    type: object
    properties:
      body:
        type: string
      code:
        type: integer
      type:
        type: string
      weight:
        type: integer
  path:
    type: string
  policies:
    type: array
    items:
      description: PolicyReference references a policy by name and
      an optional namespace.
      type: object
      properties:
        name:
          type: string
        namespace:
          type: string
  route:
    type: string
  splits:
    type: array
    items:
      description: Split defines a split.
      type: object
      properties:
        action:
          description: Action defines an action.
          type: object
          properties:
            pass:
              type: string
            proxy:
              description: ActionProxy defines a proxy in an Action.
              type: object
              properties:
                requestHeaders:
                  description: ProxyRequestHeaders defines the request
                  headers manipulation in an ActionProxy.
                  type: object
                  properties:
                    pass:
                      type: boolean
                    set:
                      type: array
                      items:
                        description: Header defines an HTTP Header.

```



```

type: object
properties:
  name:
    type: string
  value:
    type: string
responseHeaders:
  description: ProxyRequestHeaders defines the response
  headers manipulation in an ActionProxy.
type: object
properties:
  add:
    type: array
    items:
      description: Header defines an HTTP Header
      with an optional Always field to use with
      the add_header NGINX directive.
    type: object
    properties:
      always:
        type: boolean
      name:
        type: string
      value:
        type: string
  hide:
    type: array
    items:
      type: string
  ignore:
    type: array
    items:
      type: string
  pass:
    type: array
    items:
      type: string
rewritePath:
  type: string
upstream:
  type: string
redirect:
  description: ActionRedirect defines a redirect in an
  Action.
  type: object
  properties:
    code:
      type: integer
    url:
      type: string
return:
  description: ActionReturn defines a return in an Action.
  type: object
  properties:
    body:
      type: string
    code:
      type: integer
    type:
      type: string
  weight:
    type: integer
server-snippets:
  type: string
tls:
  description: TLS defines TLS configuration for a VirtualServer.
  type: object

```



Fondos Europeos

MINISTERIO
DE EDUCACIÓN,
FORMACIÓN PROFESIONAL
Y DEPORTE

Gobierno
de España

Cofinanciado por
la Unión Europea



```

properties:
redirect:
description: TLSRedirect defines a redirect for a TLS.
type: object
properties:
basedOn:
type: string
code:
type: integer
enable:
type: boolean
secret:
type: string
upstreams:
type: array
items:
description: Upstream defines an upstream.
type: object
properties:
buffer-size:
type: string
buffering:
type: boolean
buffers:
description: UpstreamBuffers defines Buffer Configuration for
an Upstream.
type: object
properties:
number:
type: integer
size:
type: string
client-max-body-size:
type: string
connect-timeout:
type: string
fail-timeout:
type: string
healthCheck:
description: HealthCheck defines the parameters for active Upstream
HealthChecks.
type: object
properties:
connect-timeout:
type: string
enable:
type: boolean
fails:
type: integer
headers:
type: array
items:
description: Header defines an HTTP Header.
type: object
properties:
name:
type: string
value:
type: string
interval:
type: string
jitter:
type: string
passes:
type: integer
path:
type: string

```

```

port:
  type: integer
read-timeout:
  type: string
send-timeout:
  type: string
statusMatch:
  type: string
tls:
  description: UpstreamTLS defines a TLS configuration for an
    Upstream.
  type: object
  properties:
    enable:
      type: boolean
keepalive:
  type: integer
lb-method:
  type: string
max-conns:
  type: integer
max-fails:
  type: integer
name:
  type: string
next-upstream:
  type: string
next-upstream-timeout:
  type: string
next-upstream-tries:
  type: integer
port:
  type: integer
queue:
  description: UpstreamQueue defines Queue Configuration for an
    Upstream.
  type: object
  properties:
    size:
      type: integer
    timeout:
      type: string
read-timeout:
  type: string
send-timeout:
  type: string
service:
  type: string
sessionCookie:
  description: SessionCookie defines the parameters for session
    persistence.
  type: object
  properties:
    domain:
      type: string
    enable:
      type: boolean
    expires:
      type: string
    httpOnly:
      type: boolean
    name:
      type: string
    path:
      type: string
    secure:
      type: boolean

```



Fondos Europeos



MINISTERIO
GOBIERNO DE ESPAÑA
DE EDUCACIÓN, FORMACIÓN PROFESIONAL
Y DEPORTE

Cofinanciado por
la Unión Europea



```

slow-start:
  type: string
subselector:
  type: object
additionalProperties:
  type: string
tls:
  description: UpstreamTLS defines a TLS configuration for an Upstream.
  type: object
  properties:
    enable:
      type: boolean
status:
  description: VirtualServerStatus defines the status for the VirtualServer
  resource.
  type: object
  properties:
    externalEndpoints:
      type: array
      items:
        description: ExternalEndpoint defines the IP and ports used to connect
        to this resource.
        type: object
        properties:
          ip:
            type: string
          ports:
            type: string
    message:
      type: string
    reason:
      type: string
    state:
      type: string

```

Ejecutamos el manifiesto:

```
$ kubectl apply -f nginx-vs-definition.yml
customresourcedefinition.apirextensions.k8s.io/virtualservers.k8s.nginx.org created
```

45.1.9. Creando el VirtualServerRoute

Ahora, toca el turno de crear el VirtualServerRoute.

Creamos el archivo **nginx-vsr-definition.yml** y le añadimos el siguiente contenido:

```

apiVersion: apirextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: virtualserverroutes.k8s.nginx.org
spec:
  group: k8s.nginx.org
  scope: Namespaced
  names:
    kind: VirtualServerRoute
    plural: virtualserverroutes
    singular: virtualserverroute
    shortNames:

```

```

- vsr
preserveUnknownFields: false
versions:
- name: v1
  served: true
  storage: true
  subresources:
    status: {}
  additionalPrinterColumns:
- name: State
  type: string
  description: Current state of the VirtualServerRoute. If the resource has a valid
    status, it means it has been validated and accepted by the Ingress Controller.
  jsonPath: .status.state
- name: Host
  type: string
  jsonPath: .spec.host
- name: IP
  type: string
  jsonPath: .status.externalEndpoints[*].ip
- name: Ports
  type: string
  jsonPath: .status.externalEndpoints[*].ports
- name: Age
  type: date
  jsonPath: .metadata.creationTimestamp
schema:
openAPIV3Schema:
  type: object
  properties:
    apiVersion:
      description: 'APIVersion defines the versioned schema of this representation
        of an object. Servers should convert recognized schemas to the latest
        internal value, and may reject unrecognized values. More info:
https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#resources'
      type: string
    kind:
      description: 'Kind is a string value representing the REST resource this
        object represents. Servers may infer this from the endpoint the client
        submits requests to. Cannot be updated. In CamelCase. More info:
https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#types-kinds'
      type: string
    metadata:
      type: object
    spec:
      type: object
      properties:
        host:
          type: string
        ingressClassName:
          type: string
        subroutes:
          type: array
          items:
            description: Route defines a route.
            type: object
            properties:
              action:
                description: Action defines an action.
                type: object
                properties:
                  pass:
                    type: string
                  proxy:
                    description: ActionProxy defines a proxy in an Action.
                    type: object
                    properties:

```

```

requestHeaders:
  description: ProxyRequestHeaders defines the request headers
    manipulation in an ActionProxy.
  type: object
  properties:
    pass:
      type: boolean
    set:
      type: array
      items:
        description: Header defines an HTTP Header.
        type: object
        properties:
          name:
            type: string
          value:
            type: string
  responseHeaders:
    description: ProxyRequestHeaders defines the response
      headers manipulation in an ActionProxy.
    type: object
    properties:
      add:
        type: array
        items:
          description: Header defines an HTTP Header with
            an optional Always field to use with the add_header
            NGINX directive.
          type: object
          properties:
            always:
              type: boolean
            name:
              type: string
            value:
              type: string
      hide:
        type: array
        items:
          type: string
      ignore:
        type: array
        items:
          type: string
    pass:
      type: array
      items:
        type: string
  rewritePath:
    type: string
  upstream:
    type: string
  redirect:
    description: ActionRedirect defines a redirect in an Action.
    type: object
    properties:
      code:
        type: integer
      url:
        type: string
  return:
    description: ActionReturn defines a return in an Action.
    type: object
    properties:
      body:
        type: string
      code:

```

```

    type: integer
    type:
    type: string
  errorPages:
    type: array
    items:
      description: ErrorCode defines an ErrorCode in a Route.
      type: object
      properties:
        codes:
          type: array
          items:
            type: integer
        redirect:
          description: ErrorPageRedirect defines a redirect for an
            ErrorCode.
          type: object
          properties:
            code:
              type: integer
            url:
              type: string
  return:
    description: ErrorCodeReturn defines a return for an ErrorCode.
    type: object
    properties:
      body:
        type: string
      code:
        type: integer
      headers:
        type: array
        items:
          description: Header defines an HTTP Header.
          type: object
          properties:
            name:
              type: string
            value:
              type: string
      type:
        type: string
  location-snippets:
    type: string
  matches:
    type: array
    items:
      description: Match defines a match.
      type: object
      properties:
        action:
          description: Action defines an action.
          type: object
          properties:
            pass:
              type: string
            proxy:
              description: ActionProxy defines a proxy in an Action.
              type: object
              properties:
                requestHeaders:
                  description: ProxyRequestHeaders defines the request
                    headers manipulation in an ActionProxy.
                  type: object
                  properties:
                    pass:
                      type: boolean

```



Fondos Europeos



Cofinanciado por
la Unión Europea



```

set:
  type: array
  items:
    description: Header defines an HTTP Header.
    type: object
    properties:
      name:
        type: string
      value:
        type: string
  responseHeaders:
    description: ProxyRequestHeaders defines the response
    headers manipulation in an ActionProxy.
    type: object
    properties:
      add:
        type: array
        items:
          description: Header defines an HTTP Header
          with an optional Always field to use with
          the add_header NGINX directive.
        type: object
        properties:
          always:
            type: boolean
          name:
            type: string
          value:
            type: string
      hide:
        type: array
        items:
          type: string
      ignore:
        type: array
        items:
          type: string
      pass:
        type: array
        items:
          type: string
  rewritePath:
    type: string
  upstream:
    type: string
  redirect:
    description: ActionRedirect defines a redirect in an
    Action.
    type: object
    properties:
      code:
        type: integer
      url:
        type: string
  return:
    description: ActionReturn defines a return in an Action.
    type: object
    properties:
      body:
        type: string
      code:
        type: integer
      type:
        type: string
  conditions:
    type: array
    items:

```



Fondos Europeos



Cofinanciado por
la Unión Europea



```

description: Condition defines a condition in a MatchRule.
type: object
properties:
  argument:
    type: string
  cookie:
    type: string
  header:
    type: string
  value:
    type: string
  variable:
    type: string
splits:
  type: array
  items:
    description: Split defines a split.
    type: object
    properties:
      action:
        description: Action defines an action.
        type: object
        properties:
          pass:
            type: string
          proxy:
            description: ActionProxy defines a proxy in an
              Action.
            type: object
            properties:
              requestHeaders:
                description: ProxyRequestHeaders defines the
                  request headers manipulation in an ActionProxy.
                type: object
                properties:
                  pass:
                    type: boolean
                  set:
                    type: array
                    items:
                      description: Header defines an HTTP
                        Header.
                      type: object
                      properties:
                        name:
                          type: string
                        value:
                          type: string
            responseHeaders:
              description: ProxyRequestHeaders defines the
                response headers manipulation in an ActionProxy.
              type: object
              properties:
                add:
                  type: array
                  items:
                    description: Header defines an HTTP
                      Header with an optional Always field
                      to use with the add_header NGINX directive.
                    type: object
                    properties:
                      always:
                        type: boolean
                      name:
                        type: string
                      value:
                        type: string

```



Fondos Europeos

MINISTERIO
DE EDUCACIÓN,
FORMACIÓN PROFESIONAL
Y DEPORTE



Cofinanciado por
la Unión Europea



```

    hide:
      type: array
      items:
        type: string
    ignore:
      type: array
      items:
        type: string
    pass:
      type: array
      items:
        type: string
  rewritePath:
    type: string
  upstream:
    type: string
  redirect:
    description: ActionRedirect defines a redirect
    in an Action.
    type: object
    properties:
      code:
        type: integer
      url:
        type: string
  return:
    description: ActionReturn defines a return in
    an Action.
    type: object
    properties:
      body:
        type: string
      code:
        type: integer
      type:
        type: string
      weight:
        type: integer
  path:
    type: string
  policies:
    type: array
    items:
      description: PolicyReference references a policy by name and
      an optional namespace.
      type: object
      properties:
        name:
          type: string
        namespace:
          type: string
  route:
    type: string
  splits:
    type: array
    items:
      description: Split defines a split.
      type: object
      properties:
        action:
          description: Action defines an action.
          type: object
          properties:
            pass:
              type: string
            proxy:
              description: ActionProxy defines a proxy in an Action.

```



Fondos Europeos



MINISTERIO
DE EDUCACIÓN, FORMACIÓN PROFESIONAL
Y DEPORTE

Cofinanciado por
la Unión Europea



```

type: object
properties:
  requestHeaders:
    description: ProxyRequestHeaders defines the request
      headers manipulation in an ActionProxy.
    type: object
    properties:
      pass:
        type: boolean
      set:
        type: array
        items:
          description: Header defines an HTTP Header.
        type: object
        properties:
          name:
            type: string
          value:
            type: string
    responseHeaders:
      description: ProxyRequestHeaders defines the response
        headers manipulation in an ActionProxy.
      type: object
      properties:
        add:
          type: array
          items:
            description: Header defines an HTTP Header
              with an optional Always field to use with
              the add_header NGINX directive.
        type: object
        properties:
          always:
            type: boolean
          name:
            type: string
          value:
            type: string
      hide:
        type: array
        items:
          type: string
      ignore:
        type: array
        items:
          type: string
      pass:
        type: array
        items:
          type: string
  rewritePath:
    type: string
  upstream:
    type: string
  redirect:
    description: ActionRedirect defines a redirect in an
      Action.
    type: object
    properties:
      code:
        type: integer
      url:
        type: string
  return:
    description: ActionReturn defines a return in an Action.
    type: object
    properties:

```



```

body:
  type: string
code:
  type: integer
type:
  type: string
weight:
  type: integer
upstreams:
type: array
items:
  description: Upstream defines an upstream.
type: object
properties:
  buffer-size:
    type: string
buffering:
  type: boolean
buffers:
  description: UpstreamBuffers defines Buffer Configuration for
  an Upstream.
type: object
properties:
  number:
    type: integer
size:
  type: string
client-max-body-size:
  type: string
connect-timeout:
  type: string
fail-timeout:
  type: string
healthCheck:
  description: HealthCheck defines the parameters for active Upstream
  HealthChecks.
type: object
properties:
  connect-timeout:
    type: string
  enable:
    type: boolean
fails:
  type: integer
headers:
  type: array
items:
  description: Header defines an HTTP Header.
  type: object
  properties:
    name:
      type: string
    value:
      type: string
interval:
  type: string
jitter:
  type: string
passes:
  type: integer
path:
  type: string
port:
  type: integer
read-timeout:
  type: string
send-timeout:

```

```

type: string
statusMatch:
  type: string
tls:
  description: UpstreamTLS defines a TLS configuration for an
    Upstream.
  type: object
  properties:
    enable:
      type: boolean
keepalive:
  type: integer
lb-method:
  type: string
max-conns:
  type: integer
max-fails:
  type: integer
name:
  type: string
next-upstream:
  type: string
next-upstream-timeout:
  type: string
next-upstream-tries:
  type: integer
port:
  type: integer
queue:
  description: UpstreamQueue defines Queue Configuration for an
    Upstream.
  type: object
  properties:
    size:
      type: integer
    timeout:
      type: string
read-timeout:
  type: string
send-timeout:
  type: string
service:
  type: string
sessionCookie:
  description: SessionCookie defines the parameters for session
    persistence.
  type: object
  properties:
    domain:
      type: string
    enable:
      type: boolean
    expires:
      type: string
    httpOnly:
      type: boolean
    name:
      type: string
    path:
      type: string
    secure:
      type: boolean
slow-start:
  type: string
subselector:
  type: object
additionalProperties:

```



Fondos Europeos

MINISTERIO
DE EDUCACIÓN, FORMACIÓN PROFESIONAL
Y DEPORTE

Cofinanciado por
la Unión Europea



```

type: string
tls:
  description: UpstreamTLS defines a TLS configuration for an Upstream.
  type: object
  properties:
    enable:
      type: boolean
status:
  description: VirtualServerRouteStatus defines the status for the VirtualServerRoute
  resource.
  type: object
  properties:
    externalEndpoints:
      type: array
      items:
        description: ExternalEndpoint defines the IP and ports used to connect
        to this resource.
        type: object
        properties:
          ip:
            type: string
          ports:
            type: string
    message:
      type: string
    reason:
      type: string
    referencedBy:
      type: string
    state:
      type: string
  
```

Ejecutamos el manifiesto:

```
$ kubectl apply -f nginx-vsr-definition.yml
customresourcedefinition.apiextensions.k8s.io/virtualserverroutes.k8s.nginx.org created
```

45.1.10. Creando el TransportServer

Creamos el archivo **nginx-ts-definition.yml** y le añadimos el siguiente contenido:

```

apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: transportservers.k8s.nginx.org
spec:
  group: k8s.nginx.org
  scope: Namespaced
  names:
    plural: transportservers
    singular: transportserver
    kind: TransportServer
    shortNames:
      - ts
  preserveUnknownFields: false
  versions:
    - name: v1alpha1
      served: true
      storage: true
  
```

```

schema:
openAPIV3Schema:
  description: TransportServer defines the TransportServer resource.
  type: object
  properties:
    apiVersion:
      description: 'APIVersion defines the versioned schema of this representation
      of an object. Servers should convert recognized schemas to the latest
      internal value, and may reject unrecognized values. More info:
https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#resources'
      type: string
    kind:
      description: 'Kind is a string value representing the REST resource this
      object represents. Servers may infer this from the endpoint the client
      submits requests to. Cannot be updated. In CamelCase. More info:
https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#types-kinds'
      type: string
    metadata:
      type: object
    spec:
      description: TransportServerSpec is the spec of the TransportServer resource.
      type: object
      properties:
        action:
          description: Action defines an action.
          type: object
          properties:
            pass:
              type: string
        host:
          type: string
        listener:
          description: TransportServerListener defines a listener for a TransportServer.
          type: object
          properties:
            name:
              type: string
            protocol:
              type: string
        upstreamParameters:
          description: UpstreamParameters defines parameters for an upstream.
          type: object
          properties:
            udpRequests:
              type: integer
            udpResponses:
              type: integer
        upstreams:
          type: array
          items:
            description: Upstream defines an upstream.
            type: object
            properties:
              name:
                type: string
              port:
                type: integer
              service:
                type: string

```



Fondos Europeos
Fondo Social Europeo



Cofinanciado por la Unión Europea



Ejecutamos el manifiesto:

```
$ kubectl apply -f nginx-ts-definition.yml
customresourcedefinition.apieextensions.k8s.io/transportservers.k8s.nginx.org created
```

45.1.11. Creando las Policy

El recurso de política nos va a permitir configurar funciones como:

- Control de acceso
- Limitación de velocidad

Las políticas se pueden agregar a los recursos personalizados de VirtualServer y VirtualServerRoute.

Creamos el archivo **nginx-policy-definition.yml** y le añadimos el siguiente contenido:

```
apiVersion: apieextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: policies.k8s.nginx.org
spec:
  group: k8s.nginx.org
  scope: Namespaced
  names:
    plural: policies
    singular: policy
    kind: Policy
    shortNames:
      - pol
  preserveUnknownFields: false
  versions:
    - name: v1alpha1
      served: true
      storage: true
      schema:
        openAPIV3Schema:
          description: Policy defines a Policy for VirtualServer and VirtualServerRoute
          resources:
            type: object
            properties:
              apiVersion:
                description: 'APIVersion defines the versioned schema of this representation
                  of an object. Servers should convert recognized schemas to the latest
                  internal value, and may reject unrecognized values. More info:
https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#resources'
                type: string
              kind:
                description: 'Kind is a string value representing the REST resource this
                  object represents. Servers may infer this from the endpoint the client
                  submits requests to. Cannot be updated. In CamelCase. More info:
https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#types-kinds'
                type: string
              metadata:
                type: object
              spec:
                description: PolicySpec is the spec of the Policy resource. The spec includes
                  multiple fields, where each field represents a different policy. Only
                  one policy (field) is allowed.
                type: object
```

```

properties:
  accessControl:
    description: AccessControl defines an access policy based on the source
    IP of a request.
    type: object
    properties:
      allow:
        type: array
        items:
          type: string
      deny:
        type: array
        items:
          type: string
  egressMTLS:
    description: EgressMTLS defines an Egress MTLS policy.
    type: object
    properties:
      ciphers:
        type: string
      protocols:
        type: string
      serverName:
        type: boolean
      sessionReuse:
        type: boolean
      sslName:
        type: string
      tlsSecret:
        type: string
      trustedCertSecret:
        type: string
      verifyDepth:
        type: integer
      verifyServer:
        type: boolean
  ingressMTLS:
    description: IngressMTLS defines an Ingress MTLS policy.
    type: object
    properties:
      clientCertSecret:
        type: string
      verifyClient:
        type: string
      verifyDepth:
        type: integer
  jwt:
    description: JWTAuth holds JWT authentication configuration.
    type: object
    properties:
      realm:
        type: string
      secret:
        type: string
      token:
        type: string
  rateLimit:
    description: RateLimit defines a rate limit policy.
    type: object
    properties:
      burst:
        type: integer
      delay:
        type: integer
      dryRun:
        type: boolean
      key:

```



Fondos Europeos

MINISTERIO
DE EDUCACIÓN, FORMACIÓN PROFESIONAL
Y DEPORTE

Cofinanciado por
la Unión Europea



```

type: string
logLevel:
  type: string
noDelay:
  type: boolean
rate:
  type: string
rejectCode:
  type: integer
zoneSize:
  type: string

```

Ejecutamos el manifiesto:

```
$ kubectl apply -f nginx-policy-definition.yml
customresourcedefinition.apiextensions.k8s.io/policies.k8s.nginx.org created
```

45.1.12. Creando el recurso personalizado APLogConf

Permite utilizar el módulo App Protect de nginx.

Creamos el archivo **nginx-ap-logconf-definition.yml** y le añadimos el siguiente contenido:

```

apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  annotations:
    controller-gen.kubebuilder.io/version: v0.2.5
  creationTimestamp: null
  name: aplogconfs.appprotect.f5.com
spec:
  group: appprotect.f5.com
  scope: Namespaced
  names:
    kind: APLogConf
    listKind: APLogConfList
    plural: aplogconfs
    singular: aplogconf
  preserveUnknownFields: false
  versions:
    - name: v1
      served: true
      storage: true
      schema:
        openAPIV3Schema:
          description: APLogConf is the Schema for the APLogConfs API
          properties:
            apiVersion:
              description: 'APIVersion defines the versioned schema of this representation
              of an object. Servers should convert recognized schemas to the latest
              internal value, and may reject unrecognized values. More info:
https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#resources'
              type: string
            kind:
              description: 'Kind is a string value representing the REST resource this
              object represents. Servers may infer this from the endpoint the client
              submits requests to. Cannot be updated. In CamelCase. More info:
https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#types-kinds'
```

```

type: string
metadata:
  type: object
spec:
  description: APLogConfSpec defines the desired state of APLogConf
properties:
  content:
    properties:
      format:
        enum:
          - splunk
          - arcsight
          - default
          - user-defined
      type: string
  format_string:
    type: string
  max_message_size:
    pattern: ^([1-9]| [1-5][0-9]| 6[0-4])k$  

    type: string
  max_request_size:
    pattern: ^([1-9]| [1-9][0-9]| [1-9][0-9]{2}| 1[0-9]{3}| 20[1-3][0-9]| 204[1-8]| any)$  

    type: string
  type: object
filter:
  properties:
    request_type:
      enum:
        - all
        - illegal
        - blocked
      type: string
  type: object
type: object
type: object

```

Ejecutamos el manifiesto:

```
$ kubectl apply -f nginx-ap-logconf-definition.yml  
  
customresourcedefinition.apiextensions.k8s.io/aplogconfs.appprotect.f5.com created
```

45.1.13. Creando el recurso personalizado APPolicy

Permite utilizar el módulo App Protect de nginx.

Creamos el archivo **nginx-ap-policy-definition.yml** y le añadimos el siguiente contenido:

```

apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  annotations:
    controller-gen.kubebuilder.io/version: v0.2.5
  creationTimestamp: null
  name: appolicies.appprotect.f5.com
spec:
  group: appprotect.f5.com
  scope: Namespaced
  names:

```

```

kind: APPolicy
listKind: APPolicyList
plural: appolicies
singular: appolicy
preserveUnknownFields: false
versions:
- name: v1
  served: true
  storage: true
schema:
  openAPIV3Schema:
    description: APPolicyConfig is the Schema for the APPolicyconfigs API
  properties:
    apiVersion:
      description: 'APIVersion defines the versioned schema of this representation
        of an object. Servers should convert recognized schemas to the latest
        internal value, and may reject unrecognized values. More info:
https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#resources'
      type: string
    kind:
      description: 'Kind is a string value representing the REST resource this
        object represents. Servers may infer this from the endpoint the client
        submits requests to. Cannot be updated. In CamelCase. More info:
https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#types-kinds'
      type: string
    metadata:
      type: object
    spec:
      description: APPolicySpec defines the desired state of APPolicy
      properties:
        modifications:
          items:
            properties:
              action:
                type: string
              description:
                type: string
              entity:
                properties:
                  name:
                    type: string
                  type: object
              entityChanges:
                properties:
                  type:
                    type: string
                  type: object
              type: object
            x-kubernetes-preserve-unknown-fields: true
          type: array
        modificationsReference:
          properties:
            link:
              pattern: ^http
              type: string
          type: object
        policy:
          description: Defines the App Protect policy
          properties:
            applicationLanguage:
              enum:
                - iso-8859-10
                - iso-8859-6
                - windows-1255
                - auto-detect
                - koi8-r
                - gb18030
  
```

```

- iso-8859-8
- windows-1250
- iso-8859-9
- windows-1252
- iso-8859-16
- gb2312
- iso-8859-2
- iso-8859-5
- windows-1257
- windows-1256
- iso-8859-13
- windows-874
- windows-1253
- iso-8859-3
- euc-jp
- utf-8
- gbk
- windows-1251
- big5
- iso-8859-1
- shift_jis
- euc-kr
- iso-8859-4
- iso-8859-7
- iso-8859-15
type: string
blocking-settings:
properties:
evasions:
items:
properties:
description:
enum:
- "%u decoding"
- Apache whitespace
- Bad unescape
- Bare byte decoding
- Directory traversals
- IIS backslashes
- IIS Unicode codepoints
- Multiple decoding
type: string
enabled:
type: boolean
maxDecodingPasses:
type: integer
type: object
type: array
http-protocols:
items:
properties:
description:
enum:
- Unescaped space in URL
- Unparseable request content
- Several Content-Length headers
- 'POST request with Content-Length: 0'
- Null in request
- No Host header in HTTP/1.1 request
- Multiple host headers
- Host header contains IP address
- High ASCII characters in headers
- Header name with no header value
- CRLF characters before request start
- Content length should be a positive number
- Chunked request with Content-Length header
- Check maximum number of parameters

```

- Check maximum number of headers
- Body in GET or HEAD requests
- Bad multipart/form-data request parsing
- Bad multipart parameters parsing
- Bad HTTP version
- Bad host header value

type: string

enabled:

- type: boolean

maxHeaders:

- type: integer

maxParams:

- type: integer

type: object

type: array

violations:

items:

properties:

- alarm:
- type: boolean
- block:
- type: boolean
- description:
- type: string
- name:
- enum:
 - VIOL_ASM_COOKIE_MODIFIED
 - VIOL_BLACKLISTED_IP
 - VIOL_COOKIE_EXPIRED
 - VIOL_COOKIE_LENGTH
 - VIOL_COOKIE_MALFORMED
 - VIOL_COOKIE_MODIFIED
 - VIOL_DATA_GUARD
 - VIOL_ENCODING
 - VIOL_EVASION
 - VIOL_FILETYPE
 - VIOL_FILE_UPLOAD
 - VIOL_FILE_UPLOAD_IN_BODY
 - VIOL_HEADER_LENGTH
 - VIOL_HEADER_METACHAR
 - VIOL_HTTP_PROTOCOL
 - VIOL_HTTP_RESPONSE_STATUS
 - VIOL_JSON_FORMAT
 - VIOL_JSON_MALFORMED
 - VIOL_JSON_SCHEMA
 - VIOL_MANDATORY_PARAMETER
 - VIOL_MANDATORY_REQUEST_BODY
 - VIOL_METHOD
 - VIOL_PARAMETER
 - VIOL_PARAMETER_DATA_TYPE
 - VIOL_PARAMETER_EMPTY_VALUE
 - VIOL_PARAMETER_LOCATION
 - VIOL_PARAMETER_MULTIPART_NULL_VALUE
 - VIOL_PARAMETER_NAME_METACHAR
 - VIOL_PARAMETER_NUMERIC_VALUE
 - VIOL_PARAMETER_REPEAT
 - VIOL_PARAMETER_STATIC_VALUE
 - VIOL_PARAMETER_VALUE_LENGTH
 - VIOL_PARAMETER_VALUE_METACHAR
 - VIOL_POST_DATA_LENGTH
 - VIOL_QUERY_STRING_LENGTH
 - VIOL_RATING_THREAT
 - VIOL_RATING_NEED_EXAMINATION
 - VIOL_REQUEST_MAX_LENGTH
 - VIOL_REQUEST_LENGTH
 - VIOL_THREAT_CAMPAIGN
 - VIOL_URL



```

        - VIOL_URL_CONTENT_TYPE
        - VIOL_URL_LENGTH
        - VIOL_URL_METACHAR
        - VIOL_XML_FORMAT
        - VIOL_XML_MALFORMED
        type: string
    type: object
    type: array
    type: object
blockingSettingReference:
properties:
link:
pattern: ^http
type: string
type: object
caseInsensitive:
type: boolean
character-sets:
items:
properties:
characterSet:
items:
properties:
isAllowed:
type: boolean
metachar:
type: string
type: object
type: array
characterSetType:
enum:
- gwt-content
- header
- json-content
- parameter-name
- parameter-value
- plain-text-content
- url
- xml-content
type: string
type: object
type: array
characterSetReference:
properties:
link:
pattern: ^http
type: string
type: object
cookie-settings:
properties:
maximumCookieHeaderLength:
pattern: any|\d+
type: string
type: object
cookieReference:
properties:
link:
pattern: ^http
type: string
type: object
cookieSettingsReference:
properties:
link:
pattern: ^http
type: string
type: object
cookies:

```



Fondos Europeos



MINISTERIO
DE EDUCACIÓN, FORMACIÓN PROFESIONAL
Y DEPORTE

Gobierno de España
Cofinanciado por
la Unión Europea



```

items:
properties:
accessibleOnlyThroughTheHttpProtocol:
  type: boolean
attackSignaturesCheck:
  type: boolean
enforcementType:
  type: string
insertSameSiteAttribute:
  enum:
    - lax
    - none
    - none-value
    - strict
  type: string
name:
  type: string
securedOverHttpsConnection:
  type: boolean
signatureOverrides:
  items:
    properties:
      enabled:
        type: boolean
      signatureId:
        type: integer
      type: object
    type: array
  type:
    enum:
      - explicit
      - wildcard
    type: string
  type: object
  type: array
data-guard:
  properties:
    creditCardNumbers:
      type: boolean
    enabled:
      type: boolean
    enforcementMode:
      enum:
        - ignore-urls-in-list
        - enforce-urls-in-list
      type: string
    enforcementUrls:
      items:
        type: string
      type: array
    lastCcnDigitsToExpose:
      type: integer
    lastSsnDigitsToExpose:
      type: integer
    maskData:
      type: boolean
    usSocialSecurityNumbers:
      type: boolean
  type: object
  dataGuardReference:
    properties:
      link:
        pattern: ^http
        type: string
    type: object
  description:
    type: string

```



Fondos Europeos



MINISTERIO
DE EDUCACIÓN, FORMACIÓN PROFESIONAL
Y DEPORTE

Cofinanciado por
la Unión Europea



```

enablePassiveMode:
  type: boolean
enforcementMode:
  enum:
    - transparent
    - blocking
  type: string
filetypeReference:
  properties:
    link:
      pattern: ^http
      type: string
  type: object
filetypes:
  items:
    properties:
      allowed:
        type: boolean
    checkPostDataLength:
      type: boolean
    checkQueryStringLength:
      type: boolean
    checkRequestLength:
      type: boolean
    checkUrlLength:
      type: boolean
  name:
    type: string
  postDataLength:
    type: integer
  queryStringLength:
    type: integer
  requestLength:
    type: integer
  responseCheck:
    type: boolean
  type:
    enum:
      - explicit
      - wildcard
    type: string
  urlLength:
    type: integer
  type: object
  type: array
fullPath:
  type: string
general:
  properties:
    allowedResponseCodes:
      items:
        format: int32
        maximum: 999
        minimum: 100
        type: integer
      type: array
    customXffHeaders:
      items:
        type: string
      type: array
    maskCreditCardNumbersInRequest:
      type: boolean
    trustXff:
      type: boolean
    type: object
  generalReference:
    properties:

```



Fondos Europeos



Cofinanciado por
la Unión Europea



```

link:
  pattern: ^http
  type: string
type: object
header-settings:
  properties:
    maximumHttpHeaderLength:
      pattern: any|\d+
      type: string
type: object
headerReference:
  properties:
    link:
      pattern: ^http
      type: string
type: object
headerSettingsReference:
  properties:
    link:
      pattern: ^http
      type: string
type: object
headers:
  items:
    properties:
      base64Decoding:
        type: boolean
      checkSignatures:
        type: boolean
      htmlNormalization:
        type: boolean
      mandatory:
        type: boolean
      maskValueInLogs:
        type: boolean
      name:
        type: string
      normalizationViolations:
        type: boolean
      percentDecoding:
        type: boolean
      type:
        enum:
          - explicit
          - wildcard
        type: string
      urlNormalization:
        type: boolean
type: object
type: array
json-profiles:
  items:
    properties:
      defenseAttributes:
        properties:
          maximumArrayLength:
            pattern: any|\d+
            type: string
          maximumStructureDepth:
            pattern: any|\d+
            type: string
          maximumTotalLengthOfJSONData:
            pattern: any|\d+
            type: string
          maximumValueLength:
            pattern: any|\d+
            type: string

```



Fondos Europeos



MINISTERIO
DE EDUCACIÓN, FORMACIÓN PROFESIONAL
Y DEPORTE

Gobierno de España
Cofinanciado por
la Unión Europea



```

tolerateJSONParsingWarnings:
  type: boolean
type: object
description:
  type: string
hasValidationFiles:
  type: boolean
metacharOverrides:
  items:
    properties:
      isAllowed:
        type: boolean
      metachar:
        type: string
    type: object
  type: array
name:
  enum:
    - Default
    type: string
signatureOverrides:
  items:
    properties:
      enabled:
        type: boolean
      signatureId:
        type: integer
    type: object
  type: array
validationFiles:
  items:
    properties:
      importUrl:
        type: string
      isPrimary:
        type: boolean
    jsonValidationFile:
      properties:
        contents:
          type: string
        fileName:
          type: string
        isBase64:
          type: boolean
        type: object
      type: object
    type: array
  type: object
type: array
json-validation-files:
  items:
    properties:
      contents:
        type: string
      fileName:
        type: string
      isBase64:
        type: boolean
      type: object
    type: object
  type: array
jsonProfileReference:
  properties:
    link:
      pattern: ^http
      type: string
    type: object
  type: object
  jsonValidationFileReference:

```



Fondos Europeos



MINISTERIO
DE EDUCACIÓN, FORMACIÓN PROFESIONAL
Y DEPORTE

Cofinanciado por
la Unión Europea



```

properties:
link:
  pattern: ^http
  type: string
type: object
methodReference:
properties:
link:
  pattern: ^http
  type: string
type: object
methods:
items:
properties:
name:
  type: string
type: object
type: array
name:
  type: string
open-api-files:
items:
properties:
link:
  pattern: ^http
  type: string
type: object
type: array
parameterReference:
properties:
link:
  pattern: ^http
  type: string
type: object
parameters:
items:
properties:
allowEmptyValue:
  type: boolean
allowRepeatedParameterName:
  type: boolean
arraySerializationFormat:
enum:
- csv
- form
- label
- matrix
- multi
- multipart
- pipe
- ssv
- tsv
type: string
attackSignaturesCheck:
  type: boolean
checkMaxValue:
  type: boolean
checkMaxValueLength:
  type: boolean
checkMetachars:
  type: boolean
checkMinValue:
  type: boolean
checkMinValueLength:
  type: boolean
checkMultipleOfValue:
  type: boolean

```



Fondos Europeos

MINISTERIO
DE EDUCACIÓN, FORMACIÓN PROFESIONAL
Y DEPORTE

Cofinanciado por
la Unión Europea



```
contentProfile:  
  properties:  
    name:  
      type: string  
    type: object  
dataType:  
  enum:  
    - alpha-numeric  
    - binary  
    - boolean  
    - decimal  
    - email  
    - integer  
    - none  
    - phone  
  type: string  
disallowFileUploadOfExecutables:  
  type: boolean  
enableRegularExpression:  
  type: boolean  
exclusiveMax:  
  type: boolean  
exclusiveMin:  
  type: boolean  
isCookie:  
  type: boolean  
isHeader:  
  type: boolean  
level:  
  enum:  
    - global  
    - url  
  type: string  
maxLength:  
  type: integer  
metacharsOnParameterValueCheck:  
  type: boolean  
minimumLength:  
  type: integer  
name:  
  type: string  
nameMetacharOverrides:  
  items:  
    properties:  
      isAllowed:  
        type: boolean  
      metachar:  
        type: string  
    type: object  
  type: array  
objectSerializationStyle:  
  type: string  
parameterEnumValues:  
  items:  
    type: string  
  type: array  
parameterLocation:  
  enum:  
    - any  
    - cookie  
    - form-data  
    - header  
    - path  
    - query  
  type: string  
regularExpression:  
  type: string
```



Fondos Europeos



Cofinanciado por
la Unión Europea



```

sensitiveParameter:
  type: boolean
signatureOverrides:
  items:
    properties:
      enabled:
        type: boolean
      signatureId:
        type: integer
      type: object
    type: array
staticValues:
  type: string
type:
  enum:
    - explicit
    - wildcard
  type: string
valueMetacharOverrides:
  items:
    properties:
      isAllowed:
        type: boolean
      metachar:
        type: string
    type: object
  type: array
valueType:
  enum:
    - array
    - auto-detect
    - dynamic-content
    - dynamic-parameter-name
    - ignore
    - json
    - object
    - openapi-array
    - static-content
    - user-input
    - xml
  type: string
type: object
type: array
response-pages:
  items:
    properties:
      ajaxActionType:
        enum:
          - alert-popup
          - custom
          - redirect
        type: string
      ajaxCustomContent:
        type: string
      ajaxEnabled:
        type: boolean
      ajaxPopupMessage:
        type: string
      ajaxRedirectUrl:
        type: string
      responseActionType:
        enum:
          - custom
          - default
          - erase-cookies
          - redirect
          - soap-fault

```



Fondos Europeos



MINISTERIO
DE EDUCACIÓN, FORMACIÓN PROFESIONAL
Y DEPORTE

Cofinanciado por
la Unión Europea



```
    type: string
  responseContent:
    type: string
  responseHeader:
    type: string
  responsePageType:
    enum:
      - ajax
      - ajax-login
      - captcha
      - captcha-fail
      - default
      - failed-login-honeypot
      - failed-login-honeypot-ajax
      - hijack
      - leaked-credentials
      - leaked-credentials-ajax
      - mobile
      - persistent-flow
      - xml
    type: string
  responseRedirectUrl:
    type: string
  type: object
  type: array
  responsePageReference:
    properties:
      link:
        pattern: ^http
        type: string
    type: object
  sensitive-parameters:
    items:
      properties:
        name:
          type: string
      type: object
    type: array
  sensitiveParameterReference:
    properties:
      link:
        pattern: ^http
        type: string
    type: object
  server-technologies:
    items:
      properties:
        serverTechnologyName:
          enum:
            - Jenkins
            - SharePoint
            - Oracle Application Server
            - Python
            - Oracle Identity Manager
            - Spring Boot
            - CouchDB
            - SQLite
            - Handlebars
            - Mustache
            - Prototype
            - Zend
            - Redis
            - Underscore.js
            - Ember.js
            - ZURB Foundation
            - ef.js
            - Vue.js
```



Fondos Europeos



MINISTERIO
DE EDUCACIÓN, FORMACIÓN PROFESIONAL
Y DEPORTE

Gobierno de España
Cofinanciado por
la Unión Europea



```

- UIKit
- TYPO3 CMS
- RequireJS
- React
- MooTools
- Laravel
- GraphQL
- Google Web Toolkit
- Express.js
- CodeIgniter
- Backbone.js
- AngularJS
- JavaScript
- Nginx
- Jetty
- Joomla
- JavaServer Faces (JSF)
- Ruby
- MongoDB
- Django
- Node.js
- Citrix
- JBoss
- Elasticsearch
- Apache Struts
- XML
- PostgreSQL
- IBM DB2
- Sybase/ASE
- CGI
- Proxy Servers
- SSI (Server Side Includes)
- Cisco
- Novell
- Macromedia JRun
- BEA Systems WebLogic Server
- Lotus Domino
- MySQL
- Oracle
- Microsoft SQL Server
- PHP
- Outlook Web Access
- Apache/NCSA HTTP Server
- Apache Tomcat
- WordPress
- Macromedia ColdFusion
- Unix/Linux
- Microsoft Windows
- ASP.NET
- Front Page Server Extensions (FPSE)
- IIS
- WebDAV
- ASP
- Java Servlets/JSP
- jQuery
type: string
type: object
type: array
serverTechnologyReference:
properties:
link:
pattern: ^http
type: string
type: object
signature-requirements:
properties:
maxRevisionDatetime:

```

```

format: date-time
type: string
minRevisionDatetime:
  format: date-time
  type: string
tag:
  type: string
type: object
signature-sets:
  items:
    properties:
      alarm:
        type: boolean
      block:
        type: boolean
      name:
        enum:
          - Command Execution Signatures
          - Cross Site Scripting Signatures
          - Directory Indexing Signatures
          - Information Leakage Signatures
          - OS Command Injection Signatures
          - Path Traversal Signatures
          - Predictable Resource Location Signatures
          - Remote File Include Signatures
          - SQL Injection Signatures
          - XPath Injection Signatures
          - Buffer Overflow Signatures
          - Denial of Service Signatures
          - Vulnerability Scanner Signatures
      type: string
    type: object
    x-kubernetes-preserve-unknown-fields: true
  type: array
signature-settings:
  properties:
    attackSignatureFalsePositiveMode:
      enum:
        - detect
        - detect-and-allow
        - disabled
      type: string
    minimumAccuracyForAutoAddedSignatures:
      enum:
        - high
        - low
        - medium
      type: string
  type: object
signatureReference:
  properties:
    link:
      pattern: ^http
      type: string
  type: object
signatureSetReference:
  properties:
    link:
      pattern: ^http
      type: string
  type: object
signatureSettingReference:
  properties:
    link:
      pattern: ^http
      type: string
  type: object

```



```

signatures:
  items:
    properties:
      enabled:
        type: boolean
      signatureId:
        type: integer
    type: object
  type: array
softwareVersion:
  type: string
template:
  properties:
    name:
      type: string
  type: object
threat-campaigns:
  items:
    properties:
      isEnabled:
        type: boolean
      name:
        type: string
    type: object
  type: array
threatCampaignReference:
  properties:
    link:
      pattern: ^http
      type: string
  type: object
urlReference:
  properties:
    link:
      pattern: ^http
      type: string
  type: object
urls:
  items:
    properties:
      attackSignaturesCheck:
        type: boolean
      description:
        type: string
      disallowFileUploadOfExecutables:
        type: boolean
      isAllowed:
        type: boolean
      mandatoryBody:
        type: boolean
      metacharOverrides:
        items:
          properties:
            isAllowed:
              type: boolean
            metachar:
              type: string
          type: object
        type: array
      metacharsOnUrlCheck:
        type: boolean
      method:
        enum:
          - ACL
          - BCOPY
          - BDELETE
          - BMOVE

```



```

- BPROPFIND
- BPROPPATCH
- CHECKIN
- CHECKOUT
- CONNECT
- COPY
- DELETE
- GET
- HEAD
- LINK
- LOCK
- MERGE
- MKCOL
- MKWORKSPACE
- MOVE
- NOTIFY
- OPTIONS
- PATCH
- POLL
- POST
- PROPFIND
- PROPPATCH
- PUT
- REPORT
- RPC_IN_DATA
- RPC_OUT_DATA
- SEARCH
- SUBSCRIBE
- TRACE
- TRACK
- UNLINK
- UNLOCK
- UNSUBSCRIBE
- VERSION_CONTROL
- X-MS-ENUMATTS
- '*'

type: string
methodOverrides:
  items:
    properties:
      allowed:
        type: boolean
      method:
        enum:
          - ACL
          - BCOPY
          - BDELETE
          - BMOVE
          - BPROPFIND
          - BPROPPATCH
          - CHECKIN
          - CHECKOUT
          - CONNECT
          - COPY
          - DELETE
          - GET
          - HEAD
          - LINK
          - LOCK
          - MERGE
          - MKCOL
          - MKWORKSPACE
          - MOVE
          - NOTIFY
          - OPTIONS
          - PATCH
          - POLL

```



Fondos Europeos



Cofinanciado por la Unión Europea



```

    - POST
    - PROPFIND
    - PROPPATCH
    - PUT
    - REPORT
    - RPC_IN_DATA
    - RPC_OUT_DATA
    - SEARCH
    - SUBSCRIBE
    - TRACE
    - TRACK
    - UNLINK
    - UNLOCK
    - UNSUBSCRIBE
    - VERSION_CONTROL
    - X-MS-ENUMATTS
    type: string
  type: object
  type: array
methodsOverrideOnUrlCheck:
  type: boolean
name:
  type: string
positionalParameters:
  items:
    properties:
      parameter:
        properties:
          allowEmptyValue:
            type: boolean
          allowRepeatedParameterName:
            type: boolean
        arraySerializationFormat:
          enum:
            - csv
            - form
            - label
            - matrix
            - multi
            - multipart
            - pipe
            - ssv
            - tsv
        type: string
attackSignaturesCheck:
  type: boolean
checkMaxValue:
  type: boolean
checkMaxValueLength:
  type: boolean
checkMetachars:
  type: boolean
checkMinValue:
  type: boolean
checkMinValueLength:
  type: boolean
checkMultipleOfValue:
  type: boolean
contentProfile:
  properties:
    name:
      type: string
  type: object
dataType:
  enum:
    - alpha-numeric
    - binary

```

```

- boolean
- decimal
- email
- integer
- none
- phone
type: string
DisallowFileUploadOfExecutables:
type: boolean
enableRegularExpression:
type: boolean
exclusiveMax:
type: boolean
exclusiveMin:
type: boolean
isCookie:
type: boolean
isHeader:
type: boolean
level:
enum:
- global
- url
type: string
maxLength:
type: integer
metacharsOnParameterValueCheck:
type: boolean
minimumLength:
type: integer
name:
type: string
nameMetacharOverrides:
items:
properties:
isAllowed:
type: boolean
metachar:
type: string
type: object
type: array
objectSerializationStyle:
type: string
parameterEnumValues:
items:
type: string
type: array
parameterLocation:
enum:
- any
- cookie
- form-data
- header
- path
- query
type: string
regularExpression:
type: string
sensitiveParameter:
type: boolean
signatureOverrides:
items:
properties:
enabled:
type: boolean
signatureId:
type: integer

```



Fondos Europeos

MINISTERIO
DE EDUCACIÓN, FORMACIÓN PROFESIONAL
Y DEPORTE



Cofinanciado por
la Unión Europea



```
        type: object
        type: array
        staticValues:
          type: string
        type:
          enum:
            - explicit
            - wildcard
          type: string
    valueMetacharOverrides:
      items:
        properties:
          isAllowed:
            type: boolean
          metachar:
            type: string
        type: object
      type: array
    valueType:
      enum:
        - array
        - auto-detect
        - dynamic-content
        - dynamic-parameter-name
        - ignore
        - json
        - object
        - openapi-array
        - static-content
        - user-input
        - xml
      type: string
    type: object
  urlSegmentIndex:
    type: integer
  type: object
  type: array
  protocol:
    enum:
      - http
      - https
    type: string
  signatureOverrides:
    items:
      properties:
        enabled:
          type: boolean
        signatureId:
          type: integer
      type: object
    type: array
  type:
    enum:
      - explicit
      - wildcard
    type: string
  urlContentProfiles:
    items:
      properties:
        headerName:
          type: string
        headerOrder:
          type: string
        headerValue:
          type: string
        name:
          type: string
```



Fondos Europeos



Cofinanciado por
la Unión Europea



```

type:
enum:
- apply-content-signatures
- apply-value-and-content-signatures
- disallow
- do-nothing
- form-data
- gwt
- json
- xml
type: string
type: object
type: array
wildcardOrder:
type: integer
type: object
type: array
whitelist-ips:
items:
properties:
blockRequests:
enum:
- always
- never
- policy-default
type: string
ipAddress:
pattern: '[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}'
type: string
ipMask:
pattern: '[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}'
type: string
type: object
type: array
whitelistIpReference:
properties:
link:
pattern: ^http
type: string
type: object
xml-profiles:
items:
properties:
attackSignaturesCheck:
type: boolean
defenseAttributes:
properties:
allowCDATA:
type: boolean
allowDTDs:
type: boolean
allowExternalReferences:
type: boolean
allowProcessingInstructions:
type: boolean
maximumAttributeValueLength:
pattern: any|\d+
type: string
maximumAttributesPerElement:
pattern: any|\d+
type: string
maximumChildrenPerElement:
pattern: any|\d+
type: string
maximumDocumentDepth:
pattern: any|\d+
type: string

```



Fondos Europeos

MINISTERIO
DE EDUCACIÓN, FORMACIÓN PROFESIONAL
Y DEPORTE

Cofinanciado por
la Unión Europea



```
maximumDocumentSize:  
  pattern: any|\d+  
  type: string  
maximumElements:  
  pattern: any|\d+  
  type: string  
maximumNSDeclarations:  
  pattern: any|\d+  
  type: string  
maximumNameLength:  
  pattern: any|\d+  
  type: string  
maximumNamespaceLength:  
  pattern: any|\d+  
  type: string  
tolerateCloseTagShorthand:  
  type: boolean  
tolerateLeadingWhiteSpace:  
  type: boolean  
tolerateNumericNames:  
  type: boolean  
type: object  
description:  
  type: string  
enableWss:  
  type: boolean  
followSchemaLinks:  
  type: boolean  
name:  
  type: string  
type: object  
type: array  
xml-validation-files:  
  items:  
    properties:  
      contents:  
        type: string  
      fileName:  
        type: string  
      isBase64:  
        type: boolean  
    type: object  
  type: array  
xmlProfileReference:  
  properties:  
    link:  
      pattern: ^http  
      type: string  
  type: object  
xmlValidationFileReference:  
  properties:  
    link:  
      pattern: ^http  
      type: string  
  type: object  
  type: object  
  type: object  
  type: object
```

Ejecutamos el manifiesto:



Fondos Europeos



MINISTERIO DE EDUCACIÓN, FORMACIÓN PROFESIONAL Y DEPORTE

Gobierno de España
Cofinanciado por la Unión Europea



```
$ kubectl apply -f nginx-ap-policy-definition.yml  
customresourcedefinition.apieextensions.k8s.io/appolicies.appprotect.f5.com created
```

45.1.14. Desplegando el Ingress Controller nginx

¡Parecía que no iba a llegar nunca!, pero si, ha llegado la hora de desplegar al fin el controlador -

Para desplegar el controlador de ingress, podemos utilizar 2 tipos de objetos de kubernetes:

- **Deployment**

- Utilizaremos este objeto si planeamos cambiar dinámicamente la cantidad de réplicas del controlador Ingress en el clúster.

- **DemonSet**

- Utilizaremos este objeto para implementar el controlador Ingress en cada nodo o un subconjunto de nodos (1 por nodo, sin replicación)

Para nuestro caso, nos vamos a decantar por la opción del **Deployment**.

Creamos el archivo **nginx-ingress.yml** y le añadimos el siguiente contenido:

```
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  name: nginx-ingress  
  namespace: nginx-ingress  
spec:  
  replicas: 1  
  selector:  
    matchLabels:  
      app: nginx-ingress  
  template:  
    metadata:  
      labels:  
        app: nginx-ingress  
    spec:  
      serviceAccountName: nginx-ingress  
      containers:  
        - image: nginx/nginx-ingress:1.9.1  
          imagePullPolicy: IfNotPresent  
          name: nginx-ingress  
          ports:  
            - name: http  
              containerPort: 80  
            - name: https  
              containerPort: 443  
            - name: readiness-port  
              containerPort: 8081
```

```
readinessProbe:  
  httpGet:  
    path: /nginx-ready  
    port: readiness-port  
  periodSeconds: 1  
securityContext:  
  allowPrivilegeEscalation: true  
  runAsUser: 101 #nginx  
capabilities:  
  drop:  
    - ALL  
  add:  
    - NET_BIND_SERVICE  
env:  
  - name: POD_NAMESPACE  
    valueFrom:  
      fieldRef:  
        fieldPath: metadata.namespace  
  - name: POD_NAME  
    valueFrom:  
      fieldRef:  
        fieldPath: metadata.name  
args:  
  - -nginx-configmaps=$(POD_NAMESPACE)/nginx-config  
  - -default-server-tls-secret=$(POD_NAMESPACE)/default-server-secret  
  #- -v=3 # Enables extensive logging. Useful for troubleshooting.  
  #- -report-ingress-status  
  #- -external-service=nginx-ingress  
  #- -enable-prometheus-metrics  
  #- -global-configuration=$(POD_NAMESPACE)/nginx-configuration
```

Ejecutamos el manifiesto:

```
$ kubectl apply -f nginx-ingress.yml  
deployment.apps/nginx-ingress created
```

45.1.15. Verificando que el controlador ingress nginx está online

Vamos a ejecutar el siguiente comando para verificar que el controlador de ingress está operando correctamente en el namespace nginx-ingress.

Si observamos a la réplica en estado Running es que está operativo -

Ejecutamos el siguiente comando:

```
$ kubectl get pods --namespace=nginx-ingress
NAME                  READY STATUS RESTARTS AGE
nginx-ingress-79bcb95659-nf5wt 1/1   Running 0      106s
```

45.1.16. Creando el servicio LoadBalancer de acceso

Ahora, vamos a crear un servicio a modo de "sumidero".

Este servicio, con el puerto que le asignemos, será el que se encargará de recibir la petición de URL y de enrutar el servicio donde corresponda.

La idea, es que cuando utilicemos reglas de ingress, no utilicemos los servicios de las aplicaciones, sino, que todo apuntará al servicio de entrada de ingress y este sabrá enrutar.



Creamos el archivo **nginx-loadbalancer.yml** y le añadimos el siguiente contenido:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-ingress
  namespace: nginx-ingress
spec:
  type: LoadBalancer
  ports:
    - name: http
      port: 80
      targetPort: 80
      nodePort: 30010
    - name: https
      port: 443
      targetPort: 443
      nodePort: 30020
  selector:
    app: nginx-ingress
  externalIPs:
    - 192.168.15.100
```



Ejecutamos el siguiente comando:

```
$ kubectl apply -f nginx-nodeport.yml
service/nginx-ingress created
```



45.1.17. Verificando que todos los componentes del controlador ingress están operativos

Para llevar a cabo una visión general de que el controlador ingress junto con su servicio están operativos, vamos a ejecutar el siguiente comando:

Observamos:

- El deployment del controlador
- El replicaset asociado
- El servicio de recepción de rutas ingress
- La réplica (el pod) del controlador nginx operando

```
$ kubectl get all -n nginx-ingress

NAME           READY   STATUS    RESTARTS   AGE
pod/nginx-ingress-79bcb95659-nf5wt  1/1     Running   0          36m

NAME      TYPE      CLUSTER-IP    EXTERNAL-IP    PORT(S)        AGE
service/nginx-ingress  LoadBalancer  10.99.94.148  192.168.15.100  80:30010/TCP,443:30020/TCP  21m

NAME           READY   UP-TO-DATE  AVAILABLE   AGE
deployment.apps/nginx-ingress  1/1     1           1          36m

NAME           DESIRED  CURRENT  READY   AGE
replicaset.apps/nginx-ingress-79bcb95659  1       1       1       36m
```

45.2. Lab: Ingress App utilizando Ingress Controller (Nginx)

Mediante este laboratorio, vamos a desplegar una aplicación que utilizará reglas de ingress para acceder a la misma.

Necesitaremos previamente antes de realizar este laboratorio, un controlador ingress nginx en funcionamiento.

La idea, es que pondremos en marcha una aplicación con dos versiones mediante el uso de dos deployments, cada deployment se accederá con su correspondiente servicio.

Finalmente crearemos las reglas ingress, de forma que indicaremos una URL para acceder a la aplicación V1 y otra URL para acceder a la aplicación V2.

45.2.1. Creando el namespace

Lo primero que vamos a hacer, es crear un namespace para albergar los componentes de nuestra aplicación.

Creamos el archivo **ingress-app-namespace.yml** y le añadimos el siguiente contenido:

```
apiVersion: v1
kind: Namespace
metadata:
  name: ingress-app
```

Ejecutamos el manifiesto:

```
$ kubectl apply -f ingress-app-namespace.yml
namespace/ingress-app created
```

45.2.2. Creando el deployment (Aplicación Versión 1.0)

Creamos el archivo **ingress-app-deployment-v1.yml** y le asignamos el siguiente contenido:

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-v1
  namespace: ingress-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: hello-v1
  template:
    metadata:
      labels:
        app: hello-v1
    spec:
      containers:
        - name: hello
          image: gcr.io/google-samples/hello-app:1.0
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 8080
          resources:
            requests:
              memory: "64Mi"
              cpu: "200m"
            limits:
              memory: "128Mi"
              cpu: "500m"

```

Ejecutamos el manifiesto:

```

$ kubectl apply -f ingress-app-deployment-v1.yml
deployment.apps/hello-v1 created

```

45.2.3. Creando el deployment (Aplicación Versión 2.0)

Creamos el archivo **ingress-app-deployment-v2.yml** y le asignamos el siguiente contenido:



```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-v2
  namespace: ingress-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: hello-v2
  template:
    metadata:
      labels:
        app: hello-v2
    spec:
      containers:
        - name: hello
          image: gcr.io/google-samples/hello-app:2.0
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 8080
          resources:
            requests:
              memory: "64Mi"
              cpu: "200m"
            limits:
              memory: "128Mi"
              cpu: "500m"

```

Ejecutamos el manifiesto:

```

$ kubectl apply -f ingress-app-deployment-v2.yml
deployment.apps/hello-v2 created

```

45.2.4. Creando los servicios

Vamos a crear 2 servicios de tipo ClusterIP para acceder a los pods que cumplen cada uno de los selectores.

Los servicios abren la puerta del puerto 80 a nivel de IP interna del clúster y apuntan

Creamos el archivo **ingress-app-services.yml** y le añadimos el siguiente contenido:

```
kind: Service
apiVersion: v1
metadata:
  name: hello-v1-svc
  namespace: ingress-app
spec:
  selector:
    app: hello-v1
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

```
---
```

```
kind: Service
apiVersion: v1
metadata:
  name: hello-v2-svc
  namespace: ingress-app
spec:
  selector:
    app: hello-v2
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

Ejecutamos el manifiesto:

```
$ kubectl apply -f ingress-app-services.yml

service/hello-v1-svc created
service/hello-v2-svc created
```

45.2.5. Creando las reglas de ingress

Vamos a crear el archivo **ingress-app-rules.yml** y le añadimos el siguiente contenido:

```

apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress
  namespace: ingress-app
spec:
  ingressClassName: nginx
  rules:
    - host: app.v1.local
      http:
        paths:
          - backend:
              service:
                name: hello-v1-svc
                port:
                  number: 80
            path: /
            pathType: Exact
    - host: app.v2.local
      http:
        paths:
          - backend:
              service:
                name: hello-v2-svc
                port:
                  number: 80
            path: /
            pathType: Exact
  defaultBackend:
    service:
      name: hello-v1-svc
      port:
        number: 80

```

Ejecutamos el manifiesto:

```

$ kubectl apply -f ingress-app-rules.yml
ingress.networking.k8s.io/ingress created

```

Verificamos que las reglas de ingress está funcionando en nuestro namespace:

```

$ kubectl get ing -n ingress-app
NAME      CLASS   HOSTS           ADDRESS   PORTS   AGE
ingress   nginx   app.v1.local,app.v2.local   80      97s

```



45.2.6. Añadiendo entradas DNS

Ahora, vamos a modificar únicamente la máquina máster, de manera que en ella, añadiremos las siguientes entradas DNS en el archivo **/etc/hosts**:

```
192.168.15.100 app.v1.local  
192.168.15.100 app.v2.local
```

 La ip 192.168.15.100 es la IP de la máquina maestra, pero podríamos poner realmente la IP de cualquiera de los servidores, ya que los servicio ClusterIP están operando a nivel de la totalidad del clúster, es visible e interceptable en todas las máquinas.

Si queremos que esta misma resolución DNS esté presente en todos los servidores, en el master y en los minions

45.2.7. Probando la ejecución de las reglas de ingress

Vamos a ejecutar una petición http con la DNS de la aplicación en versión 1, el puerto al que tenemos que apuntar es el puerto de tipo NodePort del servicio del controlador de ingress (nginx) que está operando en el namespace nginx-ingress.

```
$ curl app.v1.local
```

```
Hello, world!  
Version: 1.0.0  
Hostname: hello-v1-5bd6f947bb-n8l45
```

- Observamos que nos da respuesta una de las réplicas de la aplicación en versión 1

Ahora, ejecutamos una petición http con la DNS de la aplicación en versión 2, el puerto al que tenemos que apuntar es el puerto de tipo NodePort del servicio del controlador de ingress (nginx) que está operando en el namespace nginx-ingress.

```
$ curl app.v2.local
```

```
Hello, world!  
Version: 2.0.0  
Hostname: hello-v2-6db4bdfb96-tmg6v
```

- Observamos que nos da respuesta una de las réplicas de la aplicación en versión 2

Capítulo 46. Nexus Integration

Un componente que resulta vital en nuestra propia infraestructura es un repositorio de imágenes binarias.

Sonatype Nexus resulta una opción muy interesante cuando queremos disponer dentro de la empresa de forma privada de un repositorio de imágenes binarias.



46.1. Lab: Nexus Integration

Mediante este laboratorio pondremos en marcha un repositorio binario nexus en nuestra infraestructura local y realizaremos la integración con nuestro clúster de kubernetes.



Para llevar a cabo este laboratorio, se da por realizado el laboratorio de creación de controlador ingress Nginx

46.1.1. Creando la unidad de persistencia (PV)

Primero, vamos a crear la unidad de persistencia para almacenar el contenido del sistema nexus.

Creamos en el nodo **kubeminion1** la carpeta: **/home/kubernetes/nexus**

- Indicamos un almacenamiento máximo de 5Gb
- Como política de acceso al medio, indicamos ReadWriteOnce, de forma que sólo un pod de forma simultánea podrá tener acceso de lectura/escritura
- Mediante afinidad, indicamos el nodo sobre el que operará la unidad de PV
- Indicamos la carpeta en el nodo donde operará la unidad de PV, donde se almacenará el contenido

Creamos el archivo con nombre **nexus-pv.yml** y le agregamos el siguiente contenido:

```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: nexus-pv
  namespace: nexus
  labels:
    type: local
spec:
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteOnce
  volumeMode: Filesystem
  storageClassName: local-storage
  local:
    path: /home/kubernetes/nexus
  nodeAffinity:
    required:
      nodeSelectorTerms:
        - matchExpressions:
          - key: kubernetes.io/hostname
            operator: In
            values:
              - kubernetes-minion1.local
```

Ejecutamos la creación de la unidad de persistencia:

```
$ kubectl apply -f nexus-pv.yml  
persistentvolume/nexus-pv created
```

46.1.2. Creando el namespace para nexus

Para organizar un poco el despliegue de la plataforma, vamos a crear un namespace para nexus.

Creamos el archivo con nombre **nexus-namespace.yml** y le agregamos el siguiente contenido:

```
kind: Namespace  
apiVersion: v1  
metadata:  
  name: nexus
```

Ejecutamos la creación del namespace:

```
$ kubectl apply -f nexus-namespace.yml  
namespace/nexus created
```

46.1.3. Creando el StatefulSet

Seguidamente, vamos a crear el despliegue de la plataforma nexus, utilizando un objeto de Nexus llamado StatefulSet.

Podríamos también haber utilizado un objeto de tipo Deployment, pero en este caso, resulta más interesante el StatefulSet.

Con los StatefulSet, el sistema nexus obtendrá un identificador de red único (id) y por cada statefulset se generará una solicitud de persistencia única (PVC).

Esta forma de desplegar, ofrecerá a nuestro nexus posibilidad de uso en modo de alta disponibilidad, ya que podrá ser reiniciado de forma automática ante un eventual fallo mientras que la instancia mantendrá su identidad de persistencia (PVC) en caso de reinicios o relocalizaciones en nodos distintos.

La principal diferencia entre un StatefulSet y un Deployment, es que cuando se produce un escalado de la aplicación, con un deployment, cada réplica comparte una misma unidad pvc.

Por el contrario, con un StatefulSet cada réplica obtiene de forma separada su propia unidad pvc.

- Indicamos como imagen: 'sonatype/nexus3:3.25.0'
- Indicamos algunas etiquetas como app y group

- Indicamos que vamos a tener 1 réplica
- Indicamos puertos de exposición, contexto de seguridad y datos de la pvc que va a crearse

Creamos el archivo con nombre **nexus-statefulset.yml** y le agregamos el siguiente contenido:



```

apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: nexus
  namespace: nexus
  labels:
    app: nexus
    group: service
spec:
  replicas: 1
  serviceName: nexus
  selector:
    matchLabels:
      app: nexus
  template:
    metadata:
      labels:
        app: nexus
        group: service
    spec:
      securityContext:
        runAsUser: 200
        runAsGroup: 2000
        fsGroup: 2000
      containers:
        - name: nexus
          image: 'sonatype/nexus3:3.25.0'
          ports:
            - containerPort: 8081
              protocol: TCP
            - containerPort: 8123
              protocol: TCP
          volumeMounts:
            - name: nexus-storage
              mountPath: /nexus-data
      volumeClaimTemplates:
        - metadata:
            name: nexus-storage
          spec:
            accessModes:
              - ReadWriteOnce
            storageClassName: local-storage
            resources:
              requests:
                storage: 2Gi

```

Ejecutamos la creación del statefulset:



Fondos Europeos



MINISTERIO
DE EDUCACIÓN, FORMACIÓN PROFESIONAL
Y DEPORTE

Cofinanciado por
la Unión Europea



```
$ kubectl apply -f nexus-statefulset.yml
```

```
statefulset.apps/nexus created
```

Ejecutamos la creación del servicio:

```
$ kubectl apply -f nexus-service.yml
```

```
service/nexus created
```

46.1.4. Creando el servicio para las reglas del Ingress Controller

Eventualmente, nuestro proxy inverso dentro del controlador Ingress debe apuntar a nuestra instancia única de nexus dentro de nuestro StatefulSet.

El nombre de este pod será nexus-0.

Por lo tanto, crearemos un servicio que reduce la selección y solo servirá a nuestro pod.

Creamos el archivo **nexus-service-for-ingress-controller.yml** y le añadimos el siguiente contenido:

```
apiVersion: v1
kind: Service
metadata:
  name: nexus-0-service
  namespace: nexus
spec:
  ports:
    - name: http-main
      port: 8081
      protocol: TCP
      targetPort: 8081
    - name: http-docker-rep
      port: 8123
      protocol: TCP
      targetPort: 8123
  selector:
    app: nexus
    group: service
    statefulset.kubernetes.io/pod-name: nexus-0
```

Ejecutamos la creación del servicio:

```
$ kubectl apply -f nexus-service-for-ingress-controller.yml
```

```
service/nexus-0-service created
```

Podemos obtener los dos endpoints de nuestro servicio, ejecutamos el siguiente comando:

```
$ kubectl -n nexus describe service nexus-0-service | grep Endpoints
```

```
Endpoints: 10.38.0.1:8081  
Endpoints: 10.38.0.1:8123
```

46.1.5. Añadiendo entradas DNS

Ahora, vamos a modificar las entradas DNS de todos los servidores, de forma que añadiremos las siguientes entradas DNS en el archivo **/etc/hosts**:

```
192.168.15.100 nexus.local  
192.168.15.100 nexus-docker.local
```



La ip 192.168.15.100 es la IP de la máquina maestra, pero podríamos poner realmente la IP de cualquiera de los servidores, ya que los servicio ClusterIP están operando a nivel de la totalidad del clúster, es visible e interceptable en todas las máquinas.

Si queremos que esta misma resolución DNS esté presente en todos los servidores, en el master y en los minions

46.1.6. Creando las reglas de Ingress

Utilizaremos un objeto de kubernetes de tipo Ingress Controller para acceder a nuestro servidor nexus desde fuera de nuestro clúster.

De hecho, también usaremos la entrada desde dentro del clúster para mantener los nombres de las imágenes de forma consistente, para poder utilizar el mismo nombre de imagen exactamente tanto dentro como fuera del clúster.

- Indicamos el servicio **nexus-0-service** como el "backend" de destino de la petición http
- Mapeamos el puerto 8123 con el hostname **nexus-docker.local**
- La interfáz de acceso para administración del nexus, estará disponible si indicamos en el navegador web: **nexus.local**
- Indicamos anotaciones para el servicio Ingress
 - No redireccionar protocolo http sobre https
 - Desactivar el tamaño máximo del body de peticiones POST, para no tener problemas de tamaño en las subidas de las capas de las imágenes docker hacia nexus

- Definimos la entrada **spec.tls.hosts** hacia **nexus-docker.local**
 - Si no especificamos esto, las peticiones POST redireccionarán hacia el host default y no se enrutarán de forma correcta hacia **nexus.local**
- Indicamos la anotación **ingress.kubernetes.io/ssl-redirect: "false"**, para evitar que kubernetes automáticamente redireccione a endpoints SSL

Creamos el archivo **nexus-ingress.yml** y le añadimos el siguiente contenido:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    ingress.kubernetes.io/ssl-redirect: "false"
  name: nexus
  namespace: nexus
spec:
  ingressClassName: nginx
  rules:
    - host: nexus.local
      http:
        paths:
          - backend:
              service:
                name: nexus-0-service
                port:
                  number: 8081
              path: /
              pathType: Exact
    - host: nexus-docker.local
      http:
        paths:
          - backend:
              service:
                name: nexus-0-service
                port:
                  number: 8123
              path: /
              pathType: Prefix
  defaultBackend:
    service:
      name: nexus-0-service
      port:
        number: 8081
  tls:
    - hosts:
        - nexus-docker.local
```

Ejecutamos la creación de las reglas de ingress:

```
$ kubectl apply -f nexus-ingress.yml  
ingress.networking.k8s.io/nexus created
```

46.1.7. Obteniendo la contraseña inicial de nuestro sistema nexus

Para poder acceder a nexus, necesitamos obtener la clave inicial que este ha generado.

Cada vez que llevemos a cabo la instalación de un nuevo sistema nexus, la clave será distinta, por que se autogenera en el proceso de instalación.

Ejecutamos el siguiente comando en consola para obtenerla:

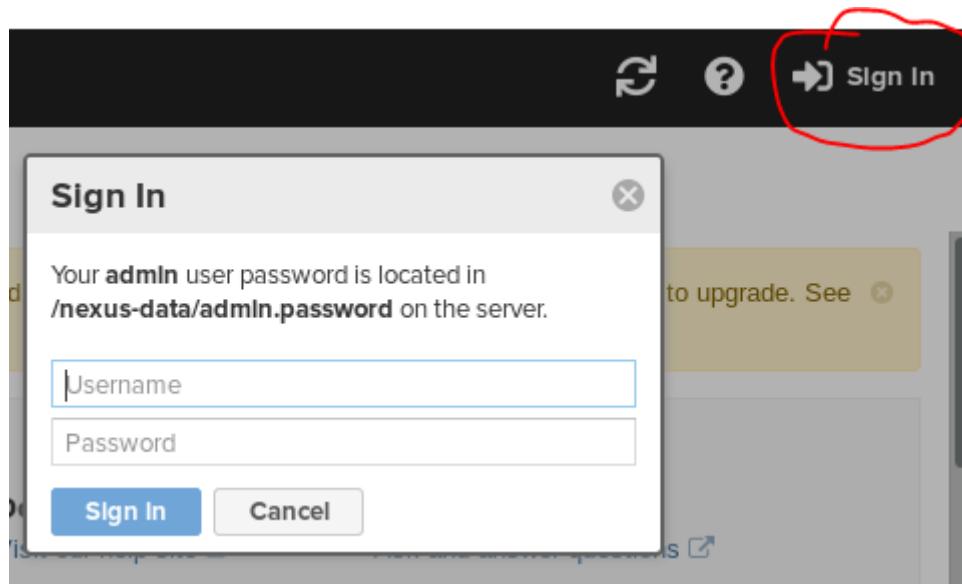
```
$ kubectl -n lab-nexus exec nexus-0 -- cat /nexus-data/admin.password  
56bcb901-f76e-4469-ae81-c98aa68bfd08
```

46.1.8. Accediendo al portal web nexus

Indicamos en nuestro navegador web la siguiente URL: <http://nexus.local>.

Introducimos las siguientes credenciales de acceso:

- **Username**
 - admin
- **Password**
 - El hash de clave que nos haya aparecido en el paso anterior
 - 56bcb901-f76e-4469-ae81-c98aa68bfd08



Una vez dentro, nos aparecerán una serie de pasos a modo de **wizard** la primera vez que accedemos.

- **Paso 1**

- Next

- **Paso 2**

- Indicamos como nueva clave la misma que nos ha salido en el proceso de autogeneración, así evitamos olvidos :)

- **Paso 3**

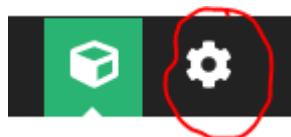
- Aquí configuramos el acceso anónimo a nexus
 - Indicamos **Enable anonymous access**

- **Paso 4**

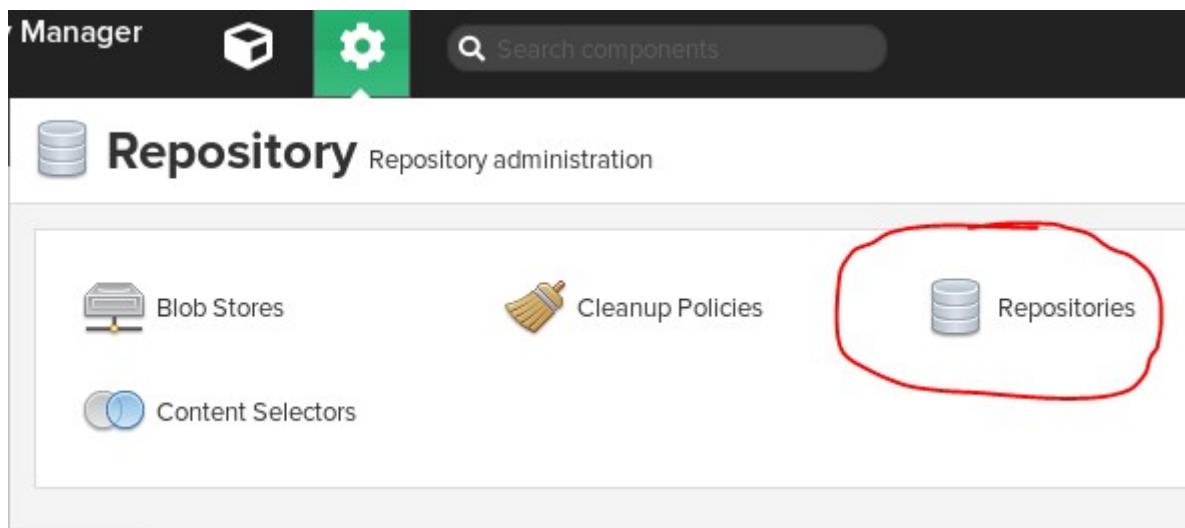
- Pulsamos sobre Finish

46.1.9. Configurando Nexus

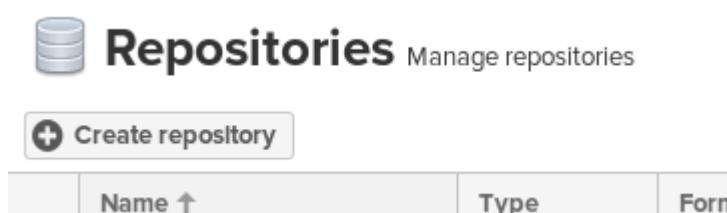
Pulsamos sobre el icono de la rueda dentada, para acceder a la configuración de Nexus.



Hacemos clic sobre la opción de **Repositories**



Pulsamos sobre la opción de **Create repository**



Pulsamos en el tipo de repositorio **docker (hosted)**

Repositories / Select Recipe

Recipe ↑
apt (hosted)
apt (proxy)
bower (group)
bower (hosted)
bower (proxy)
cocoapods (proxy)
conan (proxy)
conda (proxy)
docker (group)
docker (hosted)
docker (proxy)
gitlfs (hosted)

Indicamos los siguientes datos de repositorio:

- Name

- k8s-repo

- HTTP

- 8123

Repositories / Select Recipe / Create Repository: docker (hosted)

Name: A unique identifier for this repository

Online: If checked, the repository accepts incoming requests

Repository Connectors

Connectors allow Docker clients to connect directly to hosted registries, but are not always required. Consult our [documentation](#) for which connector is appropriate for your use case. For information on scaling the repositories see our [scaling documentation](#).

HTTP:

Create an HTTP connector at specified port. Normally used if the server is behind a secure proxy.

8123

HTTPS:

Create an HTTPS connector at specified port. Normally used if the server is configured for https.

Allow anonymous docker pull:

Allow anonymous docker pull (Docker Bearer Token Realm required)

Pulsamos sobre el botón que aparecerá abajo a la izquierda con nombre **Create repository**

46.1.10. Probando una petición contra el repositorio

Vamos a ejecutar el siguiente comando en nuestra consola:

```
$ curl -I -u admin:56bcb901-f76e-4469-ae81-c98aa68bfd08 http://nexus-docker.local/v2/  
  
HTTP/1.1 200 OK  
Server: nginx/1.19.3  
Date: Mon, 11 Jan XXXX 19:31:31 GMT  
Connection: keep-alive  
X-Content-Type-Options: nosniff  
Content-Security-Policy: sandbox allow-forms allow-modals allow-popups allow-presentation allow-scripts allow-top-navigation  
X-XSS-Protection: 1; mode=block  
Docker-Distribution-Api-Version: registry/2.0
```



Tenemos que sustituir el token de contraseña por el que tengamos en ese momento

46.1.11. Configuración del daemon docker

El siguiente paso, va a ser configurar los daemons de docker en todos los servidores para que permitan utilizar conexiones sin cifrar o bien con certificados SSL autogenerados y no nos de problemas la subida ni descarga de imágenes en nuestra infraestructura de red interna.

Creamos el archivo **/etc/docker/daemon.json** y le añadimos el siguiente contenido en todos los servidores:

```
{  
  "insecure-registries": ["nexus-docker.local"]  
}
```

Seguidamente, en cada servidor, reiniciamos el daemon docker para que coja los cambios de configuración:

```
$ sudo systemctl restart docker
```

46.1.12. Login en el nexus docker registry

Para que los diferentes docker engines, puedan descargar las imágenes docker de nuestro repositorio nexus, vamos a utilizar la forma más sencilla de autentificación, basada en usuario/contraseña en todos nuestros servidores.

Ejecutamos el siguiente comando en cada uno de ellos:

```
$ docker login -u admin -p 56bcb901-f76e-4469-ae81-c98aa68bfd08 nexus-docker.local
```

```
...  
Login Succeeded
```

46.1.13. Reetiquetando imagen docker

Descargamos la siguiente imagen docker que posteriormente re-etiquetaremos:

```
$ docker pull edc4it/hello-node
```

A continuación, vamos a proceder con el re-etiquetado de una imagen docker existente, para incluir como prefijo, nuestra URL de repositorio **nexus-docker.local**.

Ejecutamos el siguiente comando en consola:

```
$ docker image tag edc4it/hello-node nexus-docker.local/hello-node
```

Seguidamente, subimos la imagen docker a nuestro repositorio binario nexus:

```
$ docker push nexus-docker.local/hello-node
```

Por último, comprobamos dentro del propio Nexus mediante el buscador, que observamos la imagen hello-node que acabamos de subir al repositorio:

The screenshot shows the Docker Nexus search interface. On the left, there's a sidebar with options: Welcome, Search (which is expanded), Custom, Docker (selected and highlighted in green), Maven, NuGet, and Browse. The main area has a search bar with the placeholder 'Search for components i...' and two dropdown fields: 'Image Name' and 'Content Digest', both set to 'Any'. Below these is a table with one row. The row contains a small icon of a folder, the name 'hello-node', and a 'Name' column header. The entire row is circled in red.

46.2. Lab: Desplegando aplicación con imagen de Nexus

Mediante este laboratorio, vamos a poner en marcha un despliegue que obtiene una imagen que tenemos almacenada en el repositorio local nexus.



Se da por realizado el laboratorio de integración de nexus antes de realizar este.

46.2.1. Creando el namespace

Lo primero que vamos a realizar para llevar a cabo el despliegue de nuestra aplicación, es crear un namespace para tal fin.

Ejecutamos el siguiente comando para crear el namespace:

```
$ kubectl create namespace hello-app-using-nexus  
namespace/hello-app-using-nexus created
```

46.2.2. Creando el secreto (Credenciales de acceso a nexus)

Ahora, vamos a crear un objeto secret en kubernetes para almacenar el usuario y clave de acceso al repositorio nexus, de forma que kubernetes pueda obtener las imágenes de dicho repositorio de forma autenticada.

Ejecutamos el siguiente comando:

```
$ kubectl -n hello-app-using-nexus create secret docker-registry nexus-docker-credentials \  
--docker-server=nexus-docker.local \  
--docker-username=admin \  
--docker-password=56bcb901-f76e-4469-ae81-c98aa68bfd08  
  
secret/nexus-docker-credentials created
```



Deberemos de indicar el usuario y contraseña que tengamos en nuestro nexus específicamente, para escapar caracteres podemos utilizar entrecomillado simple, quedando por ejemplo los parámetros de esta forma: --docker-password='123'

46.2.3. Creando el deployment de la aplicación

Ahora, toca el turno de crear el deployment de nuestra aplicación.

La idea, es que kubernetes determine mediante el planificador el mejor nodo candidato para poner en marcha las réplicas del sistema, vaya a nexus, descargue la imagen docker y se autentifique contra este mediante las credenciales que hemos indicado en el secret.

Creamos un archivo con nombre **deployment-hello-app-using-nexus.yml** y le añadimos el siguiente contenido:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: hello-node
  name: hello-node-deployment
  namespace: hello-app-using-nexus
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hello-node-app
  template:
    metadata:
      labels:
        app: hello-node-app
    spec:
      containers:
        - image: nexus-docker.local/hello-node
          imagePullPolicy: Always
          name: hello-node
      imagePullSecrets:
        - name: nexus-docker-credentials
```

Ejecutamos el siguiente comando:

```
$ kubectl apply -f deployment-hello-app-using-nexus.yml
deployment.apps/hello-node-deployment created
```

Para probar una petición y observar que la aplicación funciona, ejecutamos el siguiente comando para obtener la ip del pod:

```
$ kubectl get pod -o wide -n hello-app-using-nexus
NAME                  READY   STATUS    RESTARTS   AGE   IP           NODE
NOMINATED NODE   READINESS GATES
hello-node-deployment-8b698cccb-w6bvn   1/1   Running   0    11m  10.38.0.5  kubernetes-minion1.local <none>
<none>
```

Finalmente, ejecutamos una petición con curl hacia la aplicación:

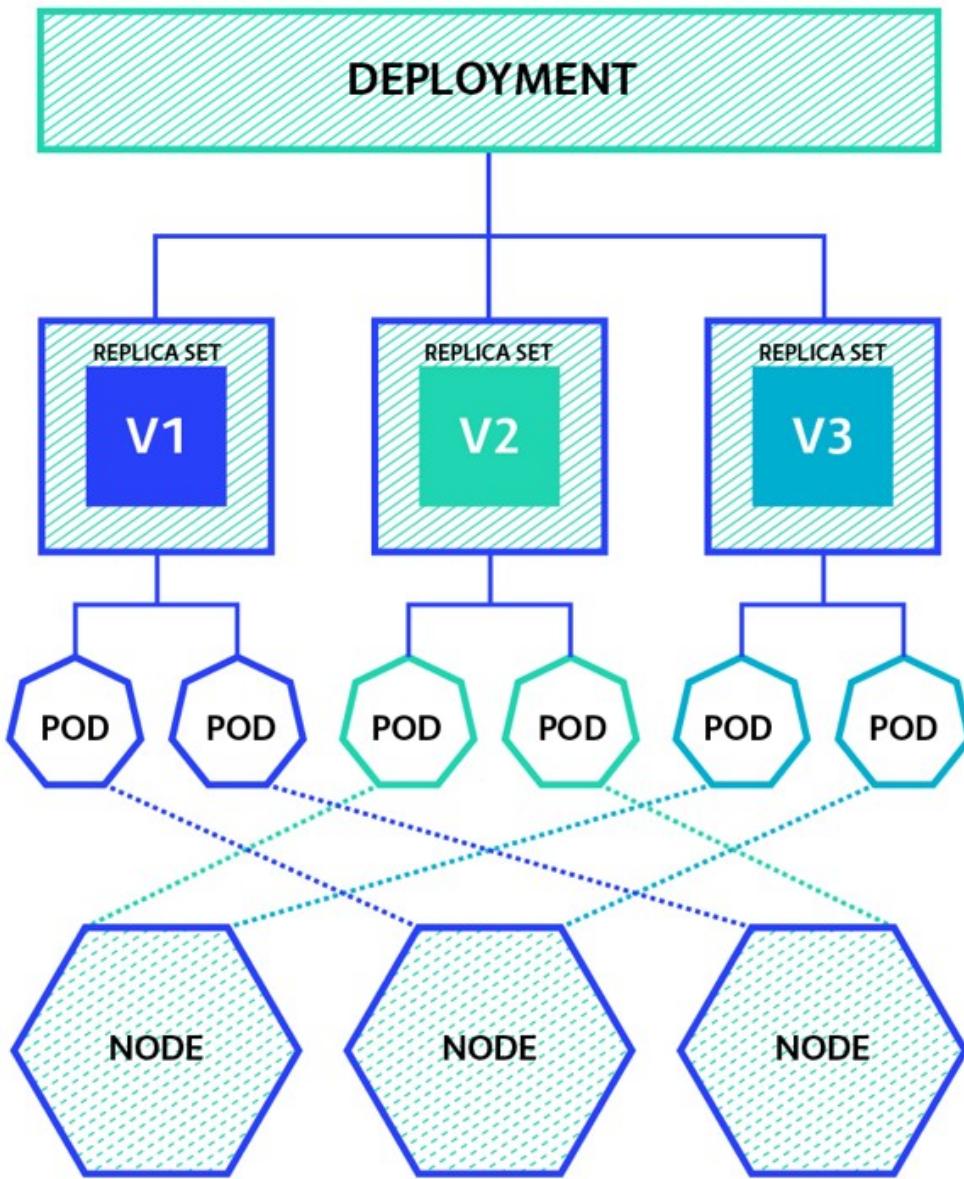


```
$ curl http://10.38.0.5:3030
```

```
hello friend
```



Capítulo 47. Deployments



Utilizando los elementos de control de Kubernetes ReplicationController y ReplicaSet hemos comprobado que nos otorgan capacidad de replicación variable, pero hay ciertas operaciones en la casuística diaria que posiblemente necesitemos utilizar y este tipo de unidades de control se nos quedan cortas

El uso de los Replication Controller en la vida real implicaba la creación de un RC por cada versión de aplicación y la gestión manual de versiones, actualizaciones o rollbacks

Con kubectl se podía realizar un rolling update contra el replication controller, pero la idea es que si necesitas hacer eso... ¡Pásate al Deployment! :D

47.1. Características del uso de Deployments

- Los deployments nos permiten realizar rolling updates y rollbacks
- Se trata de una abstracción superior al ReplicationController
- En vez de usar un ReplicationController, la unidad de deployment usa un ReplicaSet
- Los replicaSets controlan los Pods
 - Podemos actualizar el template esquemático de creación de nuevos Pods y actualizar el caliente el propio Deployment, sin tener que eliminarlo y crear otro nuevo
- Pausar el deployment
 - Por si tenemos que llevar a cabo ciertas correcciones en los templates de los Pods que vamos a desplegar
- Limpieza de ReplicaSet
 - Podemos llevar a cabo limpieza de antiguos ReplicaSets que ya no vamos a utilizar más

47.2. Implicaciones

- La construcción de un Deployment implica que cuando creamos un deployment de un pod, realmente se crea un replicaset con los pods replicados que queremos
- Si queremos agregar un nuevo pod, modificamos el fichero de despliegue, y por defecto al actualizar, crea un segundo replicaSet y por cara pod iniciado en el nuevo replicaSet se apaga (dejan de dar servicio)
- Si queremos invertir el proceso (rollback), de nuevo inicia los replicaSet en el anterior apagando los nuevos.
- La premisa es que jamás podemos de dar servicio :)

Capítulo 48. Lab: Deployments

Mediante este laboratorio, trabajaremos la unidad Deployments de Kubernetes con diferentes casuísticas

48.1. Creando 1 Deployment con 1 container

Creamos el archivo **deployment-with-1-container.yml** e indicamos el siguiente contenido:

- Servidor web nginx a su última versión
- 2 réplicas

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deployment-with-1-container
  labels:
    department: engineering
spec:
  replicas: 2
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
        - name: nginx
          image: nginx:latest
          ports:
            - containerPort: 80
```

Ejecutamos la creación

```
$ kubectl apply -f deployment-with-1-container.yml
deployment.apps/deployment-with-1-container created
```

48.2. Análisis de estructura mínima

Como observamos, el Deployment ha introducido nuevos elementos que pasamos a describir a continuación (comparte parecido a la estructura del ReplicaSet):

- **apiVersion**



- apps/v1

 La versión v1 en sí no posee la especificación de Deployment

Tenemos que subir la versión a apps/v1

- **kind**

- Deployment, indicamos la nueva unidad a utilizar

- **spec → replicas**

- Indicamos el número de Pods que tendremos en el estado final a desplegar

- **spec → selector**

- Indicamos un selector de etiqueta, con formato key-value, esto le vale a Kubernetes en la unidad de replicación, para encontrar Pods que esta unidad tenga que gestionar

- **spec → template → metadata**

- Especificamos las etiquetas que a modo de metadata tendrán los Pods cuando se creen
- La sección del template realmente crea una plantilla, de modo que los siguientes Pods que sea necesario crear, cumplirán todos la misma estructura

- **spec → template → spec**

- Especificamos en la plantilla los contenedores que tendrá dentro el Pod, su imagen, puertos, etc.

48.3. Listando los Pods creados por el Deployment

Comprobamos que efectivamente se han creado 2 Pods, ya que indicamos en la metadata del .yml replicas: 3

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
deployment-with-1-container-668764bd67-4mv4s	1/1	Running	0	5m47s
deployment-with-1-container-668764bd67-xtb26	1/1	Running	0	5m47s

48.4. Listado las unidades Deployment

Ahora, pasamos a listar las unidades Deployment que tengamos en el sistema

El sistema debería de indicarnos que efectivamente tenemos 1 de deployment, compuesto por 2 Pods, y que los 2 Pods están listos

```
$ kubectl get deployments
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
deployment-with-1-container	2/2	2	2	4m18s

48.5. Describiendo la unidad de Deployment

Ahora, vamos a realizar un describe al Deployment

```
$ kubectl describe deployment deployment-with-1-container
```

Name: deployment-with-1-container
Namespace: default
CreationTimestamp: Sun, 29 Sep 2019 20:45:19 +0200
Labels: department=engineering
Annotations: deployment.kubernetes.io/revision: 1
 kubectl.kubernetes.io/last-applied-configuration...

Selector: app=frontend
Replicas: 2 desired | 2 updated | 2 total | 2 available | 0 unavailable
StrategyType: RollingUpdate
MinReadySeconds: 0
RollingUpdateStrategy: 25% max unavailable, 25% max surge
Pod Template:
Labels: app=frontend
Containers:
nginx:
 Image: nginx:latest
 Port: 80/TCP
 Host Port: 0/TCP
 Environment: <none>
 Mounts: <none>
 Volumes: <none>
Conditions:
Type Status Reason

Available True MinimumReplicasAvailable
Progressing True NewReplicaSetAvailable
OldReplicaSets: <none>
NewReplicaSet: deployment-with-1-container-668764bd67 (2/2 replicas created)
Events:
Type Reason Age From Message

Normal ScalingReplicaSet 8m42s deployment-controller Scaled up replica set
deployment-with-1-container-668764bd67 to 2

Observamos como el describe del deployment nos indica interesantes datos:

- **Name**

- Nombre del deployment

- **Namespace**

- Espacio de nombres en el que se encuentra la unidad deployment

- **Selector**

- Selector con formato key-value que utilizará el deployment para detectar sus Pods, de forma que los identificará como suyos

- **Labels**

- Etiquetas que la unidad deployment tiene asignada como metadata propia

- **Replicas**

- Indica el número de réplicas actualmente operativas / Réplicas ordenadas a disponer

- **StrategyType**

- Indica el tipo de estrategia que seguirá el deployment, estrategia de actualización y rollback disponible

- **RollingUpdateStrategy**

- Indica el porcentaje de número de Pods que como máximo pueden no estar operativos ante un Update o Rollback (25% por defecto)
- Indica también el máximo porcentaje de Pods a que realizarles de golpe tratamientos de Update o Rollback (25% por defecto)

- **Pod Template**

- Indica la estructura que tendrán los nuevos Pods que la unidad deployment vaya a crear

- **Events**

- Histórico de eventos del Deployment
- A la unidad de Deployment no le interesa mostrar el detalle de cada Pod (Si se le ha hecho pull a la imagen, en qué Nodo se ha alojado, etc)
- Tampoco le interesa, si un ReplicaSet ha tenido que llevar a cabo ciertas operaciones para poner en marcha el despliegue
- A la unidad de Deployment lo que le interesa mostrar si la orden de Deployment ha sido ejecutado, número de réplicas escalas, su nombre, desde cuando y si está operando normalmente

48.6. Escalando el número de réplicas

Vamos a dar una orden a la unidad deployment para que aumente el número de réplicas de la unidad, pasando de 2 a 3

- Mediante archivo de manifiesto

Modificamos el archivo de manifiesto **deployment-with-1-container.yml** e indicamos el siguiente contenido

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: deployment-with-1-container
  labels:
    department: engineering
spec:
  replicas: 3
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
        - name: nginx
          image: nginx:latest
          ports:
            - containerPort: 80

```

Ejecutamos la actualización

```
$ kubectl apply -f deployment-with-1-container.yml
deployment.apps/deployment-with-1-container configured
```

Comprobamos que hemos aumentado el número de réplicas de 3 a 5

```
$ kubectl get deployments
NAME           READY  UP-TO-DATE  AVAILABLE  AGE
deployment-with-1-container  3/3   3         3         23m
```

48.7. Describiendo la unidad de replicación (ReplicaSet)

Una vez hemos ejecutado el deployment, vamos a listar las unidades de replicación (rs)

```
$ kubectl get rs
NAME           DESIRED  CURRENT  READY  AGE
deployment-with-1-container-668764bd67  3     3     3     25m
```

- El nombre es el mismo que el de despliegue pero con un hash

Ahora, vamos a describir la unidad rs que el deployment nos ha creado

```
$ kubectl describe rs deployment-with-1-container-668764bd67

Name:      deployment-with-1-container-668764bd67
Namespace:  default
Selector:   app=frontend,pod-template-hash=668764bd67
Labels:     app=frontend
            pod-template-hash=668764bd67
Annotations: deployment.kubernetes.io/desired-replicas: 3
              deployment.kubernetes.io/max-replicas: 4
              deployment.kubernetes.io/revision: 1
Controlled By: Deployment/deployment-with-1-container
Replicas:    3 current / 3 desired
Pods Status: 3 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:  app=frontend
           pod-template-hash=668764bd67
Containers:
  nginx:
    Image:  nginx:latest
    Port:   80/TCP
    Host Port:  0/TCP
    Environment: <none>
    Mounts:  <none>
    Volumes: <none>
Events:
  Type  Reason        Age   From          Message
  ----  ----        --   --           --
Normal  SuccessfulCreate  27m  replicaset-controller  Created pod: deployment-with-1-container-668764bd67-4mv4s
Normal  SuccessfulCreate  27m  replicaset-controller  Created pod: deployment-with-1-container-668764bd67-xtb26
Normal  SuccessfulCreate  6m31s replicaset-controller  Created pod: deployment-with-1-container-668764bd67-qr9mf
```

48.8. Comprobando que responde algún Pod con IP interna

Vamos a listar los Pods que tengamos operando, seleccionamos uno y le vamos a hacer un Curl a la IP/Puerto interno del Pod para comprobar que el servidor web nginx está dando servicio

```
$ kubectl get pods

NAME                      READY   STATUS    RESTARTS   AGE
deployment-with-1-container-668764bd67-4mv4s  1/1     Running   0          30m
deployment-with-1-container-668764bd67-qr9mf   1/1     Running   0          8m56s
deployment-with-1-container-668764bd67-xtb26   1/1     Running   0          30m
```

A continuación, le hacemos un describe a alguno, para conocer su IP (al primero que tengamos en la lista por ejemplo)

```
$ kubectl describe pod deployment-with-1-container-668764bd67-4mv4s
```

Name: deployment-with-1-container-**668764bd67-4mv4s**
Namespace: default
Priority: 0
Node: kubeminion2/**192.168.15.102**
Start Time: Sun, 29 Sep 2019 20:45:19 +0200
Labels: app=frontend
pod-template-hash=**668764bd67**
Annotations: <none>
Status: Running
IP: **10.44.0.1**
IPs:
IP: **10.44.0.1**
Controlled By: ReplicaSet/deployment-with-1-container-**668764bd67**
Containers:
nginx:
Container ID: docker
://f09383ab9d2947b72d60530bf231fdb5689f1148c2663eb1933ac0bb52a3d900
Image: nginx:latest
Image ID: docker-pullable
://nginx@sha256:aeded0f2a861747f43a01cf1018cf9efe2bdd02af57d2b11fcc7fcadc16ccd1
Port: 80/TCP
Host Port: 0/TCP
State: Running
Started: Sun, 29 Sep 2019 20:45:26 +0200
Ready: True
Restart Count: 0
Environment: <none>
Mounts:
/var/run/secrets/kubernetes.io/serviceaccount from default-token-hpbbg (ro)
Conditions:
Type Status
Initialized True
Ready True
ContainersReady True
PodScheduled True
Volumes:
default-token-hpbbg:
Type: Secret (a volume populated by a Secret)
SecretName: default-token-hpbbg
Optional: false
QoS Class: BestEffort
Node-Selectors: <none>
Tolerations: node.kubernetes.io/not-ready:NoExecute **for 300s**
node.kubernetes.io/unreachable:NoExecute **for 300s**
Events:

Type	Reason	Age	From	Message
Normal	Scheduled	<unknown>	default-scheduler	Successfully assigned default
/deployment-with-1-container-668764bd67-4mv4s				to kubeminion2
Normal	Pulling	34m	kubelet, kubeminion2	Pulling image " nginx:latest "
Normal	Pulled	34m	kubelet, kubeminion2	Successfully pulled image

```
"nginx:latest"
Normal Created 34m    kubelet, kubeminion2 Created container nginx
Normal Started 34m    kubelet, kubeminion2 Started container nginx
```

Nos quedamos con el valor de la IP y puerto ocupado internamente, en nuestro caso **10.44.0.1** y puerto **80**

Ejecutamos un curl

```
$ curl -I 10.44.0.1:80

HTTP/1.1 200 OK
Server: nginx/1.17.4
Date: Sun, 29 Sep 2019 19:21:14 GMT
Content-Type: text/html
Content-Length: 612
Last-Modified: Tue, 24 Sep 2019 14:49:10 GMT
Connection: keep-alive
ETag: "5d8a2ce6-264"
Accept-Ranges: bytes
```

48.9. Creando un Deployment con directivas de Update

Para esta práctica, vamos a crear un segundo deployment

Creamos el archivo **deployment-with-1-container-update.yml** y le agregamos el siguiente contenido



```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: deployment-with-1-container-update
  labels:
    department: engineering
spec:
  replicas: 2
  minReadySeconds: 10
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1
  selector:
    matchLabels:
      app: frontend-update-rollback
  template:
    metadata:
      labels:
        app: frontend-update-rollback
    spec:
      containers:
        - name: nginx
          image: nginx:latest
          ports:
            - containerPort: 80

```

Observamos que hemos agregado nuevos elementos, los comentamos

- **spec → minReadySeconds:** Indica que va a esperar 10 segundos por cada actualización de Pod, antes de pasar a actualizar el siguiente, kubernetes considera este tiempo de cortesía para determinar que el Pod está totalmente operativo
- **spec → Strategy → type:** Indicamos el tipo de estrategia, la de RollingUpdate, esta estrategia definida es la que se aplica por defecto, la hemos indicado de forma explícita
- **spec → Strategy → rollingUpdate → maxUnavailable:** Indicamos de forma explícita cuántos Pods estarán fuera de servicio cuando se inicen procedimientos de actualización o rollback de los sistemas
- **spec → Strategy → rollingUpdate → maxSurge:** Indicamos de forma explícita de "cuantos en cuantos" Pods le realizaremos el tratamiento de actualización o rollback

48.10. Desplegando un Deployment con directivas de Update (rollout status)

Ahora, toca el momento de desplegar, pero vamos a introducir una anotación posterior a modo de **CHANGE-CAUSE**, la idea, es que podamos dejar constancia de la última actualización que ha hecho

el sistema con alguna indicación por nuestra parte, por si hubiera que realizar una regresión rápida al punto anterior.

Ejecutamos en consola lo siguiente

```
$ kubectl apply -f deployment-with-1-container-update.yml  
deployment.apps/deployment-with-1-container-update created
```

Indicamos una anotación para dejar una nota personal sobre lo ocurrido:

```
$ kubectl annotate deploy/deployment-with-1-container-update kubernetes.io/change-cause='Deploy image nginx:latest'
```

Comprobamos que el deployment se ha llevado a cabo de forma correcta

```
$ kubectl rollout status deployment deployment-with-1-container-update  
deployment "deployment-with-1-container-update" successfully rolled out
```

A continuación, vamos a actualizar el despliegue, vamos a indicar que en lugar de la versión **latest** de nginx, queremos la versión **1.17.4-alpine**

Actualizamos el archivo **deployment-with-1-container-update.yml** quedando así



Fondos Europeos

MINISTERIO DE EDUCACIÓN, FORMACIÓN PROFESIONAL Y DEPORTE

Gobierno de España
Cofinanciado por la Unión Europea



```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: deployment-with-1-container-update
  labels:
    department: engineering
spec:
  replicas: 2
  minReadySeconds: 10
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1
  selector:
    matchLabels:
      app: frontend-update-rollback
  template:
    metadata:
      labels:
        app: frontend-update-rollback
    spec:
      containers:
        - name: nginx
          image: nginx:1.17.4-alpine
          ports:
            - containerPort: 80

```

Ejecutamos en consola la actualización

```

$ kubectl apply -f deployment-with-1-container-update.yml
deployment.apps/deployment-with-1-container-update configured

```

Ahora, también indicamos una anotación para dejar una nota personal sobre lo ocurrido:

```

$ kubectl annotate deploy/deployment-with-1-container-update kubernetes.io/change-cause='Change image to nginx:1.17.4-alpine'

```

48.11. Llevando a cabo una operación de Rollback (rollout history)

Vamos a consultar el histórico de despliegues de la unidad con nombre **deployment-with-1-container-update** para ver los rollbacks a los que podemos retroceder

Ejecutamos en consola lo siguiente



- Observamos que tenemos disponibles 2 puntos de revisión

```
$ kubectl rollout history deployment deployment-with-1-container-update
deployment.apps/deployment-with-1-container-update
REVISION CHANGE-CAUSE
1 Deploy image nginx:latest
2 Change image to nginx:1.17.4-alpine
```

Indicamos a kubernetes, que ejecute una regresión rápida a alta velocidad, de la revisión 2 a la revisión 1

```
$ kubectl rollout undo deployment deployment-with-1-container-update --to-revision=1
deployment.apps/deployment-with-1-container-update rolled back
```

48.12. Pausando una operación de actualización (rollout pause)

En alguna ocasión, puede que nos ocurra que damos la orden de actualización de los sistemas, y tenemos que detener el proceso en seco, por que queremos llevar a cabo alguna comprobación

Vamos a actualizar el archivo **deployment-with-1-container-update.yml** y vamos a cometer un error a propósito, que nos vamos a dar cuenta del mismo a los pocos segundos de haber pulsado la tecla "INTRO", como si de un drop database se tratara y no te acordaste de cambiar el connection string a pre - producción XD

Indicamos que en lugar de subir la versión de alpine... Cambiamos la imagen a un apache (httpd) en su última versión e indicamos 5 réplicas

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: deployment-with-1-container-update
  labels:
    department: engineering
spec:
  replicas: 5
  minReadySeconds: 10
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1
  selector:
    matchLabels:
      app: frontend-update-rollback
  template:
    metadata:
      labels:
        app: frontend-update-rollback
    spec:
      containers:
        - name: nginx
          image: httpd:latest
          ports:
            - containerPort: 80

```

Abrimos 2 terminales

En una terminal, ejecutamos la orden de actualización del despliegue

```

$ kubectl apply -f deployment-with-1-container-update.yml
deployment.apps/deployment-with-1-container-update configured

```

Procedemos a indicar alguna anotación al respecto:

```
$ kubectl annotate deploy/deployment-with-1-container-update kubernetes.io/change-cause='Update system to httpd:latest'
```

En la otra terminal, indicamos una pausa del proceso de deployment una vez en la primera consola hayamos indicado la orden

```

$ kubectl rollout pause deployment deployment-with-1-container-update
deployment.apps/deployment-with-1-container-update paused

```

Si consultamos los Pods, veremos algo así

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
deployment-with-1-container-update-5bf8bb6848-6r5c2	1/1	Running	0	86s
deployment-with-1-container-update-5bf8bb6848-z5frb	1/1	Running	0	86s
deployment-with-1-container-update-86c8b4979b-7szlf	1/1	Running	0	8m56s
deployment-with-1-container-update-86c8b4979b-j5498	1/1	Running	0	17m
deployment-with-1-container-update-86c8b4979b-kj69p	1/1	Running	0	17m
deployment-with-1-container-update-86c8b4979b-px9bb	1/1	Running	0	8m56s

- Observaremos quizás más Pods de 5 ordenados
- Algunos Pods serán aún de nginx y otros de http, podemos comprobarlo si le hacemos un describe a algunos Pods y observamos la imagen que tienen los contenedores

Consultamos el status del deployment, y observamos que está ahora mismo congelado

```
$ kubectl rollout status deployment deployment-with-1-container-update
```

Waiting for deployment "deployment-with-1-container-update" rollout to finish: 2 out of 5 new replicas have been updated...

48.13. Resumiendo una operación de actualización pausada (rollout resume)

A continuación, vamos a "darle al play" para que continúe la operación del deployment anterior

```
$ kubectl rollout resume deployment deployment-with-1-container-update  
deployment.apps/deployment-with-1-container-update resumed
```

Lanzamos rápidamente la visualización de los Pods, y observamos de nuevo movimiento

```
$ kubectl get pods
```

NAME	READY	STATUS	
RESTARTS	AGE		
deployment-with-1-container-update-5bf8bb6848-6r5c2	1/1	Running	0
3m3s			
deployment-with-1-container-update-5bf8bb6848-c6xv4	0/1	ContainerCreating	0
4s			
deployment-with-1-container-update-5bf8bb6848-z5frb	1/1	Running	0
3m3s			
deployment-with-1-container-update-5bf8bb6848-zhpr	0/1	ContainerCreating	0
4s			
deployment-with-1-container-update-86c8b4979b-7szlf	1/1	Terminating	0
10m			
deployment-with-1-container-update-86c8b4979b-j5498	1/1	Running	0
19m			
deployment-with-1-container-update-86c8b4979b-kj69p	1/1	Running	0
19m			
deployment-with-1-container-update-86c8b4979b-px9bb	1/1	Terminating	0
10m			

Consultamos nuevamente el estado de la operación de actualización del deployment, y observamos que nos indica que está completado

```
$ kubectl rollout status deployment deployment-with-1-container-update  
deployment "deployment-with-1-container-update" successfully rolled out
```

48.14. Reiniciando de nuevo la operación de despliegue (rollout restart)

Vamos a indicar que queremos que se lleve de nuevo la secuencia completa de actualización de los sistemas, reiniciando el proceso

En este caso es diferente, no queremos un rollback anterior, si no, un relanzamiento del proceso rollout tal y como se especificó la última vez

```
$ kubectl rollout restart deployment deployment-with-1-container-update  
deployment.apps/deployment-with-1-container-update restarted
```

Mientras, ejecutamos de nuevo una operación de obtención de Pods, y observaremos como se produce movimiento de nuevo

```
$ kubectl get pods
```

NAME	READY	STATUS	
RESTARTS	AGE		
deployment-with-1-container-update-5bf8bb6848-6r5c2	1/1	Running	0
6m2s			
deployment-with-1-container-update-5bf8bb6848-c6xv4	1/1	Terminating	0
3m3s			
deployment-with-1-container-update-5bf8bb6848-z5frb	1/1	Running	0
6m2s			
deployment-with-1-container-update-5bf8bb6848-zhpr	1/1	Terminating	0
3m3s			
deployment-with-1-container-update-77cbb6f48d-2zdcj	1/1	Running	0
17s			
deployment-with-1-container-update-77cbb6f48d-c44ch	1/1	Running	0
17s			
deployment-with-1-container-update-77cbb6f48d-g9w9b	0/1	ContainerCreating	0
1s			
deployment-with-1-container-update-77cbb6f48d-tnq4g	0/1	ContainerCreating	0
1s			

48.15. Eliminando los deployments

Ahora, vamos a listar las unidades deployments que tengamos

```
$ kubectl get deployment
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
deployment-with-1-container	3/3	3	3	82m
deployment-with-1-container-update	2/2	2	2	28m

Y procedemos a su eliminación

```
$ kubectl delete deployment deployment-with-1-container
```

```
deployment.apps "deployment-with-1-container" deleted
```

```
$ kubectl delete deployment deployment-with-1-container-update
```

```
deployment.apps "deployment-with-1-container-update" deleted
```

Por último, verificamos que no hay deployments activos

```
$ kubectl get deployments
```

```
No resources found in default namespace.
```

48.16. Despliegue ReadinessProbe

Otro caso de uso interesante que podemos tener, es que, temporalmente los Pods no pueden servir tráfico hasta pasado un tiempo.

Por ejemplo, una aplicación puede necesitar cargar un volumen de datos considerable para ponerse en marcha, archivos de configuración durante el inicio, depender de servicios externos después del inicio, etc.

En tales casos se nos da la casuística de que, no queremos eliminar la aplicación, pero tampoco queremos que esté disponible (Niño... ¡Báñate pero no te mojes!).

Kubernetes proporciona sondas de preparación del Pod, cuya misión es detectar y mitigar estas situaciones.

Un pod con contenedores que informan que no están listos no recibe tráfico a través de los servicios de Kubernetes.

Vamos a crear el archivo **deployment-with-readiness-probe.yml** y le agregamos el siguiente contenido:

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: deployment-with-readiness-probe
  labels:
    department: engineering
spec:
  replicas: 2
  minReadySeconds: 10
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1
  selector:
    matchLabels:
      app: frontend-update-rollback
  template:
    metadata:
      labels:
        app: frontend-update-rollback
    spec:
      containers:
        - name: tomcat
          image: tomcat:7
          ports:
            - containerPort: 8080
      readinessProbe:
        periodSeconds: 120
        httpGet:
          path: /
          port: 80

```

- **periodSeconds**

- Cada cuanto tiempo tiene que realizar el sondeo de que la APP está totalmente levantada

- **httpGet**

- Sondeamos el protocolo HTTP
- Petición al raíz /
- Petición hacia el puerto 80

Ejecutamos la creación del deployment:

```
$ kubectl apply -f deployment-with-readiness-probe.yml
deployment.apps/deployment-with-readiness-probe created
```

Ahora, vamos a comprobar el deployment:

```
$ kubectl get deployments
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
deployment-with-readiness-probe	0/2	2	0	4m35s

Y comprobamos los Pods que tengamos en operación:

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
deployment-with-readiness-probe-78444d8c68-jrdht	0/1	Running	0	5m16s
deployment-with-readiness-probe-78444d8c68-t4mgd	0/1	Running	0	5m16s

- Observamos que los Pods están en funcionamiento **Running**, pero sin embargo, el indicador READY dice que el contenedor no está operativo

El despliegue va a quedarse en este estado, por que la configuración del readinessProbe intenta sondear cada 120 segundos el puerto 80, el servidor de aplicaciones tomcat está operando en el puerto 8080, con lo cual, va a permanecer en este estado de forma indefinida

Vamos a reajustar el deployment, cambiando en la sección del **readinessProbe.httpGet.port** en lugar del puerto 80, indicamos el 8080

Actualizamos el deployment:

```
$ kubectl apply -f deployment-with-readiness-probe.yml
```

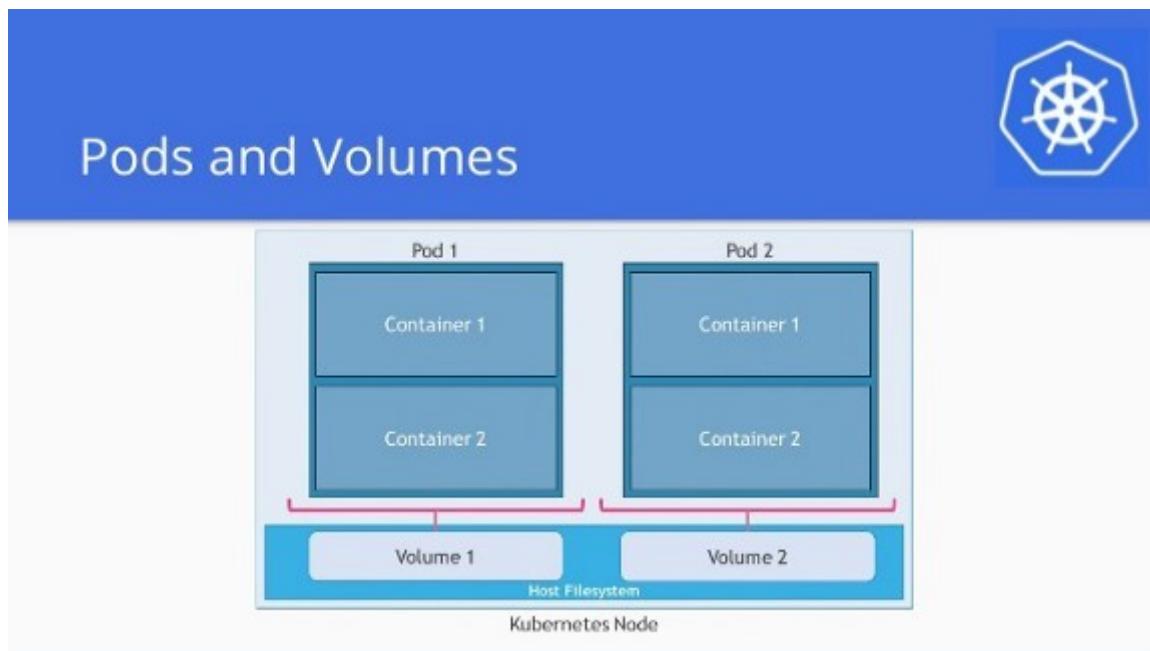
deployment.apps/deployment-with-readiness-probe configured

Dejamos pasar unos 2 minutos aprox, y obtenemos de nuevo el conjunto de Pods para ver que el indicador de READY ya estaría en 1/1:

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
deployment-with-readiness-probe-68cb4df656-6vm62	1/1	Running	0	4m22s
deployment-with-readiness-probe-68cb4df656-7g4hp	1/1	Running	0	4m22s

Capítulo 49. Volumen



En esencia, un volumen es un directorio del host accesible desde todos los contenedores que hagan uso de mismo, montados estos sobre un Pod

La forma en que se crea ese directorio, el medio que lo respalda y su contenido, están determinados por el tipo de volumen particular utilizado

El sistema de archivo que monta un volumen local sin hacer uso de un volumen, es efímero, y los datos una vez el contenedor desaparece, también lo harán los datos

El volumen sobrevive siempre a la muerte de cualquier contenedor que se encuentre computando dentro de un Pod

Kubernetes admite varios tipos de volúmenes, un Pod puede utilizar cualquiera de ellos de forma simultánea

49.1. Volúmenes Docker vs Kubernetes

49.1.1. Volúmenes en Docker

Docker posee también el concepto de volumen, en el caso de Docker resulta un tanto más flexible y menos administrado que con Kubernetes

En Docker, un volumen es simplemente un directorio en el disco o en otro contenedor, el ciclo de vida no tiene mantenimiento y hasta hace relativamente poco, únicamente se disponían de volúmenes de ámbito local

Actualmente Docker proporciona controladores de volumen, pero la funcionalidad aún sigue siendo algo limitada

A partir de la versión de Docker 1.7 sólo se permite un controlador de volumen por contenedor y

desapareció la posibilidad de pasar parámetros a los volúmenes

49.1.2. Volúmenes en Kubernetes

El volumen en Kubernetes tiene un ciclo de vida explícito, el mismo que el Pod que lo engloba

Como consecuencia, un volumen sobrevive a cualquier contenedor que se ejecute dentro de un Pod y los datos son conservados ante reinicios del contenedor

Por supuesto, cuando un Pod deja de existir, el volumen también lo hace

Kubernetes admite muchos tipos de volumen y un Pod puede usar cualquier número de ellos de forma simultánea

49.2. ¿Cómo usa el volumen un Pod?

En esencia, un volumen es sólo un directorio posiblemente con algunos datos creados al que pueden acceder los contenedores en un Pod

La forma en que se crea ese directorio, el medio que lo respalda y su contenido están determinados por el tipo de volumen particular utilizado

Para utilizar un volumen, el Pod debe de especificar que volúmenes proporciona al Pod en sí, mediante el tag `.spec.volumes`

Por otra parte, también debe de especificar dónde montarlos en los contenedores, mediante el campo `.spec.containers.volumeMounts`

Un proceso dentro de un contenedor tiene una visión interna del sistema de archivos compuesta de su imagen (FROM) junto con volúmenes de Docker

La imagen Docker está situada en la raíz de la jerarquía del sistema de archivos, de forma que cualquier volumen se monta en las rutas especificadas dentro del contenedor

Los volúmenes no pueden montarse en otros volúmenes

Cada contenedor en el Pod debe especificar independientemente dónde montar cada volúmen

49.3. Tipos de volúmenes soportados por Kubernetes

49.3.1. Ofertados por proveedores de nube privada

- `awsElasticBlockStore`
- `azureDisk`
- `azureFile`
- `gcePersistentDisk`

49.3.2. Con características de sistema de archivo distribuído

- cephfs
- glusterfs

49.3.3. Para poder utilizarlo en nube propia (OpenStack)

- cinder

49.3.4. Locales al nodo Kubernetes

- emptyDir
- hostPath
- local

49.3.5. Compartición de archivos

- nfs

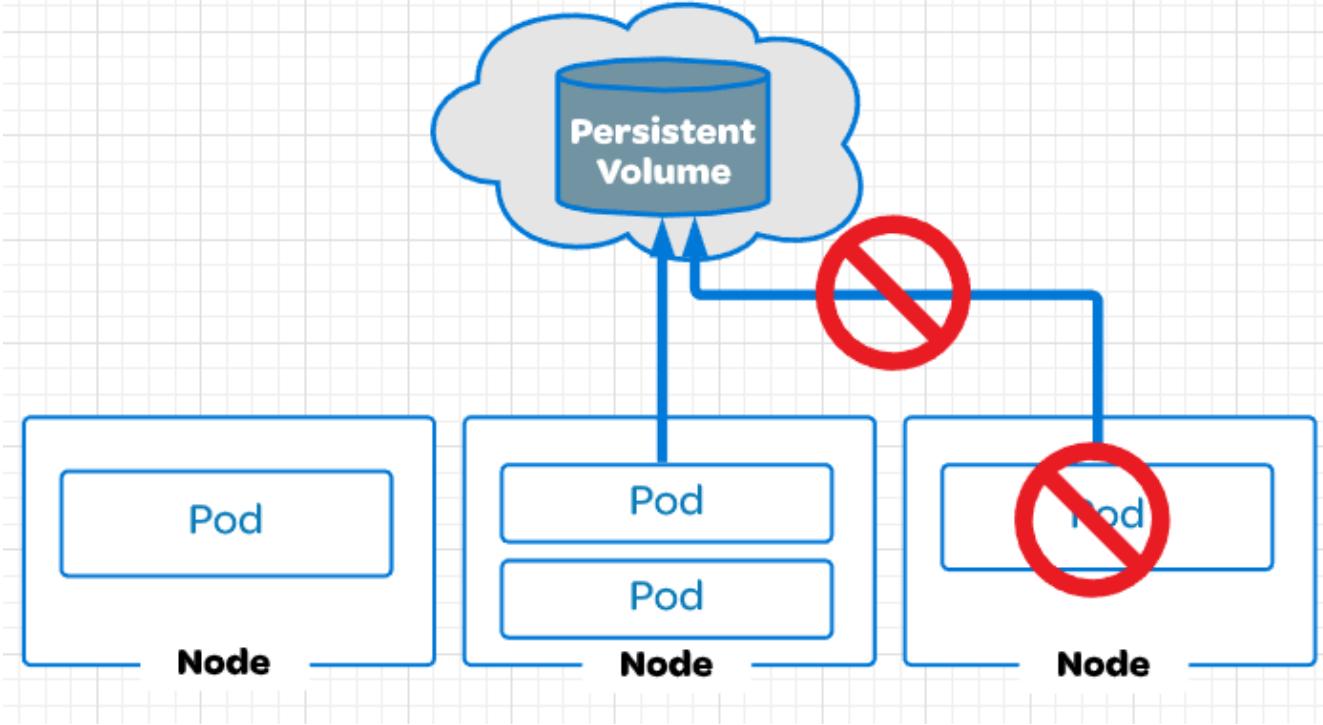
49.3.6. Otros tipos

- fc (fibre channel)
- flexVolume
- flocker
- configMap
- csi
- downwardAPI
- iscsi
- persistentVolumeClaim
- persistentVolume
- projected
- portworxVolume
- quobyte
- rbd
- scaleIO
- secret
- storageos
- vsphereVolume

49.4. Volúmen Local (PersistentVolume - PV)

PVs

Persistent Volumes allow you to create storage that can be accessed beyond the life of the pod



Un volumen local representa un elemento de almacenamiento del anfitrión, como un disco, una partición o un directorio

Los volúmenes locales sólo pueden ser utilizados de forma estática, mediante la unidad **PersistentVolume**, el aprovisionamiento dinámico todavía no está disponible

Es una pieza de almacenamiento en el clúster que ha sido aprovisionada por un administrador o aprovisionada dinámicamente mediante clases de almacenamiento

Se trata de un recurso del clúster, de igual forma que un nodo es un recurso del clúster

Los (PV) tienen un ciclo de vida independiente de cualquier Pod individual que use el PV

Este objeto de Kubernetes, captura los detalles de la implementación a bajo nivel del almacenamiento, ya sea un NFS, iSCSI o un sistema de almacenamiento específico del proveedor de la nube.

49.5. Volúmen Local (PersistentVolumeClaim - PVC)

Un **PersistentVolumeClaim** es una solicitud de almacenamiento por parte de un usuario

Es parecido a un Pod, los Pods consumen los recursos de un nodo y los (PVC) consumen recursos **PersistentVolume** (PV)

Los Pods pueden solicitar niveles específicos de recursos (CPU y memoria)

Las reclamaciones, llamadas en la jerga kubernetes **Claims** (unidades PersistentVolume (PV)) pueden solicitar tamaños específicos y modos de acceso, por ejemplo, se pueden montar una vez en modo lectura/escritura, o bien, muchas veces en modo sólo lectura



Capítulo 50. Lab: Volumen hostPath

Este volumen corresponde a un directorio o archivo del nodo lógico donde se crea el Pod, el clúster multinodo este tipo de volumen no es efectivo, ya que la información que se encuentre en los distintos nodos no estará duplicada y su contenido puede depender del nodo donde se crea el Pod

Si la carpeta que se especifica en el volumen no está presente en el nodo en el que se ejecuta el Pod, esta será creada automáticamente

Cuando se elimine el Pod que ha hecho uso o creación del volumen, este no se elimina, es persistente en disco

50.1. Creando 1 Pod con 1 container que usa un volumen

Vamos a crear el archivo **pod-with-1-container-volume-hostpath.yml** y agregamos el siguiente contenido

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-1-container-volume-hostpath
spec:
  containers:
    - name: nginx
      image: nginx:latest
      volumeMounts:
        - mountPath: /my-custom-path-inside-container
          name: volume-hostpath
  volumes:
    - name: volume-hostpath
      hostPath:
        path: /home/kubernetes
```

- Indicamos el nombre de la carpeta que aparecerá montada dentro del container **spec → containers → volumeMounts → mountPath**, esta carpeta tendrá el contenido como si de un enlace simbólico se tratara, de la carpeta que se defina en el volumen
- Hacemos referencia al nombre del volumen concreto que queremos que use el container **spec → containers → volumeMounts → name**
- Indicamos el nombre del volumen que queremos definir **spec → volumes → name**
- Indicamos el tipo de volumen que queremos utilizar **spec → volumes → hostPath**
- Indicamos el directorio lógico del nodo donde caiga el Pod, el contenido íntegro de esta carpeta aparecerá reflejado en el punto de montaje del Pod **spec → volumes → hostPath → path**

Ejecutamos la creación del Pod

```
$ kubectl apply -f pod-with-1-container-volume-hostpath.yml
```

```
pod/pod-with-1-container-volume-hostpath created
```

Ahora, nos conectamos dentro del container y vamos a comprobar si dentro del container se ha creado la ruta **/my-custom-path-inside-container**

Ejecutamos la conexión interna al container

```
$ kubectl exec -it pod-with-1-container-volume-hostpath bash
```

Estando conectados dentro del container, vamos a ejecutar un **ls -la** para comprobar que efectivamente tenemos dentro del container la ruta

```
root@pod-with-1-container-volume-hostpath:/# ls -la
total 8
drwxr-xr-x  1 root root  79 Oct  2 17:51 .
drwxr-xr-x  1 root root  79 Oct  2 17:51 ..
-rw xr-xr-x  1 root root  0 Oct 217:51 .dockerenv
drwxr-xr-x  2 root root 4096 Sep 10 00:00 bin
drwxr-xr-x  2 root root  6 Aug 30 12:31 boot
drwxr-xr-x  5 root root 360 Oct  2 17:51 dev
drwxr-xr-x  1 root root  66 Oct 217:51 etc
drwxr-xr-x  2 root root  6 Aug 30 12:31 home
drwxr-xr-x  1 root root  56 Sep 24 23:33 lib
drwxr-xr-x  2 root root 34 Sep 10 00:00 lib64
drwxr-xr-x  2 root root  6 Sep 10 00:00 media
drwxr-xr-x  2 root root  6 Sep 10 00:00 mnt
drwxr-xr-x  2 root root  6 Oct 2 17:51 my-custom-path-inside-container
drwxr-xr-x  2 root root  6 Sep 10 00:00 opt
dr-xr-xr-x 139 root root  0 Oct 2 17:51 proc
drwx----- 1 root root 27 Oct 2 17:55 root
drwxr-xr-x  1 root root 38 Oct 2 17:51 run
drwxr-xr-x  2 root root 4096 Sep 10 00:00 sbin
drwxr-xr-x  2 root root  6 Sep 10 00:00 srv
dr-xr-xr-x 13 root root  0 Sep 25 20:32 sys
drwxrwxrwt  1 root root  6 Sep 24 23:33 tmp
drwxr-xr-x  1 root root  66 Sep 10 00:00 usr
drwxr-xr-x  1 root root 19 Sep 10 00:00 var
root@pod-with-1-container-volume-hostpath:/#
```

Listamos el contenido dentro de la carpeta de **my-custom-path-inside-container**

```
root@pod-with-1-container-volume-hostpath:/my-custom-path-inside-container# ls -la
total 16
drwx----. 2 1001 1001 83 Oct 2 18:02 .
drwxr-xr-x 1 root root 79 Oct 2 18:02 ..
-rw-----. 1 1001 1001 1298 Sep 24 15:00 .bash_history
-rw-r--r--. 1 1001 1001 18 Oct 30 2018 .bash_logout
-rw-r--r--. 1 1001 1001 193 Oct 30 2018 .bash_profile
-rw-r--r--. 1 1001 1001 231 Oct 30 2018 .bashrc
root@pod-with-1-container-volume-hostpath:/my-custom-path-inside-container#
```

- Observamos el contenido interno del directorio /home/kubernetes del nodo que esté ejecutando el Pod

50.2. Creando 1 Pod con 2 container que usan un volumen compartido

Vamos a crear el archivo **pod-with-2-container-volume-shared-hostpath.yml** y agregamos el siguiente contenido

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-2-container-volume-shared-hostpath
spec:
  volumes:
  - name: shared-data
    hostPath:
      path: /home/kubernetes/shared-folder
  containers:
  - name: nginx-container
    image: nginx
    volumeMounts:
    - name: shared-data
      mountPath: /data-in-nginx-container
  - name: tomcat-container
    image: tomcat:alpine
    volumeMounts:
    - name: shared-data
      mountPath: /data-in-tomcat-container
```

Ejecutamos la creación del Pod

```
$ kubectl apply -f pod-with-2-container-volume-shared-hostpath.yml
pod/pod-with-2-container-volume-shared-hostpath created
```

Abrimos 3 consolas y nos conectamos

- Internamente al contenedor del nginx, nos situamos en su carpeta **/data-in-nginx-container**

```
$ kubectl exec -it pod-with-2-container-volume-shared-hostpath -c nginx-container bash  
total 0  
drwxr-xr-x 2 root root 6 Oct 2 18:22 .  
drwxr-xr-x 1 root root 71 Oct 2 18:22 ..  
root@pod-with-2-container-volume-shared-hostpath:/data-in-nginx-container# pwd  
/data-in-nginx-container  
root@pod-with-2-container-volume-shared-hostpath:/data-in-nginx-container#
```

- Internamente al contenedor del tomcat, nos situamos en su carpeta **/data-in-tomcat-container**

```
$ kubectl exec -it pod-with-2-container-volume-shared-hostpath -c tomcat-container bash  
/data-in-tomcat-container  
bash-4.4# ls -la  
total 0  
drwxr-xr-x 2 root root 6 Oct 2 18:22 .  
drwxr-xr-x 1 root root 71 Oct 2 18:22 ..  
bash-4.4#
```

- Directamente en el nodo en el que haya caído el Pod y vamos a la carpeta **/home/kubernetes/shared-data**

A continuación, escribimos 1 archivo desde cada sitio, 1 desde el contenedor nginx, otro desde el contenedor tomcat y otro desde el propio host donde esté el nodo computando

Deberíamos de observar que los archivos son visibles desde todos los puntos

Capítulo 51. Lab: Volumen emptyDir

Este tipo de volumen, se crea por primera vez cuando se asigna un Pod a un Nodo

El volumen existe mientras ese Pod se esté ejecutando en el nodo en cuestión

El volumen inicialmente está vacío

Todos los contenedores en el Pod pueden leer y escribir los mismos archivos en el volumen emptyDir, tengan montadas las mismas o diferentes rutas en el contenedor

Cuando un Pod se elimina de un nodo por cualquier motivo, los datos de emptyDir son eliminados de forma automática

Si el Pod se está ejecutando, y algún contenedor interno tiene algún problema, se observa por ejemplo un estado "CrashLoop" o por el estilo, los datos del volumen emptyDir están seguros, ya que el hecho de que algún contenedor tenga un problema, no afecta al Pod

De forma predeterminada, los volúmenes emptyDir se almacenan en cualquier medio que esté respaldando el nodo, que podría ser un disco o SSD o almacenamiento de red, dependiendo de su entorno.

Sin embargo, se puede configurar el campo **emptyDir.medium** en "Memoria" para decirle a Kubernetes que monte un tmpfs (sistema de archivos respaldado por RAM) en lugar de en disco.

Si bien tmpfs es muy rápido, tenemos que tener en cuenta que, a diferencia de los discos, tmpfs se borra al reiniciar el nodo y cualquier archivo que escribamos contará con un límite de memoria de su Contenedor.

51.1. Posibles usos de este tipo de volumen

- Espacio de disco temporal para operaciones temporales de procesamiento
- Como posible punto de referencia de computación, para software que lleva a cabo procesamiento de archivos (en caso de que un contenedor que está llevando a cabo operaciones de computación falle)
- Contener archivos que un contenedor de administrador de contenido recupera mientras un contenedor de servidor web sirve los datos

51.2. Creando un 1 Pod con 2 containers que usan un volumen compartido temporal

Vamos a crear el archivo **pod-with-2-container-volume-emptydir.yml** y agregamos el siguiente contenido

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-2-container-volume-emptydir
spec:
  volumes:
    - name: shared-empty-dir-data
      emptyDir: {}
  containers:
    - name: nginx-container
      image: nginx:latest
      volumeMounts:
        - name: shared-empty-dir-data
          mountPath: /var/www/html
    - name: tomcat-container
      image: tomcat:alpine
      volumeMounts:
        - name: shared-empty-dir-data
          mountPath: /data-in-tomcat-container
```

Ejecutamos la creación del Pod

```
$ kubectl apply -f pod-with-2-container-volume-emptydir.yml
pod/pod-with-2-container-volume-emptydir created
```

Si ahora le realizamos un describe al Pod, observaremos que nos indica que efectivamente se usará un directorio temporal compartido mientras el Pod no sea destruido



Fondos Europeos

MINISTERIO
DE EDUCACIÓN, FORMACIÓN PROFESIONAL
Y DEPORTE

Cofinanciado por
la Unión Europea



Capítulo 52. PersistentVolume

Un volumen local representa un elemento de almacenamiento del anfitrión, como un disco, una partición o un directorio

Los volúmenes locales sólo pueden ser utilizados de forma estática, mediante la unidad **PersistentVolume**, el aprovisionamiento dinámico todavía no está disponible

Es una pieza de almacenamiento en el clúster que ha sido aprovisionada por un administrador o aprovisionada dinámicamente mediante clases de almacenamiento

Estas unidades de persistencia son recursos del clúster, de igual forma que un nodo es un recurso del clúster

Los (PV) tienen un ciclo de vida independiente de cualquier Pod individual que use el PV

Este objeto de Kubernetes, captura los detalles de la implementación a bajo nivel del almacenamiento, ya sea un NFS, iSCSI o un sistema de almacenamiento específico del proveedor de la nube.

Se pueden utilizar de manera portable, el sistema es consciente de las restricciones de nodos del volumen, al observar afinidad de nodos en la unidad **PersistentVolume**

Los volúmenes locales están sujetos a que el nodo en cuestión donde van a situarse, esté disponible

Si un nodo que tiene un volumen local se vuelve inestable y no puede ser accedido, el volumen local también será inaccesible y un Pod que deba de usarlo no podrá ejecutarse, por que el volumen en cuestión no estará disponible

Las aplicaciones que utilicen volúmenes locales deben de poder tolerar esta posible pérdida de disponibilidad

El requisito llevar a cabo copias de seguridad periódicas de los volúmenes locales



52.1. Aprovisionando estático vs dinámico

52.1.1. Estático

El administrador del clúster crea un número de unidades PV's, llevan los detalles del almacenamiento real que está disponible para que lo usen los usuarios del clúster

Están disponibles para su consumo

52.1.2. Dinámico

Cuando ninguno de los PV estáticos que creó el administrador coincide con el PersistentVolumeClaim de un usuario, el clúster puede intentar aprovisionar dinámicamente un volumen especialmente para una unidad de **PersistentVolumeClaim (PVC)**

Este tipo de aprovisionamiento se basa en **StorageClass**, funciona de forma que el administrador

debe haber creado y configurado esa clase para que se produzca el aprovisionamiento dinámico

Para habilitar el aprovisionamiento de almacenamiento dinámico basado en la clase del almacenamiento, debe habilitar el controlador de admisión **DefaultStorageClass**

Esto se puede hacer por ejemplo, asegurando que **DefaultStorageClass** se encuentre entre la lista ordenada de valores delimitada por comas para el indicador **--enable-admission-plugins**

52.2. Capacity

Como regla general, una unidad PV tiene una capacidad de almacenamiento específico

Esto es asignado a la unidad mediante el token **capacity**

Actualmente el tamaño del almacenamiento es el único recurso que se puede configurar o solicitar

52.3. VolumeMode



El funcionamiento de este tag se encuentra en versión Beta desde la v1.13

Podemos indicar en modo de actuación del volumen

- **block**

- La persistencia utilizará bloques de almacenamiento en el dispositivo en cuestión en raw

- **filesystem**

- Es el valor por defecto
 - Utilizará el sistema de archivos tal cual

52.4. Access Modes

Una unidad **PersistentVolume** (PV) puede ser montado en un host de cualquier forma admitida por el proveedor de recursos

Access Modes

By specifying an access mode with your PersistentVolume (PV), you allow the volume to be mounted to one or many nodes, as well as read by one or many nodes.

spec:

```
capacity:  
  storage: 1Gi  
accessModes:  
  - ReadWriteOnce  
  - ReadOnlyMany
```

- ▶ **RWO (ReadWriteOnce)** – Only one node can mount the volume for writing and reading.
- ▶ **ROX (ReadOnlyMany)** – Multiple nodes can mount the volume for reading.
- ▶ **RWX (ReadWriteMany)** – Multiple nodes can mount the volume for writing and reading.

NAME	CAPACITY	ACCESSMODES	RECLAIMPOLICY	STATUS
mongodb-pv	1Gi	RWO, RWX	Retain	Available

Mediante esta característica del PV, se especifica el acceso del nodo al volumen persistente, puede ser de 3 tipos:

- **ReadWriteOnce**

- El volumen puede ser montado como lectura y escritura en un sólo nodo

- **ReadOnlyMany**

- El volumen puede ser montado para operaciones de sólo lectura en múltiples nodos

- **ReadWriteMany**

- El volumen puede ser montado para operaciones de escritura/lectura en varios nodos

Un volumen sólo se puede montar usando un modo de acceso a la vez

Por ejemplo, utilizando el plugin GCEPersistentDisk se puede montar como ReadWriteOnce por un solo nodo o ReadOnlyMany por muchos nodos, pero no al mismo tiempo.

A pesar de esto, podemos indicar una lista de modos de acceso, pero como se ha comentado, únicamente será posible utilizar 1 de los configurados a la vez

Otro punto muy importante a remarcar, es que la capacidad de montaje de la unidad PV es del nodo, no del Pod que posteriormente utilice el volumen

52.5. StorageClassName

Una unidad **PersistentVolume** (PV) puede tener una clase, que está especificada por el token **storageClassName**

Una unidad **PersistentVolume** (PV) de una clase particular sólo puede vincularse a una unidad

PersistentVolumeClaim (PVC) que solicitan ese storageClass en concreto

Una unidad **PersistentVolume** (PV) sin storageClassName no tiene clase y sólo podrá vincularse a una unidad **PersistentVolumeClaim** (PVC) que no soliciten ninguna clase en particular

52.6. PersistentVolumeReclaimPolicy

Actualmente las políticas de reciclaje de volúmenes disponibles son las siguientes

- **Retain**

- Reclamación manual

- **Recycle**

- Al eliminar la unidad, se realiza un borrado recursivo de la carpeta que estuviera haciendo uso el volumen

- `rm -rf /thevolume/*`

- **Delete**

- Al eliminar la unidad de persistencia, se elimina el almacenamiento asociado como AWS, EBS, GCE PD, Azure Disk o el volumen de OpenStack Cinder

Actualmente sólamente los plugins de volúmenes NFS and HostPath soportan el valor del parámetro recycle.



AWS EBS, GCE PD, Azure Disk, y los volúmenes Cinder de OpenStack soportan la eliminación.

52.7. MountOptions

El administrador de Kubernetes puede especificar información adicional de montaje cuando este es montado en un nodo



No todos los plugins de persistencia de volúmenes soportan opciones de montaje

Los siguientes plugins de volúmenes soportan opciones de montaje

- AWSElasticBlockStore
- AzureDisk
- AzureFile
- CephFS
- Cinder (OpenStack block storage)
- GCEPersistentDisk
- Glusterfs
- NFS

- Quobyte Volumes
- RBD (Ceph Block Device)
- StorageOS
- VsphereVolume
- iSCSI



Las opciones de montaje, el Kube API Server no le realiza ningún tipo de parseo para comprobar si están bien o mal, el montaje simplemente fallará si alguna opción de montaje no es válida

52.8. NodeAffinity

Una unidad PV puede especificar afinidad de nodos para definir restricciones que limitan desde qué nodos se puede acceder a este volumen

Los Pods que usan un PV sólo se programarán en nodos seleccionados por la afinidad de nodos



Para la mayoría de los tipos de volumen no necesita establecer este campo

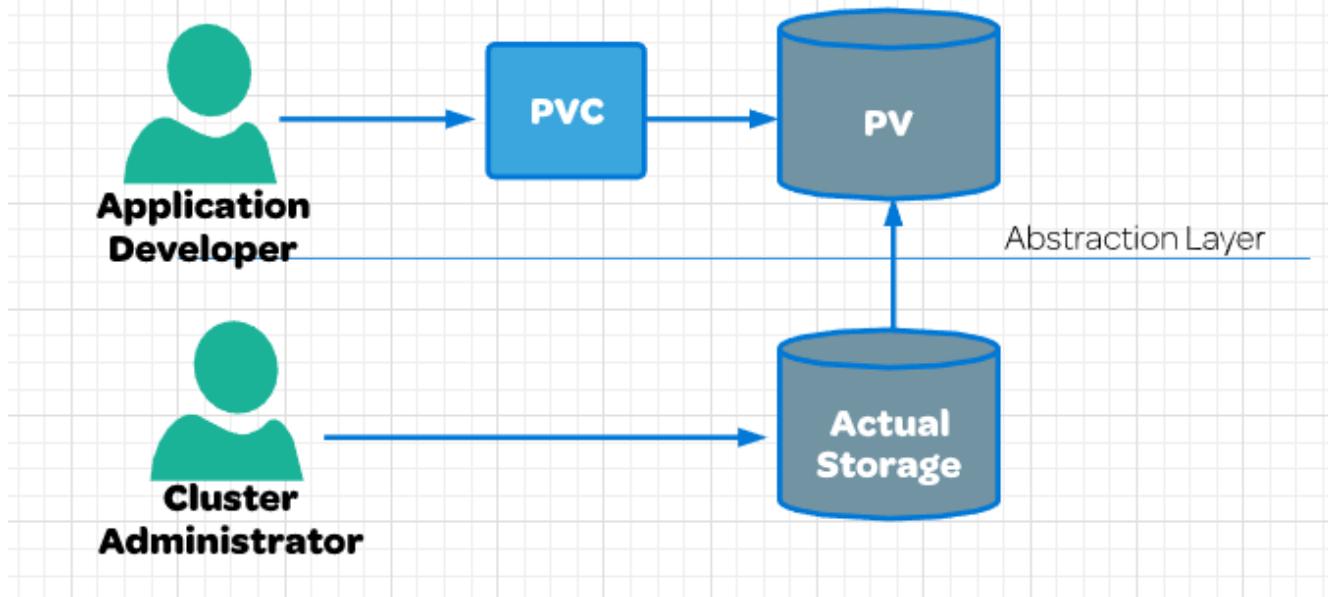
Se rellena automáticamente para los tipos de bloque de volumen AWS EBS, GCE PD y Azure Disk volume block.

Sin embargo, para volúmenes locales debemos de especificar esto de forma explícita

Capítulo 53. PersistentVolumeClaim

PV Claims

PersistentVolumeClaims (PVCs) allow the application developer to request storage for the application, without having to know about the underlying infrastructure.



Un **PersistentVolumeClaim** es una solicitud de almacenamiento por parte de un pod

Si un Pod necesita un volumen, debemos de realizar una solicitud de almacenamiento usando un elemento de tipo **PersistentVolumeClaim**

Es parecido a un Pod, los Pods consumen los recursos de un nodo y los (PVC) consumen recursos **PersistentVolume** (PV)

El elemento **PersistentVolumeClaim** es una entidad aparte del Pod, aunque el Pod haga uso de la misma

Los Pods pueden solicitar niveles específicos de recursos (CPU y memoria)

Las reclamaciones, llamadas en la jerga kubernetes **Claims** (unidades PersistentVolumeClaim (PVC)) pueden solicitar tamaños específicos y modos de acceso, por ejemplo, se pueden montar una vez en modo lectura/escritura, o bien, muchas veces en modo sólo lectura

Cuando se crea un **PersistentVolumeClaim**, se le asignará un **PersistentVolumen** que se ajuste a la petición del "claim", esta asignación se puede realizar de dos formas distintas



- **Estática**

- Primero se crea todos los PersistentVolumenClaims por parte del administrador, que se irán asignando conforme se vayan creando los PersistentVolumen.

- **Dinámica**

- En este caso necesitamos un "provisionador" de almacenamiento (para cada uno de los backend), de tal manera que cada vez que se cree un PersistentVolumenClaim, se creará

bajo demanda un PersistentVolumen que se ajuste a las características seleccionadas.



Capítulo 54. Lab: PersistentVolume

Mediante este laboratorio de PersistentVolume, implementaremos algunas de las operaciones más comunes de esta unidad de Kubernetes

54.1. Creando una unidad de persistencia de 5Gb y acceso R/W

Vamos a crear el archivo **persistent-volume-5gb.yml** y agregamos el siguiente contenido

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: volume-5gb
spec:
  capacity:
    storage: 5Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteMany
  persistentVolumeReclaimPolicy: Recycle
  storageClassName: local-storage
  local:
    path: /home/kubernetes/disk1
  nodeAffinity:
    required:
      nodeSelectorTerms:
        - matchExpressions:
          - key: kubernetes.io/hostname
            operator: In
            values:
              - kubeminion1
```

- Indicamos una afinidad de nodo, cuya etiqueta kubernetes.io/hostname tenga el valor de kubeminion1
- En el nodo kubeminion1 tiene que haber creada una carpeta en la ruta **/home/kubernetes/disk1**
- Indicamos como política del volumen **Recycle**, si queremos que ante eliminaciones de los Pods y de las PVC (Persistent Volume Claims), nada se pierda, deberemos de ajustar la política a **Retain**

Ejecutamos la creación de la unidad de persistencia

```
$ kubectl apply -f persistent-volume-5gb.yml
```

54.2. Listando las unidades de persistencia

Vamos a listar las unidades de persistencia que estén presentes en el clúster, de forma detallada

```
$ kubectl get pv -o wide
```

NAME	CAPACITY	ACCESS MODES	RECLAIM POLICY	STATUS	CLAIM
STORAGECLASS	REASON	AGE	VOLUMEMODE		
volume-5gb	5Gi	RWX	Recycle	Available	local-
storage		28s	Filesystem		



Capítulo 55. Lab: PersistentVolumeClaim

Mediante este laboratorio, vamos a llevar a cabo algunas de las operaciones más comunes con esta unidad de kubernetes

55.1. Creando PersistentVolumeClaim que usa PersistentVolume

Anteriormente hemos creado la unidad (PV) **volume-5gb**, ahora vamos crear una unidad (PVC) que haga uso del volumen

Vamos a crear el archivo **persistent-volume-claim.yml** con el siguiente contenido

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: volume-pvc
spec:
  accessModes:
    - ReadWriteMany
  storageClassName: local-storage
  resources:
    requests:
      storage: 1Gi
```

Ejecutamos la creación de la unidad

```
$ kubectl apply -f persistent-volume-claim.yml
persistentvolumeclaim/volume-pvc created
```

55.2. Listando las claims (PVC)

Procedemos a listar las unidades PVC que se encuentren en el sistema

```
$ kubectl get pvc
NAME      STATUS  VOLUME      CAPACITY  ACCESS MODES  STORAGECLASS  AGE
volume-pvc  Bound  volume-5gb  5Gi       RWX        local-storage  5s
```

- Al crear el pvc se busca un conjunto de pv, uno que cumpla los requerimientos de la unidad PVC
- Automáticamente si se encuentra, quedan asociados (status bound)
- El tamaño indicado en el pvc es el valor mínimo de tamaño que se necesita, pero el tamaño real será el mismo que el del pv asociado.

55.3. Añadiendo un volumen a un Pod a partir de un claim

Ahora, vamos a poner en marcha un Pod que haga uso de esa solicitud de claim

Creamos el archivo **pod-using-claim.yml** con el siguiente contenido

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-using-volume-claim
spec:
  containers:
    - name: nginx
      image: nginx:latest
      volumeMounts:
        - mountPath: /usr/share/nginx/html
          name: volumeclaim
  volumes:
    - name: volumeclaim
      persistentVolumeClaim:
        claimName: volume-pvc
```

Consultamos la IP del Pod para acceder al puerto 80

```
$ kubectl get pods -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
NOMINATED NODE		READINESS	GATES			
pod-using-volume-claim	1/1	Running	0	17s	10.36.0.1	kubeminion1
<none>						

Si ponemos en el navegador <http://10.36.0.1:80> observaremos que el propio nginx genera un error, y no muestra la página de bienvenida

Esto es debido, a que hemos redireccionado el volumen donde el servidor web nginx espera encontrar archivos, a un claim que hace uso de un volumen que está montando la carpeta **/home/kubernetes/disk1**, y esa carpeta está ahora mismo vacía

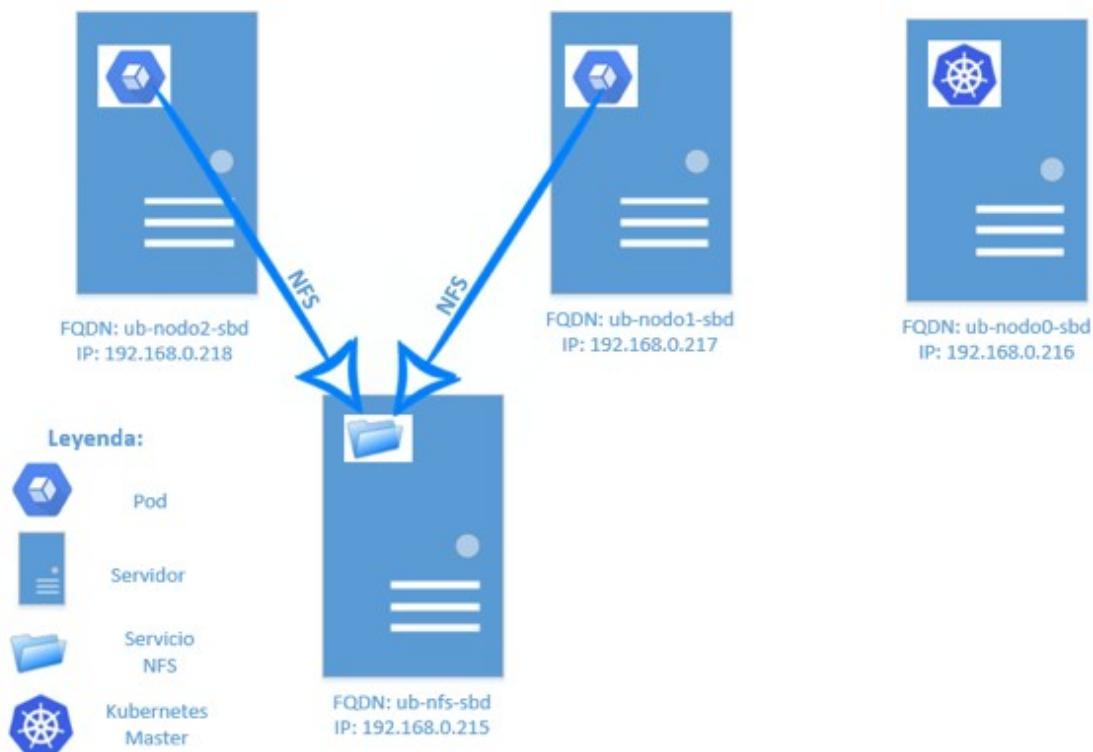
Accedemos al nodo kubeminion1 y creamos dentro de esa carpeta un archivo index.html con algún contenido, de forma que automáticamente el index.html se verá reflejado en el navegador web

Capítulo 56. Volúmenes mediante NFS

Para que los datos no se eliminen de un pod cuando este sea borrado, es necesario el empleo de los volúmenes

Para llevar a cabo este objetivo, montaremos un servidor NFS y posteriormente desplegaremos los volúmenes, para poder acceder al servicio de NFS desde nuestros pods

Nuestro objetivo es crear un esquema de este tipo:



Lo primero que vamos a hacer, es preparar nuestro servidor de "storage", posteriormente, compartiremos la carpeta del servidor para que los pods puedan montar dicha carpeta

Llevamos a cabo la instalación del servidor NFS en dicha máquina

```
# yum -y install nfs-utils
```

Habilitamos el servicio para que esté activado y lo arrancamos

```
# systemctl enable nfs-server.service  
# systemctl start nfs-server.service
```

Creamos una carpeta donde queramos, será donde NFS almacenará datos, por ejemplo, dentro del home, una que se llame nfs

```
# mkdir /home/nfs
```

Añadimos el siguiente contenido

```
/home/nfs *(rw,no_root_squash)
```

Por último, reiniciamos el servicio

```
# systemctl restart nfs
```

Instalamos también NFS en cada minion que actúa como worker de pods

```
# yum -y install nfs-utils
```

Verificamos desde cada minion, que efectivamente hay conectividad con el servidor NFS

```
# showmount -e 10.0.2.8
```

```
Export list for 10.0.2.8
```

```
/home/nfs *
```

Procedemos al montaje de la unidad en red, en cada uno de los minions

```
# mount -t nfs 10.0.2.8:/home/nfs /mnt/
```

Verificamos que el montaje se produce de forma correcta en cada uno de los minions

```
# df -h | grep nfs
```

```
10.0.2.8:/home/nfs 46G 33M 46G 1% /mnt
```

Desmontamos la unidad en cada uno de los minions

```
# umount /mnt
```

A continuación, creamos el .yml de definición del volumen de persistencia

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: kube-nfs-pv
spec:
  storageClassName: storage-nfs
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteMany
  nfs:
    server: 10.0.2.8
    path: "/home/nfs"
```

También, el archivo del volume claim

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: kube-nfs-pvc
spec:
  storageClassName: storage-nfs
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 10Gi
```

Por último, creamos un Replication Controller para probar el despliegue del volumen compartido, utilizamos nginx como web werver

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: nfs-web
spec:
  replicas: 2
  selector:
    role: nfs-web-rc
  template:
    metadata:
      labels:
        role: nfs-web-rc
    spec:
      containers:
        - name: web
          image: nginx
          ports:
            - name: web
              containerPort: 80
      volumeMounts:
        - name: kube-nfs-pvc
          mountPath: "/var/www/html"
  volumes:
    - name: kube-nfs-pvc
      persistentVolumeClaim:
        claimName: kube-nfs-pvc
```



Fondos Europeos



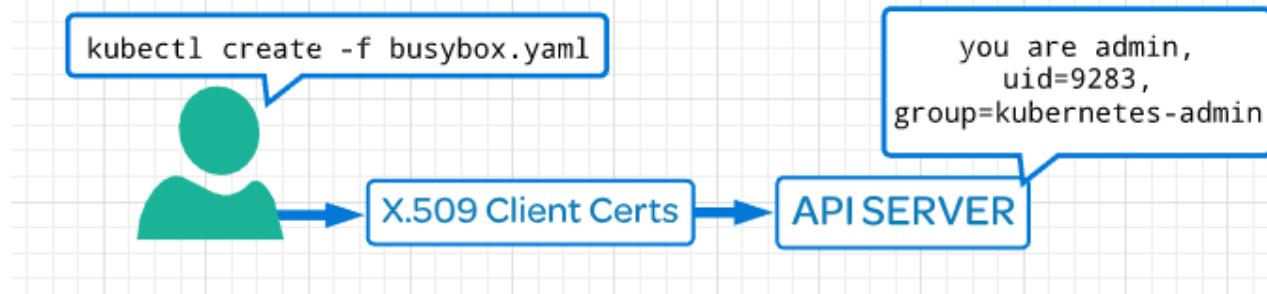
Cofinanciado por
la Unión Europea



Capítulo 57. ServiceAccount

Service Accounts and Users

The API server first evaluates if the request is coming from a service account or a normal user. A normal user may be a private key, a user store, or even a file with a list of usernames and passwords.



El componente kube-api-server es el primero que evalúa la petición de entrada a kubernetes, para determinar si se está realizando mediante una cuenta normal de usuario, o bien, mediante una cuenta de servicio

- Cuenta de usuario normal

Cuando hablamos de una cuenta de usuario normal, estamos hablando de un conjunto de autenticación formado por:

- Una clave privada
- Un almacén de usuario
- Incluso puede tratarse de un archivo con una lista de nombres de usuario y contraseñas

Kubernetes no tiene objetos que representan cuentas de usuario normales, y los usuarios normales no se pueden agregar al clúster a través de una llamada a la API

- Cuenta de servicio

Las cuentas de servicio, es lo que kubernetes utiliza para gestionar la identidad de las solicitudes que llegan a kube-api-server desde los Pods, o mediante un cliente como kubectl que pretende gestionar el clúster

Capítulo 58. Lab: Cuentas de Servicio

58.1. Consultando las cuentas de servicio por defecto en Kubernetes

Vamos a consultar las cuentas de servicio que por defecto trae kubernetes.

Ejecutamos el siguiente comando:

```
$ kubectl get serviceaccounts
```

NAME	SECRETS	AGE
default	1	24d



- Observamos la cuenta de servicio por defecto que kubernetes crea cuando el clúster se instala

58.2. Creando nuestra propia cuenta de servicio

Vamos a crear una cuenta de servicio para nuestro frontend del sistema de facturación.

```
$ kubectl create serviceaccount frontend-app-bill
```

```
serviceaccount/frontend-app-bill created
```



- Automáticamente se crea la cuenta de servicio
- También de forma conjunta se crea un secret para dicha cuenta de servicio

Ahora, listamos de nuevo las cuentas de servicio y debería de aparecer la nuestra:

```
$ kubectl get serviceaccounts
```

NAME	SECRETS	AGE
default	1	24d
frontend-app-bill	1	2m41s



- Observamos que ambas cuenta de servicio, tienen 1 secret asociado

El secreto asociado, contiene la CA (Public Certificate Authority) del kube-api-server así como un JSON web token asociado (JWT)

58.3. Obteniendo la cuenta de servicio en formato YAML

Ahora, vamos a obtener la descripción de la cuenta de servicio, en formato YAML.

Ejecutamos en consola el siguiente comando:

```
$ kubectl get serviceaccount frontend-app-bill -o yaml
```

```
apiVersion: v1
kind: ServiceAccount
metadata:
  creationTimestamp: "2020-01-09T17:19:58Z"
  name: frontend-app-bill
  namespace: default
  resourceVersion: "500317"
  selfLink: /api/v1/namespaces/default/serviceaccounts/frontend-app-bill
  uid: df76b616-1b4e-47f3-ab33-c2d8793cc84e
secrets:
- name: frontend-app-bill-token-pt5br
```



- Observamos el nombre del secret que se ha creado de forma paralela

Listamos el secret que nos ha informado el ServiceAccount:

```
$ kubectl get secret frontend-app-bill-token-pt5br
```

NAME	TYPE	DATA	AGE
frontend-app-bill-token-pt5br	kubernetes.io/service-account-token	3	9m49s

- Este token que observamos, será lo que usará la solicitud para autenticarse con el API server del servidor
- Cuando desplegamos un Pod, podemos asignar una cuenta de servicio específica a un Pod en su archivo de manifiesto .yml
- Si no incluimos ninguna cuenta de servicio en el manifiesto del Pod, se usuará la cuenta de servicio por defecto, llamada **default**

58.4. Especificando la cuenta de servicio de nuestro Pod

Vamos a especificar una cuenta de servicio específica a nuestro Pod, vamos a crear un archivo con nombre **pod-with-custom-service-account.yml** y le agregamos el siguiente contenido:

```

apiVersion: v1
kind: Pod
metadata:
  name: pod-with-custom-service-account
spec:
  serviceAccountName: frontend-app-bill
  containers:
    - name: nginx
      image: nginx:latest
      ports:
        - containerPort: 80

```

Ejecutamos la creación del Pod:

```
$ kubectl apply -f pod-with-custom-service-account.yml
pod/pod-with-custom-service-account created
```

Obtenemos el Pod en formato YAML, para comprobar la cuenta de servicio que está utilizando:

```
$ kubectl get pod pod-with-custom-service-account -o yaml
...
serviceAccount: frontend-app-bill
...
```

- Deberíamos de observar en el conjunto de elementos, una key llamada serviceAccount con el valor que hemos indicado previamente
- Todos los Pods que compartan la misma cuenta de servicio podrán gestionarse entre sí



Por ejemplo, si tuviéramos un sistema de integración continua como Jenkins y le diéramos posibilidad de usar el cliente de acceso a kubernetes kubectl, introduciendo como parámetro el token, jenkins podría controlar pods que operaran bajo esa cuenta de servicio

58.5. Comprobación de credenciales kubectl

Cuando utilizamos la herramienta kubectl necesitamos conocer la ubicación del clúster y tener credenciales para acceder al mismo

Vamos a comprobar la configuración de credenciales, ejecutando el siguiente comando:

```
$ kubectl config view

apiVersion: v1
clusters:
- cluster:
  certificate-authority-data: DATA+OMITTED
  server: https://192.168.15.100:6443
  name: kubernetes
contexts:
- context:
  cluster: kubernetes
  user: kubernetes-admin
  name: kubernetes-admin@kubernetes
current-context: kubernetes-admin@kubernetes
kind: Config
preferences: {}
users:
- name: kubernetes-admin
  user:
    client-certificate-data: REDACTED
    client-key-data: REDACTED
```

- Observamos el endpoint al cual está apuntando la configuración de conexión de nuestro cliente kubectl
 - server: <https://192.168.15.100:6443>
- También observamos el contexto, donde aparece el nombre del clúster, el usuario con el que conectamos, y el elemento name, que sería la credencial completa para conectar con el clúster

Este archivo de configuración, podemos encontrarlo en la siguiente ruta:

```
$USER_HOME/.kube/config
```

- Si visualizamos el archivo con un cat, aparecerá algo más de información que el comando config view ha acortado

En cada clúster tienen el mismo nombre:

- El clúster en sí
 - **contexts.context.cluster**
- El usuario
 - **contexts.context.user**
- El contexto completo
 - **contexts.current-context**

Siempre usamos un nombre para referirnos a un contexto, usuario o clúster

58.6. Creando otro usuario para acceder al clúster

Si quiero acceder al clúster desde otro servidor diferente, debería de suministrar los siguientes datos:

- La ubicación del clúster
- El usuario
- El contexto

Supongamos que queremos crear un usuario llamado pakito, para acceder al clúster de forma remota, ya sea desde el propio servidor maestro o desde otro servidor diferente

En el servidor maestro, tendríamos que establecer las credenciales oportunas, podemos llevarlo a cabo mediante la herramienta kubectl

Tendremos que también crear de forma paralela un nuevo ClusterRoleBinding para esa nueva cuenta de usuario

Ejecutamos en la consola la creación del usuario:

```
$ kubectl config set-credentials pakito --username=pakito --password=my-secret-key  
User "pakito" set.
```

- Como identificador de cuenta **pakito**
- Como nombre de usuario **--username=pakito**
- Como clave de usuario **--password=my-secret-key**

Ahora, tenemos que crear un **ClusterRoleBinding**.

Vamos a crear uno para conectar usuarios, ejecutando el siguiente comando:

```
$ kubectl create clusterrolebinding cluster-system-anonymous --clusterrole=cluster-admin --user=system:anonymous  
clusterrolebinding.rbac.authorization.k8s.io/cluster-system-anonymous created
```

El siguiente paso, sería copiar los archivos de los certificados al nodo **kubeminion2**, para poder controlar el clúster desde ese nodo:

Accedemos al directorio de certificados de kubernetes en el nodo maestro y copiamos el archivo **ca.crt** al home del usuario kubernetes en el servidor **kubeminion2**:

```
[kubernetes@kubemaster ~]$ scp /etc/kubernetes/pki/ca.crt kubernetes@kubeminion2:~/
```

58.7. Configuraciones en el nodo kubeminion2

Estando conectados al nodo kubeminion2, mediante la herramienta kubectl, indicamos a la herramienta, la parametrización de contexto del clúster al que nos queremos conectar:

```
[kubernetes@kubeminion2 ~]$ kubectl config set-cluster kubernetes --server=https://192.168.15.100:6443 --certificate-authority=ca.crt --embed-certs=true
```

```
Cluster "kubernetes" set.
```

- Indicamos como nombre de clúster
 - **kubernetes**
- Indicamos la URL endpoint a la cual conectar
 - **--server=https://192.168.15.100:6443**
- Indicamos el archivo del certificado de la entidad certificadora
 - **--certificate-authority=ca.crt**
- Indicamos embeber los certificados en el mismo archivo config
 - **--embed-certs=true**

Ahora, toca añadir en el nodo, las credenciales del usuario pakito, ejecutando el siguiente comando:

```
[kubernetes@kubeminion2 ~]$ kubectl config set-credentials pakito --username=pakito --password=my-secret-key
```

```
User "pakito" set.
```

Por último, nos quedaría indicarle al usuario pakito que acabamos de crear, el contexto de uso:

```
[kubernetes@kubeminion2 ~]$ kubectl config set-context kubernetes --cluster=kubernetes --user=pakito --namespace=default
```

```
Context "kubernetes" created.
```

Indicamos en la consola, el contexto que queremos usar por el intérprete kubectl:

```
[kubernetes@kubeminion2 ~]$ kubectl config use-context kubernetes
```

```
Switched to context "kubernetes".
```

Ahora, probamos desde el nodo kubeminion2, listar los nodos del clúster, para comprobar que ahora podemos hablar con el kube-api-server:

```
[kubernetes@kubeminion2 ~]$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
kubemaster	Ready	master	24d	v1.17.0
kubeminion1	Ready	minion	24d	v1.17.0
kubeminion2	Ready	<none>	3d23h	v1.17.0



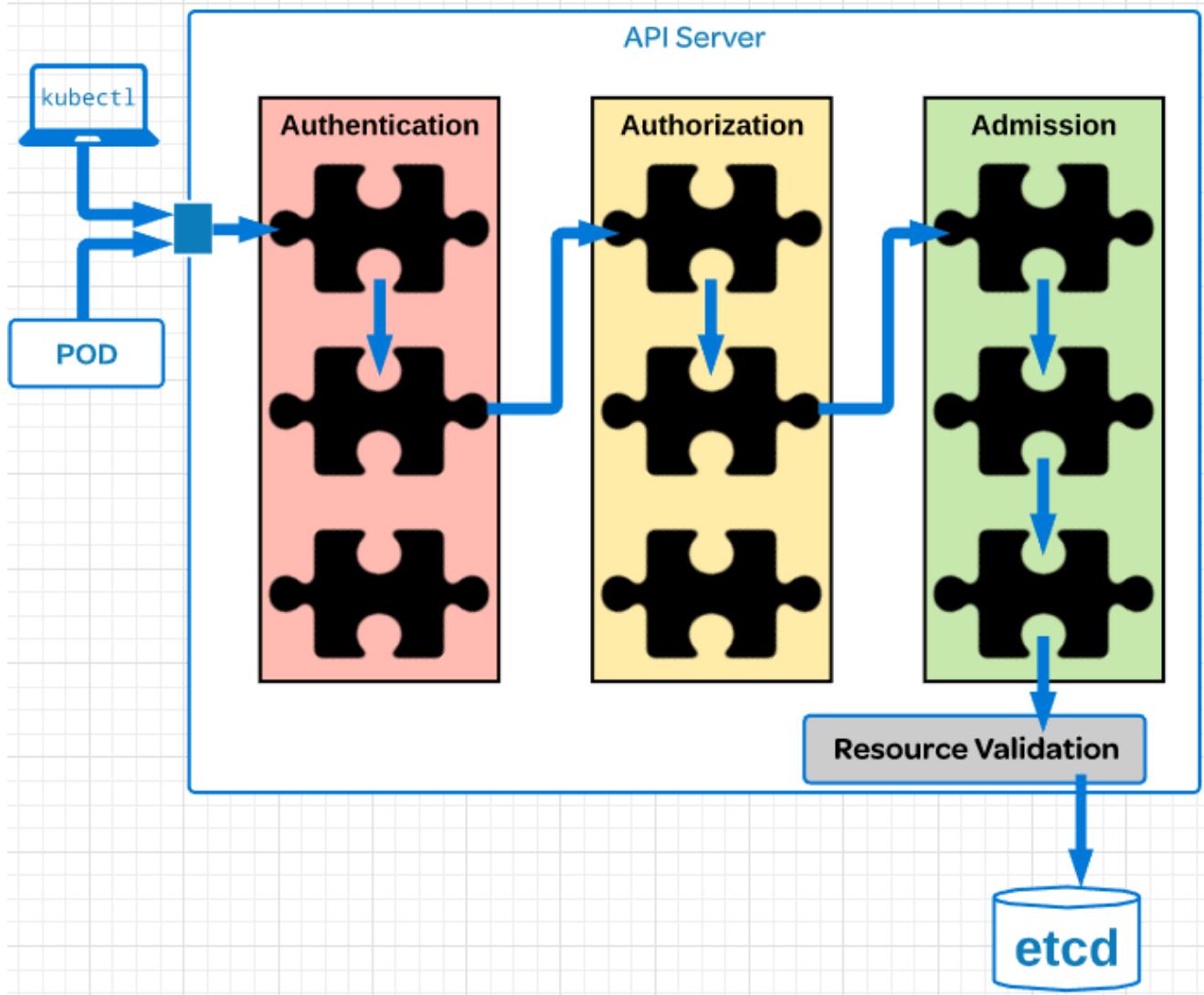
Capítulo 59. Procedimiento en las comunicaciones seguras

Para prevenir accesos no autorizados que alteren el estado del clúster, kubernetes utiliza un mecanismo de autorización denominado RBAC (Role-based access control)

RBAC es un método para regular el acceso a los recursos informáticos o de red en función de los roles de los usuarios individuales dentro de una empresa.

Securing Kubernetes API Access

The Kubernetes API server provides a CRUD (Create, Read, Update, Delete) interface for querying and modifying the cluster state over a RESTful API.



Al utilizar la herramienta de línea de comandos **kubectl**, estamos utilizando un mero traductor hacia el componente **kube-api-server**

El componente **kube-api-server** provee una interfaz CRUD (Create Read Update Delete) completa, para poder consultar y modificar el estado del clúster mediante la Rest API.

Una vez hemos ejecutado el comando por ejemplo para crear un Pod, enviamos una solicitud al

kube-api-server por POST mediante protocolo HTTP

La solicitud de cliente, pasa por un proceso de:

- Autenticación
- Autorización
- Admisión
- Validación
- Persistencia del estado en el almacén etcd

Las 3 fases principales (Autenticación, Autorización y Admisión), disponen de diferentes plugins para encontrar los recursos que estamos demandando gestionar así como el atravesar las diferentes fases en la autentificación



Un cliente del **kube-api-server** puede ser tanto un usuario que está empleando la herramienta **kubectl**, como un **Pod**, ambos atraviesan las siguientes fases de seguridad

59.1. Fase de Autenticación

El componente **kube-api-server** invoca los plugins necesarios, uno por uno, hasta que uno de ellos determina quien envía la solicitud.

El método de autentificación puede ser determinado por las cabeceras HTTP o bien mediante certificado

Una vez encontrada coincidencia con la autentificación, la solicitud devuelve el nombre de usuario (user id) y los grupos a los que pertenece el cliente al servidor **kube-api-server**. Finalmente, la fase de autenticación acaba y pasamos a la fase de autorización.

59.2. Fase de Autorización

El propósito de la fase de autorización es determinarse este usuario autenticado puede realizar la acción que está solicitando sobre el recurso solicitado

Por Ejemplo: ¿Puede el usuario crear un Pod en el namespace que está solicitando?

Si esta fase es finalizada con éxito, llega el momento de pasar a la siguiente fase

59.3. Fase de Admisión

Si la solicitud del usuario es solo para leer datos, la fase de admisión de control no se realiza

La solicitud pasa por todos los complementos, permitiendo a los complementos modificar el recurso por diferentes razones.

Por ejemplo, el complemento de cuenta de servicio aplica el servicio predeterminado cuenta dos

Pods que no especifican explícitamente eso. Y después de todo eso, el servidor **kube-api-server** valida el objeto, lo almacena en etcd y devuelve una respuesta.

Hasta ahora, hemos podido pasar la fase de autenticación utilizando el certificado autofirmado

59.4. Certificado de autorización

Podemos visualizar el certificado digital de autorización autofirmado, que se generó en la fase de creación del clúster, si ejecutamos el siguiente comando:

```
$ cat .kube/config

apiVersion: v1
clusters:
- cluster:
  certificate-authority-data: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURSO ...
  ...
```

Si observamos el archivo, podemos ver diferentes entradas como:

- **certificate-authority-data**
 - Datos de la entidad certificadora
- **client-certificate-data**
 - Datos del certificado de cliente
- **client-key-data**
 - Datos de la clave privada del certificado
- **server**
 - El endpoint URL del servidor maestro

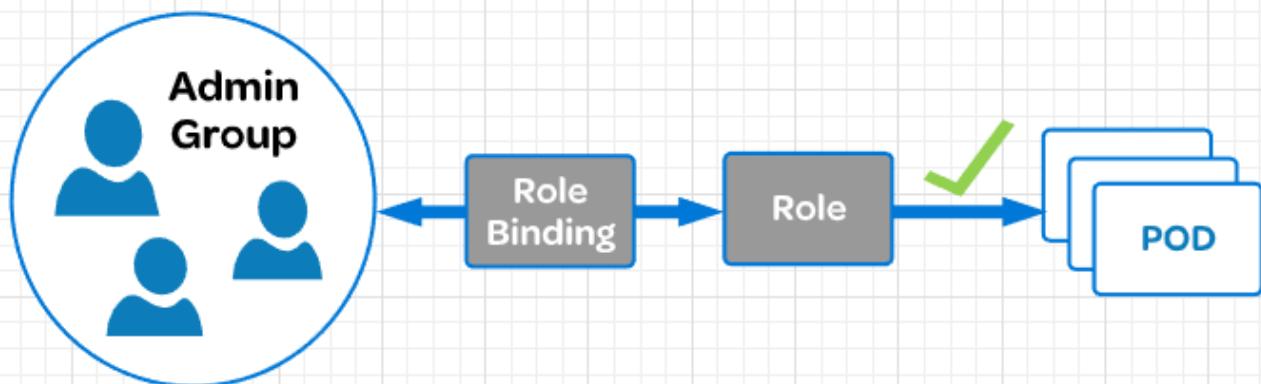
Se crea un recurso de cuenta de servicio para un pod para determinar cómo tiene control sobre el estado del clúster. Por ejemplo, la cuenta de servicio predeterminada no le permitirá enumerar los servicios en un espacio de nombres

Capítulo 60. RBAC (Role Based Access Control)

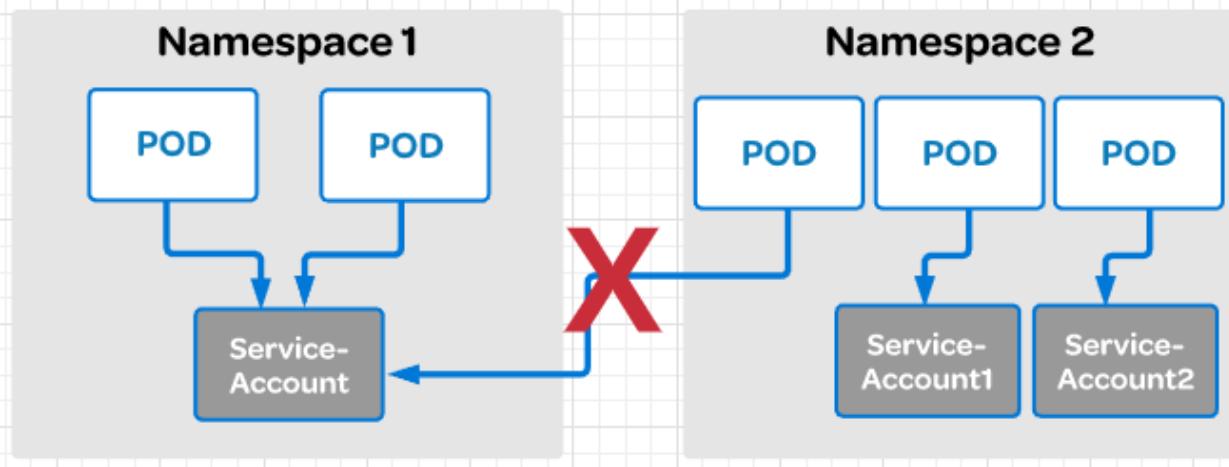
Es un método para regular el acceso a los recursos informáticos o de red en función de los roles de los usuarios individuales dentro de una empresa.

Roles and Access

RBAC (role-based access control) is used to prevent unauthorized users from modifying the cluster state.



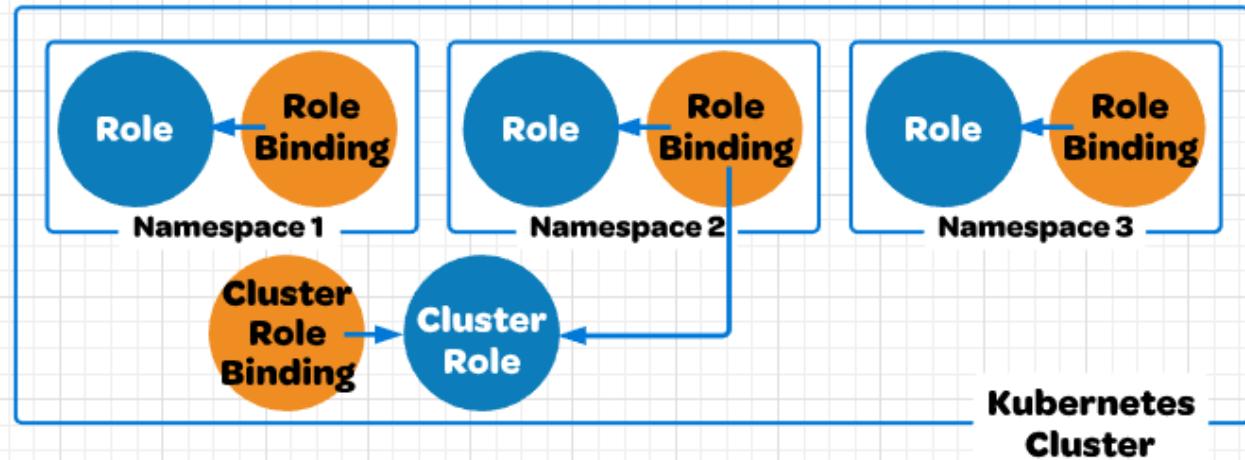
Service accounts are how a pod authenticates to the API server. A service account represents the identity of the app running in the pod.



Un usuario puede estar asociado con uno o más roles y cada rol puede realizar una acción sobre cierto recurso

Role and Role Bindings

RBAC authorization rules are configured through four resources, which can be organized into two groups.



Las reglas de autorización están configuradas en kubernetes a través de 4 recursos, que pueden resumirse en 2

- Roles y ClusterRoles
 - Determinan qué se podría realizar en qué recurso
- RoleBinding y ClusterRoleBinding
 - Determinan quién puede hacerlo



Los recursos Role y RoleBinding están sujetos a un espacio de nombres específico, mientras que los recursos ClusterRoles y ClusterRoleBinding son a nivel del clúster

60.1. ClusterRole

Puede ser usado para otorgar permiso de acceso a los nodos, acceso a endpoints que no son aplicaciones o como por ejemplo el chequeo de salud del endpoint "/healthz"

También puede ser usado para otorgar permisos a recursos sujetos a un namespace (como por ejemplo los pods) que puedan ser utilizados en todos los namespaces

Ámbito de operatividad a nivel de Clúster

60.2. Role

Un rol solo se puede utilizar para otorgar acceso a recursos dentro de un único espacio de nombres.

Ámbito de operatividad a nivel de namespace

60.3. RoleBinding

Un RoleBinding otorga los permisos definidos en un rol a un usuario o conjunto de usuarios.

Contiene una lista de elementos (usuarios, grupos o cuentas de servicio) y una referencia al rol que se otorga.

Se pueden otorgar permisos dentro de un espacio de nombres con un RoleBinding, o en todo el clúster con un ClusterRoleBinding

Un RoleBinding puede hacer referencia a un Rol en el mismo espacio de nombres.

Un RoleBinding siempre hace referencia únicamente a un Role, pero un Rol puede bindearse con más de una cuenta de servicio, usuario o grupo

Ámbito de operatividad a nivel de namespace



60.4. ClusterRoleBinding

Podemos usar un ClusterRoleBinding para otorgar permiso a nivel de clúster y en todos los espacios de nombres.

Ámbito de operatividad a nivel de Clúster



60.5. ServiceAccount

Un ServiceAccount, representa la identidad de una aplicación funcionando dentro de un Pod, el archivo que contiene el token, estará creado dentro del propio contenedor, en la ruta `/var/run/secrets/kubernetes.io/serviceaccount/token`

Cuando una aplicación, utiliza este token para comunicarse con el **kube-api-server**, el plugin interno de autentificación se pone en marcha (Este mecanismo está implementado en el núcleo del propio **kube-api-server**)



Si no se indica explícitamente qué cuenta de servicio tienen que usar los Pods en los manifiestos YAML, estos utilizarán la cuenta de servicio por defecto

La cuenta de servicio por defecto, podemos visualizarla si ejecutamos el siguiente comando:



```
$ kubectl get serviceaccounts
```

NAME	SECRETS	AGE
default	1	20d

Si intentas especificar una cuenta de servicio de un espacio de nombres diferente al que se encuentra operando el Pod, el Pod intentará alcanzar la cuenta de servicio de otro namespace diferente al actual, pero este proceso fallará, por que sólo podemos utilizar cuentas de servicio que se encuentren en el mismo namespace



Capítulo 61. Lab: RBAC (Consulta de unidades PV desde un namespace distinto al default)

Mediante este laboratorio, vamos a practicar el tema de la autorización, consiste en determinar lo que puede o no hacer, un Pod, un Usuario, etc.

Las reglas de autorización, se configuran de forma transparente al Usuario/Pods, de forma, que el control de acceso va a estar representado mediante Roles.

El objetivo del laboratorio será tener un pod funcionando con 2 contenedores, uno de los contenedores será el nginx y otro será un contenedor que actuará a modo de proxy, para que desde el contenedor del nginx podamos llevar a cabo invocaciones al kube-api-server y poder obtener información del propio kubernetes así como llevar a cabo las operaciones que determinemos necesarias.

Llevaremos a cabo la implementación de una política de acceso para que el contenedor de nginx únicamente pueda lanzar invocaciones al api de kubernetes para listar objetos de tipo **PersistentVolumes (PV)**.

61.1. Creando el archivo del laboratorio

Vamos a crear un archivo con nombre **service-reader-app.yml**, de forma que iremos añadiendo a dicho archivo todos los objetos de kubernetes que vayamos indicando.

61.2. Creando un namespace propio

Lo primero que vamos a hacer, es crear un namespace específico, añadiendo la siguiente instrucción:

```
apiVersion: v1
kind: Namespace
metadata:
  name: app-bill
```

61.3. Creando ClusterRole

Para recursos que no están sujetos a un namespace, como pueden ser los objetos de tipo node, pv, etc. Es donde encaja el ClusterRole.

Ahora, vamos a crear un ClusterRole para poder visualizar las unidades de persistencia PV

Añadimos al archivo lo siguiente:

```

apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: pods-reader-clusterrole
rules:
  - apiGroups:
    - ""
      resources:
      - persistentvolumes
    verbs:
      - get
      - list

```

61.4. Creando ClusterRoleBinding

Posteriormente, también vamos a necesitar crear el ClusterRoleBinding, añadimos lo siguiente:

```

apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: pods-reader-cr-binding
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: pods-reader-clusterrole
subjects:
  - kind: ServiceAccount
    name: default
    namespace: app-bill

```

61.5. Creando nuestro Rol personalizado

Ahora, definimos un Rol que defina que acciones se pueden realizar:

```

apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: app-bill
  name: service-reader-app
rules:
  - apiGroups: []
    verbs: ["get", "list"]
    resources: ["services"]

```

- Indicamos que el namespace de operación es app-bill

- Como nombre del rol indicamos service-reader-app
- Indicamos como reglas del rol, la posibilidad de uso de los verbos de la API Rest, get y list
- Indicamos que como recursos de uso que contempla el rol, services

61.6. Creando el RoleBinding

El rol, define qué acciones puede realizar, pero no define quién puede hacer uso del rol, para eso, tenemos el objeto RoleBinding que tiene ámbito de aplicación en un namespace concreto.

En nuestro caso, el rolebinding va vinculado con un tipo de objeto que es una cuenta de servicio, ya que nos interesa autorizar concretamente a un pod para indicar las operaciones que puede llevar a cabo:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  namespace: app-bill
  name: service-reader-app-binding
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: service-reader-app
subjects:
  - kind: ServiceAccount
    name: default
    namespace: app-bill
```

61.7. Creando un Pod que use el namespace

Ahora, llega el momento de definir el Pod que va a desplegarse en el namespace en cuestión:

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-1-container
  namespace: app-bill
spec:
  containers:
    - name: nginx
      image: nginx:latest
      ports:
        - containerPort: 80
    imagePullPolicy: IfNotPresent
    - image: linuxacademycontent/kubectl-proxy
      name: proxy
      imagePullPolicy: IfNotPresent
```

Llevamos a cabo la aplicación del manifiesto, ejecutando el siguiente comando:

```
$ kubectl apply -f service-reader-app.yml

namespace/app-bill created
clusterrole.rbac.authorization.k8s.io/pods-reader-clusterrole created
clusterrolebinding.rbac.authorization.k8s.io/pods-reader-cr-binding created
role.rbac.authorization.k8s.io/service-reader-app created
rolebinding.rbac.authorization.k8s.io/service-reader-app-binding created
pod/pod-with-1-container created

role.rbac.authorization.k8s.io/service-reader-app created
```

La imagen docker linuxacademycontent/kubectl-proxy invoca al arrancarse un script con nombre kubectl-proxy.sh con el siguiente contenido:

```
#!/bin/sh

API_SERVER="https://$KUBERNETES_SERVICE_HOST:$KUBERNETES_SERVICE_PORT"
CA_CRT="/var/run/secrets/kubernetes.io/serviceaccount/ca.crt"
TOKEN="$(cat /var/run/secrets/kubernetes.io/serviceaccount/token)"

kubectl proxy --server="$API_SERVER" --certificate-authority="$CA_CRT" --token="$TOKEN" --accept-paths='^.*'
```

61.8. Test: Levantando proxy en el host para comunicaciones con la API

Ahora, vamos a crear un proxy para comunicaciones internas en el propio host, de manera que al ejecutar el comando kubectl proxy sin indicar ningún otro argumento, estaremos consiguiendo hacer un **forwarding** directo al kube-api-server:

Ejecutamos en otra terminal el siguiente comando:

```
$ kubectl proxy

Starting to serve on 127.0.0.1:8001
```

Desde otra consola, vamos a intentar de listar los servicios del namespace **app-bill**, ejecutando el siguiente comando:

```
$ curl localhost:8001/api/v1/namespaces/app-bill/services
```

```
{  
  "kind": "ServiceList",  
  "apiVersion": "v1",  
  "metadata": {  
    "selfLink": "/api/v1/namespaces/app-bill/services",  
    "resourceVersion": "532865"  
  },  
  "items": []  
}
```

- Observamos que podemos listar los servicios del namespace app-bill a pensar de estar en la consola situados en el namespace default

61.9. Test: Utilizando el proxy del pod para comunicaciones con la API

Listamos en primer lugar los pods del namespace app-bill:

```
$ kubectl get pods --namespace app-bill  
  
NAME           READY   STATUS    RESTARTS   AGE  
pod-with-1-container   2/2     Running   0          78s
```

A continuación, abrimos una consola dentro del container del propio nginx (el primer container):

```
$ kubectl exec -it pod-with-1-container --namespace app-bill -- sh  
  
Defaulting container name to nginx.  
Use 'kubectl describe pod/pod-with-1-container -n app-bill' to see all of the containers in this pod.  
#
```

Intentamos listar las unidades de persistencia (Cluster Level), desde dentro del propio Pod, haciendo uso del proxy, que lo tendrá levantado el segundo contenedor que está definido en el propio pod.

Como las comunicaciones intra-pod van por localhost, ejecutamos dicho comando:

```
# curl localhost:8001/api/v1/persistentvolumes
```

```
{  
  "kind": "PersistentVolumeList",  
  "apiVersion": "v1",  
  "metadata": {  
    "selfLink": "/api/v1/persistentvolumes",  
    "resourceVersion": "535251"  
  },  
  "items": [  
...  
]
```

- Podemos acceder desde dentro del Pod a recursos que están a nivel de Clúster, como por ejemplo, las unidades PV



Podemos acceder a los recursos de nivel de clúster, utilizando el binding al ClusterRole que hemos creado, ya que la cuenta de servicio que hemos indicado con el clusterRoleBinding es con el namespace de app-bill

Realizamos una prueba final, intentando acceder a recursos que no nos son permitidos, como por ejemplo, llevar a cabo el listado de los namespaces:

```
# curl localhost:8001/api/v1/namespaces
```

```
{  
  "kind": "Status",  
  "apiVersion": "v1",  
  "metadata": {},  
  "status": "Failure",  
  "message": "namespaces is forbidden: User \"system:serviceaccount:default:default\" cannot list resource \"namespaces\"  
in API group \"\" at the cluster scope",  
  "reason": "Forbidden",  
  "details": {  
    "kind": "namespaces"  
  },  
  "code": 403  
}
```

Capítulo 62. Lab: RBAC (Consulta de unidades PV desde el namespace default)

Mediante este laboratorio, vamos a practicar el tema de la autorización, consiste en determinar lo que puede o no hacer, un Pod, un Usuario, etc.

Las reglas de autorización, se configuran de forma transparente al Usuario/Pods, de forma, que el control de acceso va a estar representado mediante Roles.

El objetivo del laboratorio será tener un pod funcionando con 1 contenedor, la imagen de docker que utilizaremos será el nginx, desde el contenedor del nginx podremos llevar a cabo invocaciones al kube-api-server y poder obtener información del propio kubernetes así como llevar a cabo las operaciones que determinemos necesarias.

Llevaremos a cabo la implementación de una política de acceso para que el contenedor de nginx únicamente pueda lanzar invocaciones al api de kubernetes para listar objetos de tipo **PersistentVolumes (PV)**.



62.1. Creando el archivo del laboratorio

Vamos a crear un archivo con nombre **service-reader-app-withouth-forwarding.yml**, de forma que iremos añadiendo a dicho archivo todos los objetos de kubernetes que vayamos indicando.

62.2. Creando ClusterRole

Para recursos que no están sujetos a un namespace, como pueden ser los objetos de tipo node, pv, etc. Es donde encaja el ClusterRole.

Ahora, vamos a crear un ClusterRole para poder visualizar las unidades de persistencia PV

Añadimos al archivo lo siguiente:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: pods-reader-clusterrole
rules:
  - apiGroups:
    - ""
      resources:
        - persistentvolumes
      verbs:
        - get
        - list
```

62.3. Creando ClusterRoleBinding

Posteriormente, también vamos a necesitar crear el ClusterRoleBinding, añadimos lo siguiente:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: pods-reader-cr-binding
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: pods-reader-clusterrole
subjects:
  - kind: ServiceAccount
    name: default
    namespace: default
```

62.4. Creando nuestro Rol personalizado

Ahora, definimos un Rol que defina qué acciones se pueden realizar:

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

- Indicamos que el namespace de operación es app-bill
- Como nombre del rol indicamos service-reader-app
- Indicamos como reglas del rol, la posibilidad de uso de los verbos de la API Rest, get y list
- Indicamos que como recursos de uso que contempla el rol, services

62.5. Creando el RoleBinding

El rol, define qué acciones puede realizar, pero no define quién puede hacer uso del rol, para eso, tenemos el objeto RoleBinding que tiene ámbito de aplicación en un namespace concreto.

En nuestro caso, el rolebinding va vinculado con un tipo de objeto que es una cuenta de servicio, ya que nos interesa autorizar concretamente a un pod para indicar las operaciones que puede llevar a cabo:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  namespace: default
  name: service-reader-app-binding
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: service-reader-app
subjects:
- kind: ServiceAccount
  name: default
  namespace: default
```



62.6. Creando un Pod que use el namespace

Ahora, llega el momento de definir el Pod que va a desplegarse en el namespace en cuestión:

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-1-container
  namespace: default
spec:
  containers:
    - name: nginx
      image: nginx:latest
      ports:
        - containerPort: 80
      imagePullPolicy: IfNotPresent
```



Llevamos a cabo la aplicación del manifiesto, ejecutando el siguiente comando:

```
$ kubectl apply -f service-reader-app-withouth-forwarding.yml
role.rbac.authorization.k8s.io/service-reader-app-withouth-forwarding created
```



62.7. Test: Utilizando el proxy del pod para comunicaciones con la API

Listamos en primer lugar los pods del namespace app-bill:

```
$ kubectl get pods --namespace app-bill

NAME                  READY  STATUS    RESTARTS  AGE
pod-with-1-container  1/1    Running   0          78s
```

A continuación, abrimos una consola dentro del container del propio nginx:

```
$ kubectl exec -it pod-with-1-container --namespace app-bill -- sh
#
```

Ejecutamos dentro de dicho contenedor las siguientes instrucciones:

```
# Point to the internal API server hostname
APISERVER=https://kubernetes.default.svc

# Path to ServiceAccount token
SERVICEACCOUNT=/var/run/secrets/kubernetes.io/serviceaccount

# Read this Pod's namespace
NAMESPACE=$(cat ${SERVICEACCOUNT}/namespace)

# Read the ServiceAccount bearer token
TOKEN=$(cat ${SERVICEACCOUNT}/token)

# Reference the internal certificate authority (CA)
CACERT=${SERVICEACCOUNT}/ca.crt
```

Seguidamente, también dentro de la consola de dicho contenedor, procedemos en primer lugar a explorar el API con el token:

```
# curl --cacert ${CACERT} --header "Authorization: Bearer ${TOKEN}" -X GET ${APISERVER}/api
{
  "kind": "APIVersions",
  "versions": [
    "v1"
  ],
  "serverAddressByClientCIDRs": [
    {
      "clientCIDR": "0.0.0.0/0",
      "serverAddress": "192.168.15.100:6443"
    }
  ]
}
```

Después, intentamos listar las unidades de persistencia (Cluster Level), desde dentro del propio Pod:

```
# curl --cacert ${CACERT} --header "Authorization: Bearer ${TOKEN}" -X GET ${APISERVER}/api/v1/persistentvolumes
{
  "kind": "PersistentVolumeList",
  "apiVersion": "v1",
  "metadata": {
    "resourceVersion": "124317"
  },
  "items": []
}
```

- Podemos acceder desde dentro del Pod a recursos que están a nivel de Clúster, como por ejemplo, las unidades PV



Podemos acceder a los recursos de nivel de clúster, utilizando el binding al ClusterRole que hemos creado, ya que la cuenta de servicio que hemos indicado con el clusterRoleBinding es con el namespace de app-bill

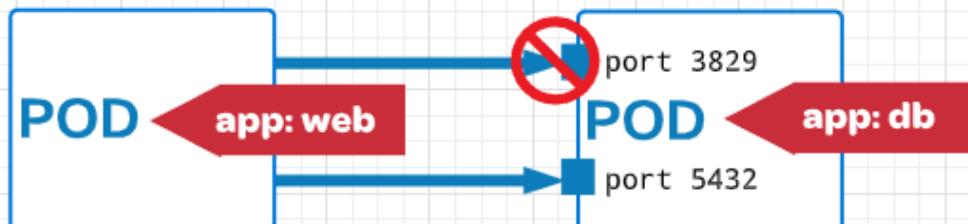
Realizamos una prueba final, intentando acceder a recursos que no nos son permitidos, como por ejemplo, llevar a cabo el listado de los namespaces:

```
# curl --cacert ${CACERT} --header "Authorization: Bearer ${TOKEN}" -X GET ${APISERVER}/api/v1/namespaces
{
  "kind": "Status",
  "apiVersion": "v1",
  "metadata": {},
  "status": "Failure",
  "message": "namespaces is forbidden: User \"system:serviceaccount:default:default\" cannot list resource \"namespaces\" in API group \"\" at the cluster scope",
  "reason": "Forbidden",
  "details": {
    "kind": "namespaces"
  },
  "code": 403
}
```

Capítulo 63. NetworkPolicy

Network Policies

Network policies use selectors to apply rules to pods for communication throughout the cluster.



Las políticas de red, nos permiten especificar qué pods pueden comunicarse con otros pods.

Este tema, nos va a ayudar a asegurar la comunicación entre Pods, lo que nos va a permitir identificar reglas de entrada y de salida.

Podemos aplicar una política de red a un Pod utilizando selectores de Pod o espacios de nombres.

Incluso, podríamos elegir un rango de bloque CIDR para aplicar la política de red.

Por defecto, cuando se crean Pods en cuanto a comunicaciones están abiertos, pueden ser accedidos por cualquier persona, así que es muy importante asegurarnos que sólo los servicios o Pods necesarios pueden comunicarse entre sí.

Las políticas de red se aplican a los Pods que coinciden con un selector de etiqueta determinado o namespace.

Capítulo 64. Lab: NetworkPolicy

Mediante este laboratorio, vamos a practicar algunas operaciones de reglas ingress y egress, de forma que podremos regular el tráfico en las comunicaciones de nuestros Pods.

64.1. Creando política de red, denegando comunicaciones

Vamos a crear una política de red, que impida la comunicación entre Pods, la política estará afectada a todo Pod que se encuentre en funcionamiento en el mismo namespace.

Creamos el archivo con nombre **network-policy-deny-all-communications.yml** y le agregamos el siguiente contenido:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: network-policy-deny-all-communications
spec:
  podSelector: {}
  policyTypes:
    - Ingress
```

Ejecutamos la creación de la política:

```
$ kubectl apply -f network-policy-deny-all-communications.yml
networkpolicy.networking.k8s.io/network-policy-deny-all-communications created
```

64.2. Creando Pods afectados por la política

Ahora, vamos a crear en formato inline, un deployment de un sistema nginx:

```
$ kubectl create deployment nginx --image=nginx
deployment.apps/nginx created
```

A continuación, vamos a crear un servicio que exponga el deployment:

```
$ kubectl expose deployment nginx --port=80
service/nginx exposed
```

64.3. Comprobando comunicaciones

Ahora, vamos a comprobar las comunicaciones con el Pod

Vamos a crear directamente un Pod y nos conectaremos dentro del mismo para interactuar, ejecutaremos el siguiente comando:

```
$ kubectl run --generator=run-pod/v1 busybox --rm -it --image=busybox sh
```

```
If you don't see a command prompt, try pressing enter.  
/ #
```

Ahora, vamos a intentar acceder al servicio, desde dentro del contenedor ejecutamos:

```
/ # wget --spider --timeout=1 nginx
```

```
Connecting to nginx (10.96.173.78:80)  
wget: download timed out
```

- Observamos que la petición al servicio es denegada, produciéndose un timeout

64.4. Estableciendo regla de comunicaciones (Política Ingress a nivel de Pod)

Supongamos que tenemos el caso de un frontend y de un backend.

Queremos conseguir, que el el Pod que sea de frontend, sólo pueda comunicarse con el pod de backend mediante un puerto determinado, para conseguir ese efecto, debemos de especificar la regla de ingress.

Vamos a crear el archivo **frontend-backend-network-policy.yml** y le agregamos el siguiente contenido:

```

apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: frontend-backend-network-policy
spec:
  podSelector:
    matchLabels:
      app: backend
  ingress:
    - from:
        - podSelector:
            matchLabels:
              app: frontend
  ports:
    - port: 8080

```

- Indicamos que la política afecta a los pods que tengan la etiqueta **app:backend**
- Indicamos como regla de ingress, la comunicación intera sólo será permitida por pods que tenga la etiqueta **app:frontend**
- Adicionalmente, indicamos que el pod que tenga la etiqueta **app:frontend** sólo se podrá comunicar con el pod que tenga la etiqueta de **app:backend** a través del puerto **8080**

Ejecutamos la creación de la política de ámbito de Pod:

```

$ kubectl apply -f frontend-backend-network-policy.yml

networkpolicy.networking.k8s.io/frontend-backend-network-policy created

```

A continuación, vamos a crear un archivo, donde vamos a definir 3 pods.

Un Pod será el backend, otro un pod de un sistema frontend y otro pod, otro sistema frontend distinto.

Vamos a crear un archivo con nombre **pods-for-network-policy-scope-pod.yml**

```

apiVersion: v1
kind: Pod
metadata:
  name: pod-with-label-frontend
  labels:
    app: frontend
spec:
  containers:
    - name: nginx
      image: nginx:alpine
      ports:
        - containerPort: 80

```

```

apiVersion: v1
kind: Pod
metadata:
  name: pod-with-label-httpd
  labels:
    app: httpd
spec:
  containers:
    - name: httpd
      image: httpd:alpine
      ports:
        - containerPort: 80

```

```

apiVersion: v1
kind: Pod
metadata:
  name: pod-with-label-backend
  labels:
    app: backend
spec:
  containers:
    - name: tomcat
      image: tomcat:alpine
      ports:
        - containerPort: 8080

```

Ejecutamos la creación de los Pods:

```

$ kubectl apply -f pods-for-network-policy-scope-pod.yml

pod/pod-with-label-frontend created
pod/pod-with-label-httpd created
pod/pod-with-label-backend created

```

Nos conectamos dentro del Pod **pod-with-label-httpd** y probamos a hacer una petición

directamente a la IP del Pod, para comprobar que no nos es posible:

```
$ kubectl get pods -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE	NOMINATED NODE	READINESS	GATES
nginx-86c57db685-fmgkh	1/1	Running	0	31m	10.46.0.3	kubeminion1	<none>	<none>	
pod-with-label-backend	1/1	Running	0	2m9s	10.36.0.2	kubeminion2	<none>	<none>	
pod-with-label-frontend	1/1	Running	0	2m9s	10.36.0.0	kubeminion2	<none>	<none>	
pod-with-label-httdp	1/1	Running	0	2m9s	10.46.0.5	kubeminion1	<none>	<none>	

```
$ kubectl exec -it pod-with-label-httdp bash
```

```
bash-5.0# wget --spider --timeout=10 10.36.0.2:8080
Connecting to 10.36.0.2:8080 (10.36.0.2:8080)
wget: download timed out
```

- Observamos que la petición alcanza el tiempo de timeout de 10s y falla

Ahora, vamos a probar a conectarnos dentro del Pod **pod-with-label-frontend** y probamos a hacer una petición directamente a la IP del Pod, para comprobar que en este caso sí es posible:

```
$ kubectl exec -it pod-with-label-frontend sh
```

```
/# wget --spider --timeout=10 10.36.0.2:8080
Connecting to 10.36.0.2:8080 (10.36.0.2:8080)
/#
```

- En este caso, observamos que no se muestra error y la petición transcurre de forma correcta, con lo que la comunicación, ha sido posible

64.5. Estableciendo regla de comunicaciones (Política Ingress a nivel de NameSpace)

Otro caso de establecimiento de políticas de red, es cerrar a nivel del namespace, en este caso se podrán comunicar con el Pod que tenga la etiqueta **app:backend** cualquier pod que tenga la etiqueta **app:frontend** mientras estén en el mismo namespace.

Vamos a crear el archivo **frontend-backend-network-policy-namespace.yml** y le agregamos el siguiente contenido:

```

apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  namespace: namespace-with-network-policy
  name: frontend-backend-network-policy-namespace
spec:
  podSelector:
    matchLabels:
      app: backend
  ingress:
    - from:
        - namespaceSelector:
            matchLabels:
              app: frontend
      ports:
        - port: 8080

```

Vamos a crear primero 2 namespaces, uno donde estará presente la política de red y otro en la que no:

Creamos un archivo con nombre **namespace-with-network-policy.yml** y le agregamos el siguiente contenido:

```

apiVersion: v1
kind: Namespace
metadata:
  name: namespace-with-network-policy
  labels:
    app: frontend

```

Ejecutamos la creación del namespace:

```

$ kubectl apply -f namespace-with-network-policy

namespace/namespace-with-network-policy created

```

Ahora, vamos a crear el segundo namespace:

```

$ kubectl create namespace namespace-without-network-policy

namespace/namespace-without-network-policy created

```

Por último, vamos a ejecutar la creación de la política de red:

```
$ kubectl apply -f frontend-backend-network-policy-namespace.yml  
networkpolicy.networking.k8s.io/frontend-backend-network-policy-namespace created
```

A continuación, vamos a crear un archivo para desplegar 4 Pods:

- 1 Pod será el backend
 - Tendrá la etiqueta **app:backend**
 - Indicaremos que se desplegará en el namespace **namespace-with-network-policy**
- 1 Pod será el frontend nginx
 - Tendrá la etiqueta **app:frontend**
 - Indicaremos que se desplegará en el namespace **namespace-with-network-policy**
- 1 Pod será el frontend httpd
 - Tendrá la etiqueta **app:frontend**
 - Indicaremos que se desplegará en el namespace **namespace-without-network-policy**

Creamos el archivo **pods-for-network-policy-scope-namespace.yml** y le agregamos el siguiente contenido:



Fondos Europeos



GOBIERNO
DE ESPAÑA
MINISTERIO
DE EDUCACIÓN,
FORMACIÓN PROFESIONAL
Y DEPORTE

Cofinanciado por
la Unión Europea



```

apiVersion: v1
kind: Pod
metadata:
  namespace: namespace-with-network-policy
  name: pod-with-label-frontend-namespace-with-network-policy
  labels:
    app: frontend
spec:
  containers:
    - name: nginx
      image: nginx:alpine
      ports:
        - containerPort: 80
---
apiVersion: v1
kind: Pod
metadata:
  namespace: namespace-withouth-network-policy
  name: pod-with-label-httpd
  labels:
    app: httpd
spec:
  containers:
    - name: httpd
      image: httpd:alpine
      ports:
        - containerPort: 80
---
apiVersion: v1
kind: Pod
metadata:
  namespace: namespace-with-network-policy
  name: pod-with-label-backend
  labels:
    app: backend
spec:
  containers:
    - name: tomcat
      image: tomcat:alpine
      ports:
        - containerPort: 8080

```

Ejecutamos la creación de los Pods:



```
$ kubectl apply -f pods-for-network-policy-scope-namespace.yml

pod/pod-with-label-frontend-namespace-with-network-policy created
pod/pod-with-label-httpd created
pod/pod-with-label-backend created
```

Visualizamos los Pods del namespace **namespace-with-network-policy**:

```
$ kubectl get pods --namespace namespace-with-network-policy

NAME                               READY   STATUS    RESTARTS   AGE     IP      NODE
NOMINATED NODE  READINESS GATES
pod-with-label-backend             1/1    Running   0          5m38s  10.46.0.8  kubeminion1
<none>    <none>
pod-with-label-frontend-namespace-with-network-policy  1/1    Running   0          5m38s  10.36.0.3  kubeminion2
<none>    <none>
```

Ahora, visualizamos los Pods del namespace **namespace-withouth-network-policy**:

```
$ kubectl get pods --namespace namespace-withouth-network-policy

NAME           READY   STATUS    RESTARTS   AGE     IP      NODE   NOMINATED NODE  READINESS GATES
pod-with-label-httpd  1/1    Running   0          6m7s   10.46.0.7  kubeminion1  <none>    <none>
```

A continuación, vamos a intentar hacer un curl desde el Pod que se encuentra en el namespace **namespace-withouth-network-policy** cuyo nombre es **pod-with-label-httpd**, hacia el pod que se encuentra en el namespace **namespace-with-network-policy** cuyo nombre es **pod-with-label-backend**.

Nos conectamos primero al Pod:

```
$ kubectl exec -it pod-with-label-httpd --namespace namespace-withouth-network-policy bash

bash-5.0# wget --spider --timeout=10 10.46.0.8:8080
Connecting to 10.46.0.8:8080 (10.46.0.8:8080)
wget: download timed out
```

- Observamos que nos da un timeout de conexión

Ahora, vamos a intentar hacer el curl desde el Pod que se encuentra en el namespace **namespace-with-network-policy** cuyo nombre es **pod-with-label-frontend-namespace-with-network-policy**, hacia el pod que se encuentra en el namespace **namespace-with-network-policy** cuyo nombre es **pod-with-label-backend**.

Nos conectamos al Pod:

```
$ kubectl exec -it pod-with-label-frontend-namespace-with-network-policy --namespace namespace-with-network-policy sh  
/ # wget --spider --timeout=10 10.46.0.8:8080  
Connecting to 10.46.0.8:8080 (10.36.0.4:8080)
```

- Observamos que en este caso, sí que obtenemos respuesta

64.6. Estableciendo regla de comunicaciones (Política Ingress a nivel de IP)

Otro uso interesante en el establecimiento de las comunicaciones entre los Pods, podría ser bloquear comunicaciones de entrada según ip del nodo donde se encuentre el pod

Para realizar esta práctica, utilizaremos el namespace por defecto que nos trae kubernetes **default**

Vamos a crear un Pod que desplegaremos de manera explícita en el nodo **kubeminion1**.

Creamos un archivo con nombre **pods-to-nodes-ingress-ip.yml** y le agregamos el siguiente contenido:



```

apiVersion: v1
kind: Pod
metadata:
  name: pod-to-node-kubeminion1
  labels:
    app: frontend
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: kubernetes.io/hostname
                operator: In
                values:
                  - kubeminion1
  containers:
    - name: httpd-in-node-kubeminion2
      image: httpd:alpine
      ports:
        - containerPort: 80
---
apiVersion: v1
kind: Pod
metadata:
  name: pod-to-node-kubeminion2
  labels:
    app: frontend
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: kubernetes.io/hostname
                operator: In
                values:
                  - kubeminion2
  containers:
    - name: httpd-in-node-kubeminion2
      image: httpd:alpine
      ports:
        - containerPort: 80

```

Ejecutamos la creación de los Pods:



```
$ kubectl apply -f pod-to-nodes-ingress-ip.yml
```

```
pod/pod-to-node-kubeminion1 created  
pod/pod-to-node-kubeminion2 created
```

Comprobamos que cada Pod ha caído en un nodo diferente:

```
$ kubectl get pods -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE	NOMINATED NODE	READINESS GATES
pod-to-node-kubeminion1	1/1	Running	0	68s	10.46.0.3	kubeminion1	<none>	<none>
pod-to-node-kubeminion2	1/1	Running	0	68s	10.36.0.0	kubeminion2	<none>	<none>

Ahora, vamos a crear el archivo **network-policy-ingress-ip-level.yml** y le agregamos el siguiente contenido:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: network-policy-ingress-ip-level
spec:
  podSelector:
    matchLabels:
      app: frontend
  ingress:
  - from:
    - ipBlock:
      cidr: 192.168.15.102/24
```

- La política, bloqueará el tráfico desde el nodo cuya IP sea 192.168.15.102 (kubeminion2) a cuyos Pods estén etiquetados con **app:frontend**

Ejecutamos la creación de la política:

```
$ kubectl apply -f network-policy-ingress-ip-level.yml
```

```
networkpolicy.networking.k8s.io/network-policy-ingress-ip-level created
```

Comprobamos también las reglas de red que hay presentes en el namespace:

```
$ kubectl get networkpolicies
```

NAME	POD-SELECTOR	AGE
network-policy-ingress-ip-level	app=frontend	3m18s

Probamos a obtener desde el pod **pod-to-node-kubeminion2** el index.html del pod **pod-to-node-**

kubeminion1:

```
$ kubectl exec -it pod-to-node-kubeminion2 bash  
  
bash-5.0# wget --spider --timeout=10 10.46.0.3:80  
Connecting to 10.46.0.3:80 (10.46.0.3:80)  
wget: can't connect to remote host (10.36.0.0): Operation timed out
```

- Observamos que no obtenemos respuesta, el Pod por encontrarse en la IP 192.168.15.102 tiene cortado el tráfico entrante

64.7. Estableciendo regla de comunicaciones (Política Egress a nivel de Pod)

De igual forma que podemos configurar las reglas de Ingress (Entrada), podemos llevar a cabo configuración de reglas de salida (Egress)

Primero, vamos a crear un archivo para denegar de forma explícita todo el tráfico de salida de los Pods que se encuentren en el namespace default

Vamos a crear un archivo con nombre **network-policy-egress-policy-deny-all-by-default.yml** y le agregamos el siguiente contenido:

```
apiVersion: networking.k8s.io/v1  
kind: NetworkPolicy  
metadata:  
  name: default-deny-egress  
spec:  
  podSelector: {}  
  policyTypes:  
    - Egress
```

Ejecutamos la creación de la regla:

```
$ kubectl apply -f network-policy-egress-policy-deny-all-by-default.yml  
  
networkpolicy.networking.k8s.io/default-deny-egress created
```

Ahora creamos un archivo con nombre **network-policy-egress-policy-scope-pod.yml** y le agregamos el siguiente contenido:

```

apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: egress-netpol
spec:
  podSelector:
    matchLabels:
      app: bill-system
  egress:
    - to:
        - podSelector:
            matchLabels:
              app: bill-system
      ports:
        - port: 80
  ingress:
    - from:
        - podSelector:
            matchLabels:
              app: bill-system
      ports:
        - port: 80

```

- Permitimos reglas de comunicación de salida de los Pods, a través del puerto 80, a todo Pod que cumpla el etiquetado **app:frontend**

Ejecutamos la creación de la regla:

```
$ kubectl apply -f network-policy-egress-policy-scope-pod.yml
networkpolicy.networking.k8s.io/egress-netpol created
```

Ahora, vamos a crear un archivo con nombre **pods-to-egress-rules.yml** y le agregamos el siguiente contenido:



```

apiVersion: v1
kind: Pod
metadata:
  name: pod-with-frontend
  labels:
    app: bill-system
spec:
  containers:
    - name: nginx
      image: nginx:alpine
      ports:
        - containerPort: 80
---

```

```

apiVersion: v1
kind: Pod
metadata:
  name: pod-with-backend
  labels:
    app: bill-system
spec:
  containers:
    - name: tomcat
      image: tomcat:alpine
      ports:
        - containerPort: 8080

```

Ejecutamos la creación de los Pods:

```
$ kubectl apply -f pods-to-egress-rules.yml
```

```
pod/pod-with-frontend created
pod/pod-with-backend created
```

Listamos los Pods:

```
$ kubectl get pods -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE	NOMINATED NODE	READINESS GATES
pod-with-backend	1/1	Running	0	4m14s	10.36.0.1	kubeminion2	<none>	<none>
pod-with-frontend	1/1	Running	0	4m14s	10.36.0.0	kubeminion2	<none>	<none>

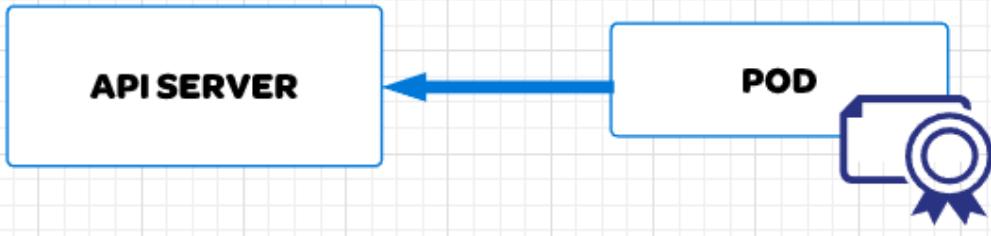
Si nos conectamos al Pod de frontend o al de backend, tendremos el siguiente comportamiento:

- Desde el pod backend podrá comunicarse a través del puerto 80 con el pod de frontend
- Desde el pod frontend no será posible comunicarse a través del puerto 8080 con el pod de backend

Capítulo 65. Certificados TLS

CA and TLS

The certificate authority is used to generate a TLS certificate and authenticate with the API server.



Cuando hablamos de la CA de nuestro clúster de kubernetes, estamos hablando de la autoridad de certificación que se utiliza para generar certificados TLS y autenticarse con el kube-api-server

Una entidad certificadora (CA) se utiliza para generar certificados TLS

Las comunicaciones mediante el kube-api-server van cifradas y autenticadas mediante certificados

Cuando se crean Pods en el clúster de kubernetes, automáticamente se monta un paquete de certificado accesible por ellos, de forma que los Pods si no se dice lo contrario, utilizan una cuenta de servicio para montar secrets que contendrán información de autentificación con el kube-api-server

Capítulo 66. Lab: Certificados TLS

Mediante este laboratorio, llevaremos a cabo la obtención de un certificado que está mataselado por la CA que está utilizando la maquinaria interna de kubernetes, de manera que podamos llevar a cabo la sustitución de los certificados que usan los kubelet en caso necesario, o bien, utilizar dicho certificado para algún otro propósito en alguna de nuestras aplicaciones.

66.1. Mostrando la CA de un pod del clúster

Vamos a mostrar la CA que se monta dentro de un Pod cuando este se pone en marcha

Primero, vamos a ejecutar un Pod:

```
$ kubectl run nginx --image=nginx:latest --generator=run-pod/v1 --port=80  
pod/nginx created
```

A continuación, vamos a listar los archivos que están montados dentro del pod, en lo que respecta a la cuenta de operación de servicio:

```
$ kubectl exec nginx -- ls /var/run/secrets/kubernetes.io/serviceaccount  
ca.crt  
namespace  
token
```

- Observamos
 - El archivo del certificado de la CA
 - El archivo que contiene dentro el nombre del namespace al que el Pod está sujeto, por defecto default
 - El archivo que contiene el token de la cuenta de servicio para que el Pod pueda interactuar con normalidad con el kube-api-server



Si queremos generar un nuevo certificado para un objeto en kubernetes, en realidad, ya hay uno construido con la API que nos permite crear y utilizar certificados personalizados

66.2. Instalando la herramienta CFSSL

Podríamos crear nuestro propio certificado, para ello, lo primero que vamos a necesitar es crear una solicitud de certificado (CSR), para que posteriormente se firme la solicitud con la (CA) y el certificado sea válido.

Vamos a utilizar como herramienta de ayuda la herramienta CFSSL, podemos encontrar más

información al respecto en la siguiente URL: <https://github.com/cloudflare/cfssl>

Ejecutamos el comando de descarga de la herramienta:

```
$ wget -q --timestamping https://pkg.cfssl.org/R1.2/cfssl_linux-amd64 https://pkg.cfssl.org/R1.2/cfssljson_linux-amd64
```

Ahora, hacemos ejecutables los binarios:

```
$ chmod +x cfssl_linux-amd64 cfssljson_linux-amd64
```

Movemos los archivos a una localización del sistema para que puedan ser ejecutados desde cualquier punto:

```
$ sudo mv cfssl_linux-amd64 /usr/local/bin/cfssl  
$ sudo mv cfssljson_linux-amd64 /usr/local/bin/cfssljson
```

Verificamos que el binario se consigue ejecutar:

```
$ cfssl version
```

Version: 1.2.0
Revision: dev
Runtime: go1.6

66.3. Generando la solicitud de certificado CSR

Ahora, toca el momento de generar la solicitud de certificado CSR, ejecutamos en consola lo siguiente:



```

cat <<EOF | cfssl genkey - | cfssljson -bare server
{
  "hosts": [
    "my-svc.my-namespace.svc.cluster.local",
    "my-pod.my-namespace.pod.cluster.local",
    "192.168.15.100",
    "10.96.0.1"
  ],
  "CN": "system:node:kubernetes-master",
  "key": {
    "algo": "ecdsa",
    "size": 256
  },
  "names": [
    {"O": "system:nodes"
  }]
}
EOF
2020/01/11 00:04:40 [INFO] generate received request
2020/01/11 00:04:40 [INFO] received CSR
2020/01/11 00:04:40 [INFO] generating key: ecdsa-256
2020/01/11 00:04:40 [INFO] encoded CSR

```

Listamos los archivos del directorio, y observamos que se han creado 2 archivos:

```

$ 
server.csr
server-key.pem

```

- El archivo de solicitud de certificado
- La clave privada

A continuación, vamos a proceder a pasarle al kube-api-server la solicitud de certificado (CSR) para que pueda firmarla, para ello, ejecutamos la siguiente instrucción en consola:

```
$ cat <<EOF | kubectl create -f -
apiVersion: certificates.k8s.io/v1beta1
kind: CertificateSigningRequest
metadata:
  name: pod-csr.web
spec:
  groups:
    - system:authenticated
  request: $(cat server.csr | base64 | tr -d '\n')
  signerName: kubernetes.io/kube-apiserver-client
  usages:
    - digital signature
    - key encipherment
    - server auth
EOF
```

certificatesigningrequest.certificates.k8s.io/pod-csr.web created

Vamos a listar en kubernetes, las solicitudes de certificado CSR:

```
$ kubectl get csr
NAME      AGE REQUESTOR      CONDITION
pod-csr.web  85s  kubernetes-admin  Pending
```

- Observamos que la solicitud se encuentra en estado pendiente
- También que el solicitante es el usuario kubernetes-admin

Procedemos a continuación, a ejecutar una descripción detallada del certificado:

```
$ kubectl describe csr pod-csr.web
Name:      pod-csr.web
Labels:    <none>
Annotations: <none>
CreationTimestamp: Sat, 11 Jan 2020 00:08:50 +0100
Requesting User: kubernetes-admin
Status:     Pending
Subject:
  Common Name: my-pod.my-namespace.pod.cluster.local
  Serial Number:
Subject Alternative Names:
  DNS Names: my-svc.my-namespace.svc.cluster.local
               my-pod.my-namespace.pod.cluster.local
  IP Addresses: 192.168.15.100
               10.96.0.1
Events: <none>
```

66.4. Aprobando el certificado

A continuación, vamos a aprobar la solicitud de certificado, ejecutando en consola lo siguiente:

```
$ kubectl certificate approve pod-csr.web  
certificatesigningrequest.certificates.k8s.io/pod-csr.web approved
```

Listamos de nuevo las CSR en kubernetes y obtenemos que el estado del CSR ha cambiado:

```
$ kubectl get csr  
NAME      AGE      REQUESTOR      CONDITION  
pod-csr.web  6m10s  kubernetes-admin  Approved
```

Podemos ver el detalle completo del certificado dentro del archivo CSR, ejecutando el siguiente comando:

```
$ kubectl get csr pod-csr.web -o yaml  
  
apiVersion: certificates.k8s.io/v1beta1  
kind: CertificateSigningRequest  
metadata:  
  creationTimestamp: "2020-01-10T23:08:50Z"  
  name: pod-csr.web  
  resourceVersion: "627181"  
  selfLink: /apis/certificates.k8s.io/v1beta1/certificatesigningrequests/pod-csr.web  
  uid: eacccd30-738d-4d2f-8b8b-8a62c4856fb4  
spec:  
  groups:  
    - system:masters  
    - system:authenticated  
  request: LS0tLS1CRUdjTiBDRVJUSUZJQ  
...  
...
```

66.5. Extrayendo el certificado

Ahora, podemos extraer el certificado y decodificar el certificado para obtener el archivo .crt con el certificado en sí

Ejecutamos en la consola el siguiente comando:

```
$ kubectl get csr pod-csr.web -o jsonpath='{.status.certificate}' \  
| base64 --decode > server.crt
```

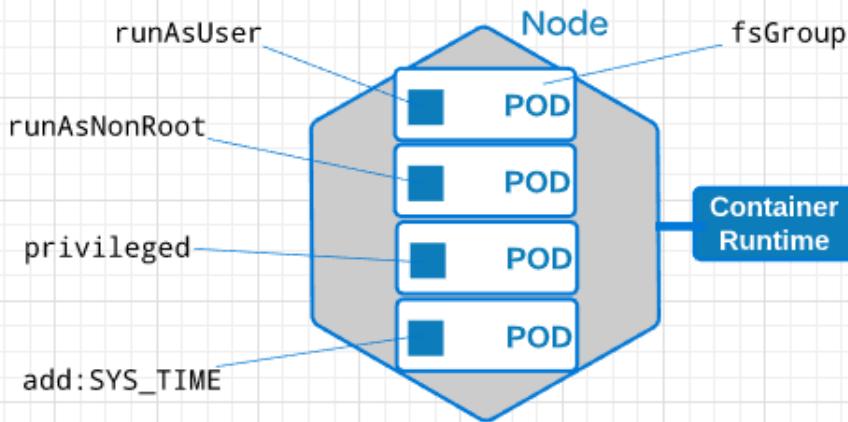
Con esto último ya tendríamos nuestro archivo server.crt listo para poder usarlo



Capítulo 67. Contexto de seguridad de ejecución

Container and Pod Access

You can limit the access to certain objects at the pod and container level. This will allow your images to remain stable.



La definición de contextos de seguridad, nos va a permitir bloquear en cierto modo los contenedores, de modo que sólo ciertos procesos van a poder realizar ciertas cosas.

Esto nos va a asegurar la estabilidad de los contenedores y nos va a permitir otorgar o quitar privilegios.

Los contexto de seguridad pueden ser asignados a nivel de Pod, de forma que todos los contenedores sujetos al Pod comparten estas políticas.

Capítulo 68. Lab: Contexto de seguridad de ejecución

68.1. Comprobando contexto de seguridad

Vamos a ejecutar un contenedor para observar el contexto de seguridad por defecto

Ejecutamos la siguiente instrucción:

```
$ kubectl run pod-with-default --image alpine --restart Never -- /bin/sleep 999999  
pod/pod-with-default created
```

Obtenemos el userId y el groupId de ejecución del contenedor:

```
$ kubectl exec pod-with-default id  
uid=0(root) gid=0(root)  
groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel),11(floppy),20(dialout),26(tape),27(video)
```

- El userId es 0, corresponde con el usuario root
- El groupId también es 0, que corresponde también con el usuario root

El contenedor obtiene esta información del archivo Dockerfile de donde se creó la imagen Docker, cuando se crea el contenedor, se otorga ese usuario en concreto por defecto (salvo que el archivo Dockerfile diga lo contrario mediante la directiva USER de Docker)

Podemos echarle un vistazo al archivo del Dockerfile de la imagen alpine en la siguiente URL:
<https://github.com/alpinelinux/docker-alpine/blob/master/Dockerfile>

No observamos directiva USER

Supongamos entonces que no queremos ejecutar el contenedor como root, sino, como otro usuario, ¿Cómo hacemos esto si en el archivo Dockerfile no se especificó?

68.2. Arrancando Pods con otro contexto de usuario

Cuando arrancamos un Pod, podemos especificar el contexto de seguridad en el propio archivo YAML

Vamos a crear el archivo **pod-with-custom-user-context.yml** y le agregamos el siguiente contenido:

```

apiVersion: v1
kind: Pod
metadata:
  name: alpine-user-context
spec:
  containers:
  - name: main
    image: alpine
    command: ["/bin/sleep", "999999"]
    securityContext:
      runAsUser: 405

```

- Observamos la entrada del securityContext, la cual posibilita customizar el contexto del arranque del contenedor, el usuario interno con el que se ejecutará el mismo
- Arrancamos el contenedor como usuario 405, que corresponde en contexto de seguridad al del usuario invitado
 - Indicamos en formato octal
 - Correspondría al permiso linux sólo lectura **-r----****r-x**

Ejecutamos la creación del Pod:

```

$ kubectl apply -f pod-with-custom-user-context.yml

pod/alpine-user-context created

```

Ahora, vamos a comprobar el contexto de usuario del contenedor, ejecutando el siguiente comando:

```

$ kubectl exec alpine-user-context id

uid=405(guest) gid=100(users)

```

- Observamos que el uid ahora ha cambiado
- El gid también es diferente al caso de por defecto

68.3. Arrancando Pod con contexto de no-root

Ahora, vamos a arrancar otro Pod y le vamos a indicar a kubernetes mediante manifiesto YAML que el contexto de seguridad del usuario sea no root de forma explícita, sin indicar un código de usuario específico

Vamos a crear el archivo **pod-with-non-root-context.yml** y le agregamos el siguiente contenido:

```

apiVersion: v1
kind: Pod
metadata:
  name: alpine-nonroot
spec:
  containers:
  - name: main
    image: alpine
    command: ["/bin/sleep", "999999"]
  securityContext:
    runAsNonRoot: true

```

Ejecutamos la creación del Pod:

```
$ kubectl apply -f pod-with-non-root-context.yml
pod/alpine-nonroot created
```

Obtenemos información de los Pods que se encuentran en operación:

```
$ kubectl get pods
NAME           READY   STATUS            RESTARTS   AGE
alpine-nonroot  0/1    CreateContainerConfigError  0          70s
alpine-user-context 1/1    Running          0          6m59s
pod-with-default 1/1    Running          0          21m
```

- Observamos que el Pod con nombre alpine-nonroot parece que tiene algún problema

Vamos a realizarle una operación describe, a ver si obtenemos más detalles sobre lo que puede estar sucediendo:

```
$ kubectl describe pod alpine-nonroot
...
Events:
Type Reason Age From Message
Normal Scheduled <unknown> default-scheduler Successfully assigned default/alpine-nonroot to kubeminion2
Normal Pulled 92s (x8 over 3m2s) kubelet, kubeminion2 Successfully pulled image "alpine"
Warning Failed 92s (x8 over 3m2s) kubelet, kubeminion2 Error: container has runAsNonRoot and image will run as root
Normal Pulling 81s (x9 over 3m5s) kubelet, kubeminion2 Pulling image "alpine"
```

- Nos fijamos en la parte de abajo, en los Events
- El error es que el contenedor ha intentado ejecutarse como un usuario distinto de root, pero por construcción, estaba destinado a ejecutarse como root

- Estamos intentado ejecutar como no-root un contenedor creado de una imagen como (USER root en directiva del Dockerfile)
- Para corregir esto, deberíamos de ir al Dockerfile y corregir esto, indicando la directiva USER con un usuario que fuera diferente de root, para que esto funcionara

A continuación, vamos a hacer una edición del archivo **pod-with-non-root-context.yml** y vamos a dejarlo tal que así:

```
apiVersion: v1
kind: Pod
metadata:
  name: alpine-nonroot
spec:
  containers:
    - name: main
      image: alpine
      command: ["/bin/sleep", "999999"]
      securityContext:
        runAsUser: 405
        runAsNonRoot: true
```

- Hemos añadido la directiva **runAsUser: 405**

Eliminamos el Pod:

```
$ kubectl delete pod alpine-nonroot
pod "alpine-nonroot" deleted
```

Volvemos a ejecutar la recreación del Pod después de haber actualizado la definición YAML:

```
$ kubectl apply -f pod-with-non-root-context.yml
pod/alpine-nonroot created
```

Consultamos de nuevo los Pods para ver si ahora si ha podido arrancar de forma correcta:

```
$ kubectl get pods
NAME           READY   STATUS    RESTARTS   AGE
alpine-nonroot  1/1     Running   0          12s
alpine-user-context 1/1     Running   0          17m
pod-with-default 1/1     Running   0          31m
```

68.4. Arrancado Pod con contexto Privileged

En ocasiones, puede darse el caso de que necesitemos utilizar acceso a funciones de alto privilegio del nodo, y puede venirnos bien el uso del contexto en modo privilegiado

Vamos a crear un archivo con nombre **pod-with-privileged-context.yml** y le agregamos el siguiente contenido:

```
apiVersion: v1
kind: Pod
metadata:
  name: privileged-pod
spec:
  containers:
  - name: main
    image: alpine
    command: ["/bin/sleep", "999999"]
    securityContext:
      privileged: true
```

- Indicamos en el contexto de seguridad, privileged: true

Ejecutamos la creación del Pod:

```
$ kubectl apply -f pod-with-privileged-context.yml
pod/privileged-pod created
```

Vamos primero a observar los dispositivos del contexto del primer Pod que hemos creado **pod-with-default**:

```
$ kubectl exec -it pod-with-default ls /dev
core      null      shm      termination-log
fd        ptmx      stderr   tty
full      pts       stdin    urandom
mqueue    random   stdout   zero
```

- Observamos una visión de dispositivos en modo "SandBox" encapsulada, no vemos realmente todos los dispositivos que el nodo tiene donde se ejecuta el Pod

Y comparamos esa vista, con los dispositivos que observamos del Pod **privileged-pod**:

```
$ kubectl exec -it privileged-pod ls /dev
autofs    stdout    tty50
```

```

bsg          termination-log    tty51
btrfs-control  tty        tty52
bus          tty0        tty53
core         tty1        tty54
cpu          tty10       tty55
cpu_dma_latency  tty11       tty56
crash        tty12       tty57
dm-0          tty13       tty58
dm-1          tty14       tty59
dm-2          tty15       tty6
dri          tty16       tty60
fb0          tty17       tty61
fd           tty18       tty62
full         tty19       tty63
fuse          tty2        tty7
hpet         tty20       tty8
hwrng        tty21       tty9
input         tty22       ttyS0
kmsg          tty23       ttyS1
loop-control   tty24       ttyS2
mapper        tty25       ttyS3
mcelog        tty26       uhid
mem           tty27       uinput
mqueue        tty28       urandom
net            tty29       usbmon0
network_latency  tty3        usbmon1
network_throughput  tty30      vboxguest
null          tty31       vboxuser
nvram         tty32       vcs
oldmem        tty33       vcs1
port          tty34       vcs2
ppp           tty35       vcs3
ptmx         tty36       vcs4
pts           tty37       vcs5
random        tty38       vcs6
raw            tty39       vcsa
rtc0          tty4        vcsa1
sda           tty40       vcsa2
sda1          tty41       vcsa3
sda2          tty42       vcsa4
sg0           tty43       vcsa5
sg1           tty44       vcsa6
shm           tty45       vfio
snapshot       tty46      vga_arbiter
snd            tty47       vhci
sr0           tty48       vhost-net
stderr        tty49       zero
stdin         tty5        
```



- Observamos que tenemos acceso a una visión completa de los dispositivos que el nodo puede

utilizar

Ahora, vamos a intentar cambiar la hora del Pod por defecto **pod-with-default**:

```
$ kubectl exec -it pod-with-default -- date +%T -s "12:00:00"  
date: can't set date: Operation not permitted  
12:00:00
```

- Observamos que esta operación no nos es permitida

Otra opción, si no queremos ir "a lo bestia" :D, en lugar de arrancar un contenedor en modo privileged=true, podríamos añadir una capability al contenedor que le permitiera realizar lo que necesita y sólo eso.



68.5. Arrancando un Pod añadiéndole capabilities

Vamos a crear un archivo con nombre **pod-add-capability.yml** y le agregamos el siguiente contenido:

```
apiVersion: v1  
kind: Pod  
metadata:  
  name: kernelchange-pod  
spec:  
  containers:  
    - name: main  
      image: alpine  
      command: ["/bin/sleep", "999999"]  
  securityContext:  
    capabilities:  
      add:  
        - SYS_TIME
```

- La capability que nos interesa se llama **SYS_TIME**
- Podemos encontrar información sobre las capabilites en la documentación oficial de Docker

Ejecutamos la creación del Pod:

```
$ kubectl apply -f pod-add-capability.yml  
pod/kernelchange-pod created
```

Intentamos ahora, cambiarle la hora al contenedor del Pod:



```
$ kubectl exec -it kernelchange-pod -- date +%T -s "12:00:00"
```

```
12:00:00
```

Observamos que ahora si se ha podido realizar la operación:

```
$ kubectl exec -it kernelchange-pod -- date
```

```
Sat Jan 11 12:00:02 UTC 2020
```

68.6. Arrancando un Pod eliminándole capabilities

Otro caso que se nos puede dar, es que queramos arrancar un contenedor y deseemos eliminar capabilities de las que el propio contenedor trae por defecto otorgadas

Podemos encontrar información sobre las capabilities que un contenedor Docker trae por defecto en la propia documentación oficial de Docker

Vamos a crear el archivo **pod-remove-capability.yml** y le agregamos el siguiente contenido:

```
apiVersion: v1
kind: Pod
metadata:
  name: remove-capabilities
spec:
  containers:
    - name: main
      image: alpine
      command: ["/bin/sleep", "999999"]
  securityContext:
    capabilities:
      drop:
        - CHOWN
```

- La capability CHOWN interviene en el privilegio de poder obtener el propietario de archivos y directorios

Ejecutamos la creación del Pod:

```
$ kubectl apply -f pod-remove-capability.yml
```

```
pod/remove-capabilities created
```

Intentamos ahora lanzar un comando chown dentro del contenedor, para ver quien es el propietario de la carpeta interna /tmp:

```
$ kubectl exec remove-capabilities chown guest /tmp
```

```
chown: /tmp: Operation not permitted  
command terminated with exit code 1
```

- Observamos que la operación no la podemos llevar a cabo por falta de privilegios

68.7. Arrancando un Pod con contexto de seguridad de volumen, sólo lectura

Otra opción interesante de configuración que puede interesarnos, es arrancar un Pod con volúmenes con operación de sólo lectura, de manera que prevenimos por seguridad que el contenedor pueda llevar a cabo cualquier tipo de operación de escritura

Vamos a crear el archivo **pod-with-readonly-filesystem.yml** y le agregamos el siguiente contenido:

```
apiVersion: v1  
kind: Pod  
metadata:  
  name: readonly-pod  
spec:  
  containers:  
    - name: main  
      image: alpine  
      command: ["/bin/sleep", "999999"]  
  securityContext:  
    readOnlyRootFilesystem: true  
  volumeMounts:  
    - name: my-volume  
      mountPath: /volume  
      readOnly: false  
  volumes:  
    - name: my-volume  
      emptyDir:
```

Ejecutamos la creación del pod:

```
$ kubectl apply -f pod-with-readonly-filesystem.yml  
pod/readonly-pod created
```

Ahora, vamos a crear un nuevo archivo en la ruta /new-file

```
$ kubectl exec -it readonly-pod touch /new-file
```

```
touch: /new-file: Read-only file system  
command terminated with exit code 1
```

Efectivamente, el sistema indica un error por que el contexto de seguridad del contenedor es de sólo lectura

A continuación, vamos a intentar crear un archivo dentro de la carpeta del volumen:

```
$ kubectl exec -it readonly-pod touch /volume/new-file
```

- En este caso, sí que podemos hacerlo y la consola no muestra error

Listamos el archivo del volumen para comprobarlo:

```
$ kubectl exec -it readonly-pod ls /volume
```

```
new-file
```

- Observamos el archivo dentro del volumen
- De esta forma tendríamos un contexto de seguridad de funcionamiento a nivel de estructura de archivos del contenedor de sólo lectura
- Sólo permitimos las escrituras controladas dentro de los volúmenes específicos montados

68.8. Arrancando un Pod con contexto de seguridad a nivel de Pod

Este contexto de seguridad es parecido a establecerlo a nivel de contenedor, la diferencia es que en lugar de establecerlo a nivel del YAML en el contenedor, lo establecemos a nivel del Pod

Vamos a crear el archivo **pod-level-security-context.yml** y le agregamos el siguiente contenido:

```

apiVersion: v1
kind: Pod
metadata:
  name: group-context
spec:
  securityContext:
    fsGroup: 555
    supplementalGroups: [666, 777]
  containers:
    - name: first
      image: alpine
      command: ["/bin/sleep", "999999"]
      securityContext:
        runAsUser: 1111
    volumeMounts:
      - name: shared-volume
        mountPath: /volume
        readOnly: false
    - name: second
      image: alpine
      command: ["/bin/sleep", "999999"]
      securityContext:
        runAsUser: 2222
    volumeMounts:
      - name: shared-volume
        mountPath: /volume
        readOnly: false
  volumes:
    - name: shared-volume
      emptyDir:

```

- Observamos un contexto de seguridad a nivel del Pod
- También observamos varios contenedores y varios contextos de seguridad a cada Pod

Ejecutamos la creación del Pod:

```
$ kubectl apply -f pod-level-security-context.yml
pod/group-context created
```

Por último, abrimos una terminal de conexión con el contenedor cuyo nombre es **first**:

```
$ kubectl exec -it group-context -c first sh
/ $
```

Comprobamos el Id del usuario:

```
/ $ id
```

```
uid=1111 gid=0(root) groups=555,666,777
```

Creamos un archivo dentro del volumen:

```
/ $ echo file > /volume/file
```

Listamos los archivos dentro del volumen:

```
/ $ ls -l /volume
```

```
total 4  
-rw-r--r-- 1 1111 555 5 Jan 11 00:59 file
```

- Observamos que hay diferencia en cuanto al grupo de usuario que ha creado el archivo, respecto al grupo del usuario del Pod

Creamos un archivo en la carpeta /tmp:

```
/ $ echo file > /tmp/file
```

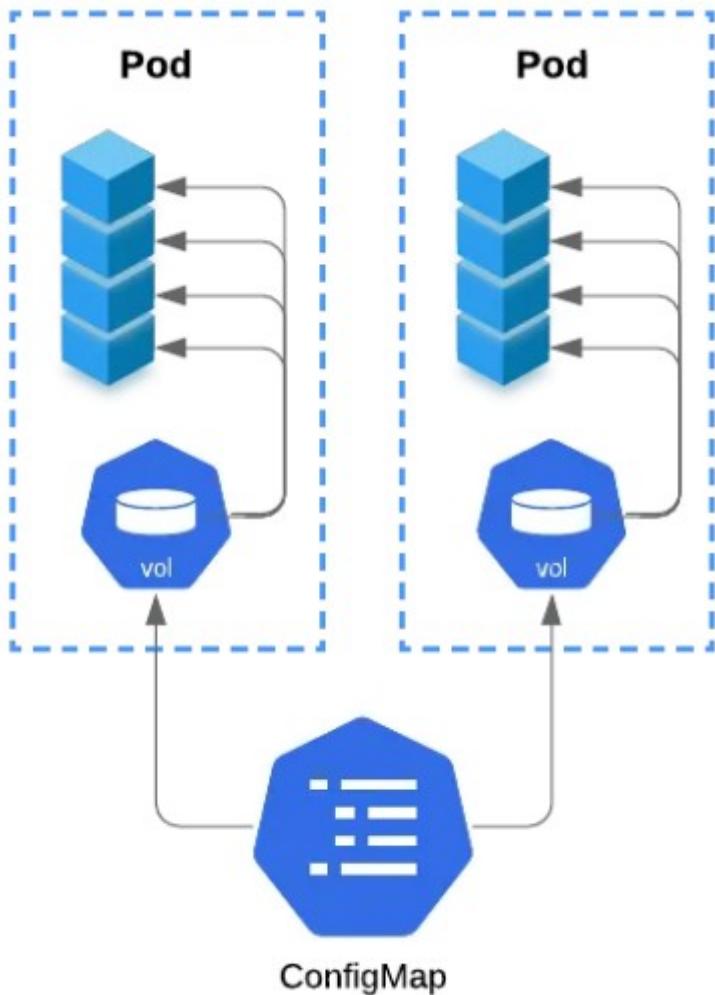
Listamos el archivo de la carpeta /tmp:

```
/ $ ls -l /tmp
```

```
total 4  
-rw-r--r-- 1 1111 root 5 Jan 11 01:03 file
```

- Observamos que si el archivo se crea en la carpeta /tmp el grupo del usuario cambia, y en este caso es root

Capítulo 69. ConfigMap



El objeto de kubernetes ConfigMap, nos permite llevar a cabo un desacople de configuración respecto al contenido de la imagen Docker.

El objetivo es hacer las aplicaciones lo más portables posibles.

Por ejemplo, una aplicación, la misma, apunta en un momento determinado al entorno de pre, entorno sqa, entorno pro, sin cambiar el código o el aplicativo en sí, únicamente pasando una configuración de funcionamiento diferente.

Capítulo 70. Lab: ConfigMap

Mediante este laboratorio, pondremos en práctica el uso del configMap en combinación con el objeto secret

El objetivo va a ser poner en marcha un servidor web nginx, emitiendo tráfico en un puerto seguro (443) con un certificado autogenerado.

70.1. Generando la clave privada

Lo primero que vamos a hacer, es autogenerar una clave privada para el servidor:

```
$ openssl genrsa -out https.key 2048
Generating RSA private key, 2048 bit long modulus
.....+++
.....+++
e is 65537 (0x10001)
```

A continuación, vamos a generar el certificado para el servidor, haciendo uso de la clave privada que hemos generado:

```
$ openssl req -new -x509 -key https.key -out https.cert -days 3650 -subj /CN=www.example.com
```

70.2. Creando el secreto

Ahora, vamos a crear un archivo para crear posteriormente el secreto:

```
$ touch file
```

Creamos el secreto en sí:

```
$ kubectl create secret generic example-https --from-file=https.key --from-file=https.cert --from-file=file
secret/example-https created
```

- El secreto lleva 3 archivos de configuración
 - El contenido de 1 archivo con la clave privada **https.key**
 - El contenido de 1 archivo con el certificado **https.cert**
 - El contenido de nuestro propio archivo **file**

Ahora, vamos a observar el YAML del secreto que hemos generado:



Fondos Europeos



MINISTERIO
DE EDUCACIÓN,
FORMACIÓN PROFESIONAL
Y DEPORTE

Gobierno de España
Cofinanciado por
la Unión Europea



```
$ kubectl get secrets example-https -o yaml
```

```
apiVersion: v1
data:
  file: ""
  https.cert: LS0tLS1CRUdJTiBDRVJUSU
...
...
```

- Deberíamos de observar las 3 entradas que hemos comentado a nivel interno

70.3. Creando el ConfigMap

Ahora, ha llegado el momento de crear el objeto ConfigMap, con la configuración que tendrá el propio nginx y algún que otro dato más

Vamos a crear el archivo **config-map-for-nginx.yml** y le agregamos el siguiente contenido:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: config
data:
  my-nginx-config.conf: |
    server {
      listen      80;
      listen      443 ssl;
      server_name www.example.com;
      ssl_certificate certs/https.cert;
      ssl_certificate_key certs/https.key;
      ssl_protocols   TLSv1 TLSv1.1 TLSv1.2;
      ssl_ciphers     HIGH:!aNULL:!MD5;

      location / {
        root /usr/share/nginx/html;
        index index.html index.htm;
      }
    }
  sleep-interval: |
    25
```

- Mediante una key, especificamos configuración del propio nginx
- Mediante otra key especificamos un tiempo de espera para utilizarlo después

Ejecutamos la creación del ConfigMap:

```
$ kubectl apply -f config-map-for-nginx.yml
```

```
configmap/config created
```

70.4. Creando el Pod que usa el ConfigMap y el Secret

Por último, vamos a crear un Pod que usa tanto el ConfigMap como el Secret

Creamos un archivo que se llame **pod-using-config-map-and-secret.yml** y le agregamos el siguiente contenido:



```

apiVersion: v1
kind: Pod
metadata:
  name: example-https
spec:
  containers:
    - image: linuxacademycontent/fortune
      name: html-web
      env:
        - name: INTERVAL
          valueFrom:
            configMapKeyRef:
              name: config
              key: sleep-interval
      volumeMounts:
        - name: html
          mountPath: /var/htdocs
    - image: nginx:alpine
      name: web-server
      volumeMounts:
        - name: html
          mountPath: /usr/share/nginx/html
          readOnly: true
        - name: config
          mountPath: /etc/nginx/conf.d
          readOnly: true
        - name: certs
          mountPath: /etc/nginx/certs/
          readOnly: true
      ports:
        - containerPort: 80
        - containerPort: 443
  volumes:
    - name: html
      emptyDir: {}
    - name: config
      configMap:
        name: config
        items:
          - key: my-nginx-config.conf
            path: https.conf
    - name: certs
      secret:
        secretName: example-https

```

- Tenemos un primer contenedor que mediante variables de entorno, se le suministra una KEY que se llama INTERVAL, el valor del intervalo, lo cargamos mediante el objeto configMap
- El segundo contenedor, es un servidor web nginx, monta un directorio de configuración llamado **config**, ese directorio de configuración, está cargado mediante un configMap,

utilizando el contenido de la clave **my-nginx-config.conf**

Ejecutamos la creación del Pod:

```
$ kubectl apply -f pod-using-config-map-and-secret.yml  
pod/example-https created
```

A continuación, vamos a describir el ConfigMap que hemos creado:

```
$ kubectl describe configmap config  
  
Name: config  
Namespace: default  
Labels: <none>  
Annotations: kubectl.kubernetes.io/last-applied-configuration:  
 {"apiVersion":"v1","data":{"my-nginx-config.conf":"server {\n    listen 80;\n    listen 443 ssl;\n    server_name www.example.com;\n    ssl_certificate certs/https.cert;\n    ssl_certificate_key certs/https.key;\n    ssl_protocols TLSv1 TLSv1.1 TLSv1.2;\n    ssl_ciphers HIGH:!aNULL:!MD5;\n  
    location / {\n        root /usr/share/nginx/html;\n        index index.html index.htm;\n    }\n}  
sleep-interval:  
25  
Events: <none>
```

Ahora, vamos a observar el montado del volumen **certs** para ver si aparece dentro del contenedor y presente únicamente en el sistema de archivo en memoria tmpfs:

```
$ kubectl exec example-https -c web-server -- mount | grep certs  
tmpfs on /etc/nginx/certs type tmpfs (ro,relatime)
```

Por último, vamos a abrir un forwarding al puerto 443, para acceder directamente al servidor web

y comprobar que accedemos al nginx de forma cifrada:

```
$ kubectl port-forward example-https 8443:443
```

```
Forwarding from 127.0.0.1:8443 -> 443
```

```
Forwarding from [::1]:8443 -> 443
```

Ejecutamos una petición contra el puerto que estamos con el forwarding:

```
$ curl https://localhost:8443 -k
```

Q: What's tiny and yellow and very, very, dangerous?

A: A canary with the super-user password.

- Cada vez que invoquemos la petición, se mostrará frases aleatorias :)



Capítulo 71. Service Mesh

El concepto de malla de servicios (Service Mesh), es una práctica de arquitectura para administrar y visualizar conjuntos de múltiples microservicios basados en contenedores.

71.1. ¿Por qué es conveniente la malla de servicios para la arquitectura de microservicios?

Los microservicios requieren un conjunto de funcionalidades comunes como:

- Autenticación
- Políticas de seguridad
- Protección contra intrusos
- Ataques de denegación de servicio (DDoS)
- Balanceo de carga
- Enrutamiento
- Monitoreo
- Gestión ante fallas.
- Etc.

En una aplicación monolítica, implementamos estos requerimientos una sola vez, pero en entornos con decenas o cientos de contenedores no es práctico repetirlo en cada uno de los sistemas a desplegar.

Si las funcionales que hemos comentado, deben implementarse y repetirse en múltiples microservicios, tenemos la carga adicional de hacerlo en diferentes lenguajes de programación (si es el caso) y por diferentes equipos.

Debido a esta necesidad, surge la necesidad de una arquitectura más efectiva.

Una solución a este problema, es la creación de una malla de servicios o también llamado **Service Mesh** para ofrecer **servicios integrados dentro del clúster**.

Los diferentes niveles de servicios que se mantienen a través de los entornos y que contienen aplicaciones y microservicios pueden ser aprovechados según sea necesario.

71.2. ¿Qué es una malla de servicio - Service Mesh?

En términos generales, **un service mesh puede ser considerado como una infraestructura de software dedicada a manejar la comunicación entre microservicios**.

Proporciona y permite aplicaciones basadas en contenedores y microservicios, los cuales se integran directamente desde el interior del clúster.

Un service mesh proporciona servicios como:

- Vigilancia
- Escalabilidad
- Alta disponibilidad

Los servicios citados, se pueden proporcionar a través de una API de servicios.



En lugar de obligar a su implementación en cada sistema.

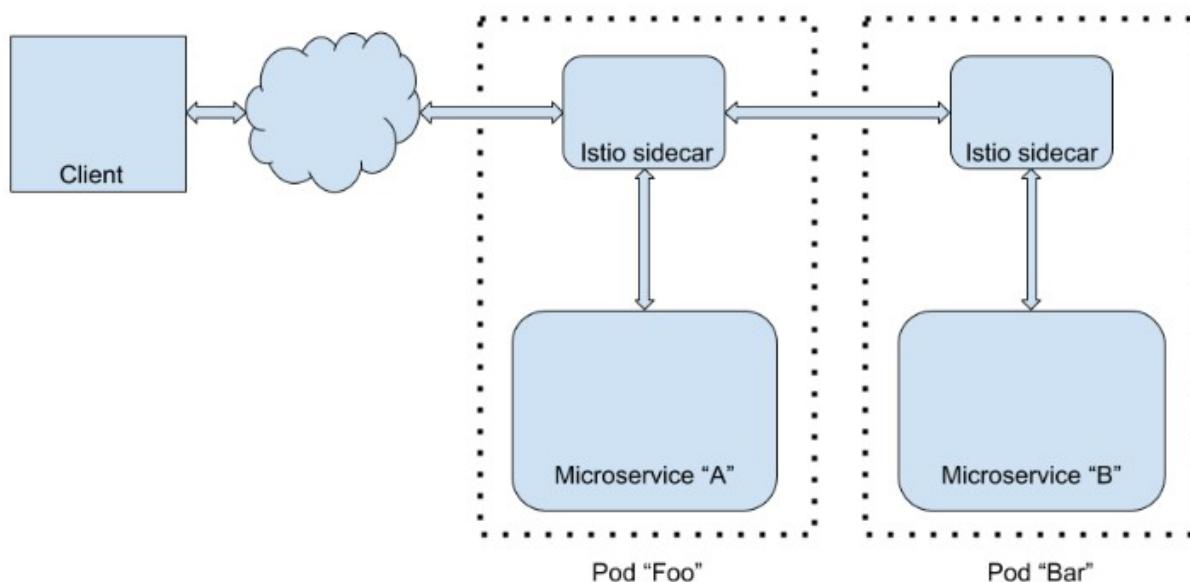
El beneficio consiste en **reducir la complejidad asociada con las aplicaciones modernas de microservicios.**

¿Cómo hace una malla de servicios para abstraer esta complejidad en cada microservicio?... ¡Con ayuda del patrón de arquitectura conocido como **sidecar**!

Con un patrón sidecar, se crea un **pequeño contenedor especial** que se ejecuta lado de cada microservicio.

El nombre viene de los sidecar, las pequeñas cápsulas o sillas que se integran al lado de las motocicletas.

Sidecar no es el único patrón de arquitectura, existen diferentes patrones de arquitectura de microservicios, podemos encontrar información adicional en la siguiente URL: <https://kubernetes.io/blog/2015/06/the-distributed-system-toolkit-patterns/>



El contenedor sidecar funciona como proxy, implementa las funcionalidades comunes como:

- Proxy
- Autenticación
- Monitoreo
- Etc.

Dejando de esta manera los microservicios libres, para enfocarse en su funcionalidad específica.

Un controlador central (control plane) organiza las conexiones, dirige el flujo de tráfico entre los proxies y el plano de control, recolectando las métricas de rendimiento.

71.3. ¿Cómo funciona una malla de servicio - Service Mesh?

Existen diferentes formas de implementar la malla de servicios o service mesh.

Principalmente depende de donde existe la funcionalidad compartida entre los microservicios.

- **Proxy de servicio por nodo**

- Cada nodo en el clúster tiene su propio contenedor de servicios proxy.

- **Proxy de servicio por aplicación**

- Cada aplicación tiene su propio servicio proxy y su acceso a instancias de la aplicación del mismo.

- **Dispositivos discretos**

- Un conjunto de dispositivos discretos re-enrutan el tráfico de encadenamiento de servicio.
- Esto sucede a través de una configuración manual o con un conjunto de adaptadores y plugins necesarios para la automatización de servicios.

- **Cada instancia de la aplicación proxy de servicio**

- Cada una de las instancias tiene su propio "sidecar" proxy.

71.4. Principales ventajas de un Service Mesh

- Permite la posibilidad de que empresas pequeñas puedan crear funciones y arquitecturas altamente escalables.
- Acelera el desarrollo, prueba y despliegue de las aplicaciones.
- Las aplicaciones se actualizan de manera más rápida y eficiente
- Una capa ágil de datos no agregados de proxies situados junto al contenedor clúster puede resultar muy útil y eficaz a la hora de la gestión de servicios de red.
- Mayor libertad en la creación de aplicaciones innovadoras con entornos basados en contenedores
- Conjunto de servicios de infraestructura necesarios para toda aplicación: equilibrio de carga, gestión de tráfico, enrutamiento, vigilancia, control de la aplicación, características de configuración y seguridad, etc.

71.5. Principales desventajas de un Service Mesh

- **Mayor complejidad**

- El uso de un service mesh va a aumentar la complejidad general de la solución, al agregar

componentes para coordinar la malla de servicios, en los control plane y en cada Pod o servidor, de acuerdo al esquema de la solución implementada.

- **Tecnología aún en sus inicios: plataformas como Istio, Envoy y Bouyant**

- Todavía están en sus primeros años de desarrollo, aunque ofrecen funcionalidades usadas en producción en diferentes empresas.

Como recomendación:

- Explorar técnicas como despliegues tipo canary
 - Simultáneamente conviene varias versiones del sistema, el tráfico es redirigido en un % hacia la antigua o nueva versión
- Actualizaciones progresivas como rolling updates
- Monitorización usando APM (Application Performance Monitoring)
- Logs o rastreo distribuidos.
- Aprovechar adicionalmente funcionalidades de auto-escalado y verificación de estabilidad (health check) de plataformas como Kubernetes - OpenShift.
 - Disponen de mecanismos como el (readyness - liveness) donde permiten especificar a cada Pod, las reglas de checking para saber si están los sistemas operando o hay que tomar alguna medida para reiniciarlos.



Capítulo 72. Linkerd



linkerd

Linkerd es un sistema del tipo malla de servicio (service mesh) para kubernetes y otros orquestadores.

Hace que la ejecución de los servicios sea más fácil y segura al brindar:

- Características de depuración en tiempo de ejecución
- Observabilidad
- Confiabilidad
- Seguridad
- Etc.

Todo... Sin requerir ningún tipo de cambio en el código de los sistemas que se encuentren ya operando

Linkerd es totalmente OpenSource, funciona bajo licencia Apache v2 y es un proyecto que está bajo el amparo de la fundación Cloud Native Computing Foundation.

El Cloud Native es un patrón de arquitectura de software para desarrollar aplicaciones usando principios esenciales de cloud computing, como:



- La escalabilidad
- Elasticidad
- Agilidad

El código fuente del proyecto Linkerd se encuentra disponible en el propio GitHub:
<https://github.com/linkerd>

72.1. ¿Cómo funciona?

Linkerd, posee tres componentes básicos:

- UI
- Data Plane
- Control Plane

Funciona instalando un conjunto de servidores proxy ultraligeros y transparentes junto a cada instancia de servicio.

Estos servidores proxy manejan automáticamente todo el tráfico hacia y desde el servicio.

Debido a que son transparentes, estos proxies actúan como pilas de red fuera de proceso, altamente instrumentadas, que envían telemetría y reciben señales de control desde el control plane.

Este diseño, permite que Linkerd mida y manipule el tráfico hacia y desde su servicio sin introducir una latencia excesiva.



Capítulo 73. Lab: Linkerd

Mediante este laboratorio, vamos a poner en marcha la plataforma Linkerd dentro de nuestro clúster de Kubernetes.

73.1. Verificar la versión de kubernetes

Para poder utilizar Linkerd, lo primero que necesitamos es conocer la versión que tenemos operando de kubernetes, mínimo, vamos a necesitar disponer de una versión de kubernetes 1.13 o superior.

Ejecutamos el siguiente comando:

```
$ kubectl version --short
```

```
Client Version: v1.17.4  
Server Version: v1.17.4
```

73.2. Instalación del cliente de consola

Para gestionar Linkerd, vamos a proceder con la instalación del cliente de consola, ejecutando la siguiente instrucción:

```
$ curl -sL https://run.linkerd.io/install | sh
```

```
...  
Download complete!
```

```
Validating checksum...  
Checksum valid.
```

```
Linkerd stable-2.7.0 was successfully installed !
```

Add the linkerd CLI to your path with:

```
export PATH=$PATH:/home/kubernetes/.linkerd2/bin
```

Now run:

```
linkerd check --pre          # validate that Linkerd can be installed  
linkerd install | kubectl apply -f - # install the control plane into the 'linkerd' namespace  
linkerd check                # validate everything worked!  
linkerd dashboard            # launch the dashboard
```

Looking for more? Visit <https://linkerd.io/2/next-steps>

73.3. Añadir binario de ejecución al path

A continuación, vamos a añadir el binario de ejecución de Linkerd a nuestro path del sistema.

Editamos el archivo que se encuentra en la siguiente ruta:

```
/home/kubernetes/.bashrc
```

Y añadimos lo siguiente al final del archivo:

```
export PATH=$PATH:$HOME/.linkerd2/bin
```

Guardamos el archivo finalmente

73.4. Comprobando versión instalada

Seguidamente, ejecutamos la siguiente instrucción en consola, para comprobar que efectivamente tenemos instalado el cliente:

```
$ linkerd version  
Client version: stable-2.7.0  
Server version: unavailable
```

73.5. Validando el clúster de kubernetes

Kubernetes puede ser instalado de varias formas, y dependiendo de la variante del sistema operativo incluir unos u otros procedimientos.

Para asegurarnos de que el control-plane de kubernetes está instalado y operando de forma correcta, el cliente de Linkerd puede chequear y validar que está todo configurado correctamente.

Ejecutamos el siguiente comando:

```
$ linkerd check --pre

kubernetes-api
-----
✓ can initialize the client
✓ can query the Kubernetes API

kubernetes-version
-----
✓ is running the minimum Kubernetes API version
✓ is running the minimum kubectl version

pre-kubernetes-setup
-----
✓ control plane namespace does not already exist
✓ can create non-namespaced resources
✓ can create ServiceAccounts
✓ can create Services
✓ can create Deployments
✓ can create CronJobs
✓ can create ConfigMaps
✓ can create Secrets
✓ can read Secrets
✗ no clock skew detected
    clock skew detected for node(s): kubernetes-master.local, kubernetes-minion1.local, kubernetes-minion2.local
    see https://linkerd.io/checks/#pre-k8s-clock-skew for hints
```

```
pre-kubernetes-capability
-----
```

```
✓ has NET_ADMIN capability
✓ has NET_RAW capability
```

```
linkerd-version
-----
```

```
✓ can determine the latest version
✓ cli is up-to-date
```

Status check results are ✗

- Observamos en nuestro caso, que nos alerta sobre una configuración que no es correcta, se trata de la hora del sistema, nuestros servidores no tienen la zona horaria correcta, en nuestro caso, Europe/Madrid
- Si accedemos a la URL que nos indica Linkerd, nos informa que esto puede ser una causa potencial de fallos, por lo que procederemos a ajustar en los 3 servidores la zona horaria correcta.

73.6. Ajustando la zona horaria en el sistema

En primer lugar, podemos comprobar la información sobre la hora/fecha que nos comunica el agente kubelet acerca de los nodos:

```
$ kubectl describe nodes | grep Ready | grep posting

Ready True Wed, 18 Mar 2020 10:43:59 +0100 Mon, 16 Mar 2020 13:17:34 +0100 KubeletReady
kubelet is posting ready status
Ready True Wed, 18 Mar 2020 10:42:59 +0100 Tue, 17 Mar 2020 09:57:37 +0100 KubeletReady
kubelet is posting ready status
Ready True Wed, 18 Mar 2020 10:45:18 +0100 Wed, 18 Mar 2020 10:24:59 +0100 KubeletReady
kubelet is posting ready status
```

Ejecutamos el siguiente comando en todos los servidores que compongan el clúster de kubernetes para sincronizar la zona horaria:

```
$ sudo timedatectl set-timezone Europe/Madrid
```

A pesar de haber llevado a cabo estos ajustes, actualmente existe un bug en linkerd2 que provoca que siga apareciendo el mensaje de error **no clock skew detected**



<https://github.com/linkerd/linkerd2/issues/3943>

73.7. Instalando Linkerd

Una vez todos los pre-requisitos para llevar a cabo la instalación de linkerd han sido cubiertos, llega el momento de proceder a su instalación.

Linkerd se situará en un namespace propio, con nombre **linkerd**.

Ejecutamos el siguiente comando:

```
$ linker install | kubectl apply -f -
```

El comando de instalación linker, genera un manifiesto de Kubernetes con todos los recursos necesarios para el control-plane.

Al instalar el manifiesto en kubernetes mediante el comando kubectl apply, le estamos indicando a kubernetes que agregue esos recursos al clúster.



Dependiendo de la velocidad de la conexión a internet del clúster, el clúster podría tardar de media 2 minutos de media en extraer las imágenes docker que corresponden a Linkerd.

73.8. Comprobando la instalación de Linkerd

Una vez ejecutado el comando de instalación, podemos lanzar mientras, el comando de verificación donde nos irá informando sobre el proceso de la instalación.

Ejecutamos en consola el siguiente comando:

```
$ linker check

...
linkerd-version
-----
✓ can determine the latest version
✓ cli is up-to-date

control-plane-version
-----
✓ control plane is up-to-date
✓ control plane and cli versions match

Status check results are ✓
```

También, comprobamos que todos los componentes del namespace **linkerd** se encuentran operando de forma correcta:

```
$ kubectl --namespace=linkerd get deploy
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
linkerd-controller	1/1	1	1	6m45s
linkerd-destination	1/1	1	1	6m44s
linkerd-grafana	1/1	1	1	6m39s
linkerd-identity	1/1	1	1	6m46s
linkerd-prometheus	1/1	1	1	6m41s
linkerd-proxy-injector	1/1	1	1	6m38s
linkerd-sp-validator	1/1	1	1	6m35s
linkerd-tap	1/1	1	1	6m33s
linkerd-web	1/1	1	1	6m42s

73.9. Activando el dashboard de control

Para acceder al dashboard de control, vamos a ejecutar el siguiente comando:

```
$ linker dashboard &
```

El comando configura un port-forwarding localmente en el sistema, hacia un Pod con nombre **linkerd-web**



Todos los componentes de linkerd tienen instalado un proxy en sus pods, lo que proporciona la capacidad de monitoreo del propio linkerd con todos sus componentes.

Si accedemos al navegador con la siguiente URL deberíamos de ver el dashboard de control:

<http://localhost:50750/namespaces>

Adicionalmente, podemos monitorear en la consola las peticiones a los diferentes componentes que linkerd está realizando, obteniendo una información bastante detallada:

Ejecutamos en la consola el siguiente comando:

```
$ linkerd -n linkerd top deploy/linkerd-web
```

Source	Destination	Method	Path	Count	Best
Worst					
linkerd-web-78c596c5f4-ns5s6	linkerd-controller-9c6d879f7-9s2j9	POST	/api/v1/StatSummary	26	5ms
2s					
...					



Capítulo 74. Lab: Desplegando una aplicación con Linkerd

Mediante este laboratorio, vamos a poner en marcha el despliegue de una aplicación en la infraestructura que tenemos funcionando con la combinación Kubernetes - Linkerd.

74.1. Instalación de la aplicación

Para ver como funciona realmente Linkerd para uno de sus servicios, vamos a instalar una aplicación de demostración.

La aplicación **emojivoto** es un sistema que va a operar dentro del clúster de kubernetes de forma independiente, utiliza una combinación de llamadas gRPC y HTTP para permitir a los usuarios del sistema, votar los emojis favoritos :)

Ejecutamos el siguiente comando en la consola:

```
$ curl -sL https://run.linkerd.io/emojivoto.yml \
| kubectl apply -f -\n\nnamespace/emojivoto created
serviceaccount/emoji created
serviceaccount/voting created
serviceaccount/web created
service/emoji-svc created
service/voting-svc created
service/web-svc created
deployment.apps/emoji created
deployment.apps/vote-bot created
deployment.apps/voting created
deployment.apps/web created
```

- Entre otros elementos, se ha creado un namespace específico llamado **emojivoto** donde están desplegados todos los componentes que forman parte del sistema

74.2. Ejecutando un proxy de acceso

Para acceder a la aplicación, vamos a ejecutar un proxy para hacer port-forwarding con el servicio que ha creado la aplicación, el nombre del servicio que nos interesa se llama **service/web-svc**

Ejecutamos en la consola el siguiente comando:

```
$ kubectl -n emojivoto port-forward svc/web-svc 8080:80
```

- Estamos realizando un binding hacia localhost, abriendo el puerto 8080 y conectando con el

puerto 80 del servicio internamente

Abrirmos en el navegador la siguiente URL y deberíamos de observar el sistema funcionando:

```
$ http://localhost:8080/
```

A conciencia, la aplicación tiene algunos fallos, por ejemplo si hacemos clic en el emoji del donut, nos dará un error 404.



Esto sucede a conciencia, para posteriormente observar como Linkerd identifica este tipo de problemas

74.3. Aplicando la malla de servicios Linkerd a la aplicación



Fondos Europeos



MINISTERIO
DE EDUCACIÓN, FORMACIÓN PROFESIONAL
Y DEPORTE

Gobierno de España
Cofinanciado por
la Unión Europea



```
$ kubectl get -n emojivoto deploy -o yaml \  
| linkerd inject - \  
| kubectl apply -f -
```

```
deployment "emoji" injected  
deployment "vote-bot" injected  
deployment "voting" injected  
deployment "web" injected
```

```
deployment.apps/emoji configured  
deployment.apps/vote-bot configured  
deployment.apps/voting configured  
deployment.apps/web configured
```

El comando recupera todos los despliegues en funcionamiento del namespace **emojivoto**, ejecuta el manifiesto a través del comando **linkerd inject** y luego reconfigura los despliegues de ese namespace del clúster.



El comando de inyección de linkerd agrega anotaciones a la especificación del pod, indicando a Linkerd que agrege ("inject") el proxy como contenedor adicional a la especificación del pod, para que este sea desplegado de forma conjunta.

Una vez aplicado el service mesh, posiblemente haya que reconectar el proxy de la aplicación emoji, por lo que volvemos a ejecutar el siguiente comando:

```
$ kubectl -n emojivoto port-forward svc/web-svc 8080:80
```

A continuación, vamos a verificar el estado de Linkerd con el namespace **emojivoto**, ejecutamos el siguiente comando:

```
$ linkerd --namespace=emojivoto check --proxy
```

kubernetes-api

-
- ✓ can initialize the client
- ✓ can query the Kubernetes API

kubernetes-version

-
- ✓ is running the minimum Kubernetes API version
- ✓ is running the minimum kubectl version

linkerd-existence

-
- ✓ 'linkerd-config' config map exists
- ✓ heartbeat ServiceAccount exist
- ✓ control plane replica sets are ready
- ✓ no unschedulable pods
- ✓ controller pod is running
- ✓ can initialize the client
- ✓ can query the control plane API

linkerd-config

-
- ✓ control plane Namespace exists
- ✓ control plane ClusterRoles exist
- ✓ control plane ClusterRoleBindings exist
- ✓ control plane ServiceAccounts exist
- ✓ control plane CustomResourceDefinitions exist
- ✓ control plane MutatingWebhookConfigurations exist
- ✓ control plane ValidatingWebhookConfigurations exist
- ✓ control plane PodSecurityPolicies exist

linkerd-identity

-
- ✓ certificate config is valid
- ✓ trust roots are using supported crypto algorithm
- ✓ trust roots are within their validity period
- ✓ trust roots are valid for at least 60 days
- ✓ issuer cert is using supported crypto algorithm
- ✓ issuer cert is within its validity period
- ✓ issuer cert is valid for at least 60 days
- ✓ issuer cert is issued by the trust root

linkerd-identity-data-plane

✓ data plane proxies certificate match CA

linkerd-api

✓ control plane pods are ready

✓ control plane self-check

✓ [kubernetes] control plane can talk to Kubernetes

✓ [prometheus] control plane can talk to Prometheus

✓ tap api service is running

linkerd-version

✓ can determine the latest version

✓ cli is up-to-date

linkerd-data-plane

✓ data plane namespace exists

✓ data plane proxies are ready

✓ data plane proxy metrics are present in Prometheus

✓ data plane is up-to-date

✓ data plane and cli versions match

Status check results are ✓

74.4. Comprobando el funcionamiento

Ahora, vamos a observar el panel de Linkerd, donde podremos ver todos los servicios que está operando.

La aplicación lleva a conciencia un generador de carga para poder observar métricas de tráfico de red en tiempo real.

Ejecutamos el siguiente comando en consola:

```
$ linkerd -n emojivoto stat deploy
```

NAME	MESHED	SUCCESS	RPS	LATENCY_P50	LATENCY_P95	LATENCY_P99	TCP_CONN
emoji	1/1	100.00%	1.8rps	1ms	5ms	5ms	2
vote-bot	1/1	-	-	-	-	-	-
voting	1/1	87.50%	0.9rps	1ms	1ms	2ms	2
web	1/1	93.64%	1.8rps	15ms	55ms	91ms	2

- Se observan las métricas clave de un despliegue:

- Success rate (Tasa de éxito de las peticiones)
- Request rates (El ratio de peticiones/segundo)

- Latency distribution percentiles (Latencia de las peticiones/milisegundos)

También podemos lanzar el comando top para obtener una visión en tiempo real de qué rutas se están llamando:

```
$ linkerd -n emojivoto top deploy
```

- Observaremos entre otras, simulación de peticiones votando emojis

Paralelamente, también podemos utilizar el comando tap, que muestra la secuencia de solicitudes de un pod, deployment o incluso de todo el espacio de nombres **emojivoto**.

Ejecutamos en consola lo siguiente:

```
$ linkerd -n emojivoto tap deploy/web

req id=7:0 proxy=in src=10.44.0.8:41138 dst=10.36.0.7:8080 tls=true :method=GET :authority=web-svc.emojivoto:80
:path=/api/list
req id=7:1 proxy=out src=10.36.0.7:54522 dst=10.44.0.9:8080 tls=true :method=POST :authority=emoji-svc.emojivoto:8080
:path=/emojivoto.v1.EmojiService/ListAll
rsp id=7:1 proxy=out src=10.36.0.7:54522 dst=10.44.0.9:8080 tls=true :status=200 latency=16232µs
end id=7:1 proxy=out src=10.36.0.7:54522 dst=10.44.0.9:8080 tls=true grpc-status=OK duration=151µs response-length=2140B
...
```



Capítulo 75. Monitorización

- Al iniciar los servicios y aplicaciones, necesitamos comprobar si los mismos están funcionando correctamente.
- En esta sección explicaremos como monitorizar los recursos de K8s.

75.1. describe pod

- Por medio de **kubectl get pods** podemos comprobar el estado de cada pod y comprobar si están listos para el servicio
- Sin embargo, para obtener mayor información, podemos usar **kubectl describe pod** para comprobar porqué un contenedor no se inicia, lo que nos da una ligera pista de lo que podría pasar con el contenedor.
- En este comando podemos observar:
 - Información de los contenedores
 - Información del pod como etiquetas, recursos, etc.
 - Estado del contenedor
 - Estado del pod
 - Dependiendo del estado, podemos recibir información adicional
 - Reinicios del pod
 - Para detección de crashloops, si tenemos una política de reinicio de **always**
 - Si ha pasado la prueba de vida
 - Prueba de vida del pod, indicando si está listo para el servicio.
 - Y un log de eventos recientes.



- Es posible que no se desplieguen pods por falta de recursos, lo que indica que debemos reducir el factor de escala.

- La información de descripción se persiste en **etcd**, lo cual permite tener una visión general del estado del cluster

75.2. get events

- Permite mostrar los eventos en base a un namespace.

```
kubectl get events --namespace=mi-espacio
```

75.3. get pod

- Podemos pedir la información de un pod en formato yaml, lo que nos muestra más información que un describe pod

```
kubectl get pod <nombre-pod> -o yaml
```

75.4. status

- Los pods poseen distintos estados de inicio
 - Init:N/M : M contenedores, N iniciados
 - Init:Error : Un contenedor a fallado al iniciar
 - Init:CrashLoopBackOff : Fallo repetitivo de un contenedor de un pod
 - Pending : Pod no inicializado, no ha comenzado a iniciar sus contenedores.
 - PodInitializing|running : Finalizaciómn de ejecución de contenendores.
- Podemos acceder al log de un InitContainer indicando su nombre y el nombre del pod que lo está iniciando

```
kubectl logs <nombre-pod> -c <nombre-contenedor-inicio>
```



Fondos Europeos

MINISTERIO
DE EDUCACIÓN, FORMACIÓN PROFESIONAL
Y DEPORTE

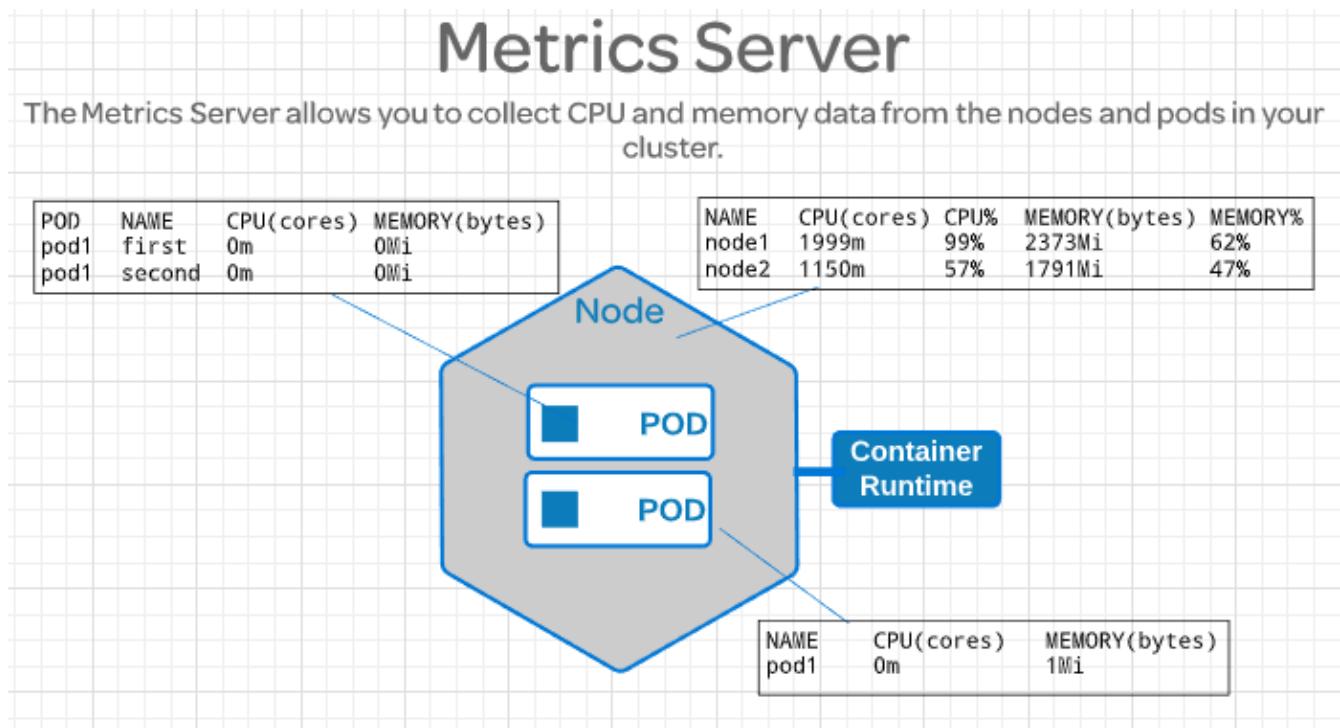
COFINANCIADO POR
LA UNIÓN EUROPEA



Capítulo 76. Lab: Monitorización

En lo que respecta a la monitorización de un clúster de kubernetes, vamos a poder encontrar decenas de herramientas y utilidades que van a hacernos la vida un poco más fácil en cuanto a la labor de supervisión como operadores DevOps

76.1. Instalando el Servidor de métricas



Podemos llevar a cabo la monitorización en cuanto al uso de CPU y memoria de nuestros Pods y Nodos mediante el uso de un servidor de métricas.

Kubernetes provee información detallada sobre el uso de los recursos de las aplicaciones

Llevar a cabo una buena monitorización, va a permitirnos ser más eficientes con nuestra infraestructura y evitar en la medida de lo posible cuellos de botella en el funcionamiento diario de los sistemas

El servidor de métricas, tiene capacidad de descubrimiento, por lo que se mostrarán todos los nodos del clúster y las consultas sobre consumo de CPU / RAM

Lo primero que vamos a hacer, es clonarnos una copia del servidor de métricas que está disponible en GitHub:

```
$ git clone https://github.com/linuxacademy/metrics-server

Cloning into 'metrics-server'...
remote: Enumerating objects: 9589, done.
remote: Total 9589 (delta 0), reused 0 (delta 0), pack-reused 9589
Receiving objects: 100% (9589/9589), 11.23 MiB | 2.97 MiB/s, done.
Resolving deltas: 100% (4853/4853), done.
```

Debemos de llevar a cabo un ajuste en el archivo de manifiesto **metrics-server-deployment.yaml**, por que contiene un error:

```
$ ./metrics-server/deploy/1.8+/metrics-server-deployment.yaml
```

Debemos de cambiar para últimas versiones de Kubernetes, la versión de la apli del deployment:

```
...
apiVersion: extensions/v1beta1
kind: Deployment
...
```

Debemos de cambiarlo a: ***apiVersion: apps/v1***

A continuación, vamos a hacer el despliegue en kubernetes a partir de los manifiestos que nos hemos descargado:

```
$ kubectl apply -f ./metrics-server/deploy/1.8+/

clusterrole.rbac.authorization.k8s.io/system:aggregated-metrics-reader created
clusterrolebinding.rbac.authorization.k8s.io/metrics-server:system:auth-delegator created
rolebinding.rbac.authorization.k8s.io/metrics-server-auth-reader created
apiservice.apiregistration.k8s.io/v1beta1.metrics.k8s.io created
serviceaccount/metrics-server created
service/metrics-server created
clusterrole.rbac.authorization.k8s.io/system:metrics-server created
clusterrolebinding.rbac.authorization.k8s.io/system:metrics-server created
```

76.2. Comprobando el servidor de métricas

Vamos a comprobar si el servidor de métricas está operando, ejecutamos el siguiente comando:

```
$ kubectl get --raw /apis/metrics.k8s.io/
{"kind":"APIGroup","apiVersion":"v1","name":"metrics.k8s.io","versions":[{"groupVersion":"metrics.k8s.io/v1beta1","version":"v1beta1"}],"preferredVersion":{"groupVersion":"metrics.k8s.io/v1beta1","version":"v1beta1"}}
```



Una vez el servidor de métricas se ha puesto en marcha, debemos de dejarlo un par de minutos que comience la recolección de métricas del clúster

76.3. kubectl top

Mediante el comando top vamos a poder consultar el uso de CPU y memoria por los nodos del clúster:

```
$ kubectl top node
```

NAME	CPU(cores)	CPU%	MEMORY(bytes)	MEMORY%
kubemaster	272m	13%	2933Mi	51%
kubeminion1	114m	11%	533Mi	59%
kubeminion2	89m	8%	484Mi	54%

- Observamos el uso de CPU (1000m = 1Core)
- Observamos el porcentaje de uso de los cores de cada nodo
- También visualizamos la RAM que está utilizando cada nodo
- Y por último, el porcentaje de memoria RAM que está usando cada nodo

Por otra parte, también vamos a poder consultar los consumos de los pods:

```
$ kubectl top pod
```

NAME	CPU(cores)	MEMORY(bytes)
example-https	0m	9Mi

- Observamos por cada Pod, el uso de CPU y RAM
- Los Pods que aparecen son los del namespace sobre el que nos encontramos

Si nos interesa monitorizar todos los pods de todos los namespaces, podemos parametrizar la consulta de la siguiente forma:

```
$ kubectl top pods --all-namespaces
```

NAMESPACE	NAME	CPU(cores)	MEMORY(bytes)
default	example-https	7m	9Mi
kube-system	coredns-6955765f44-bzz62	4m	17Mi
kube-system	coredns-6955765f44-klj6b	4m	13Mi
kube-system	etcd-kubemaster	30m	324Mi
kube-system	kube-apiserver-kubemaster	53m	275Mi
kube-system	kube-controller-manager-kubemaster	21m	43Mi
kube-system	kube-proxy-gg55k	5m	14Mi
kube-system	kube-proxy-gnpx2	1m	20Mi
kube-system	kube-proxy-jpvr8	2m	9Mi
kube-system	kube-scheduler-kubemaster	6m	18Mi
kube-system	metrics-server-f76f648c7-2gvk5	2m	11Mi
kube-system	my-scheduler-789cb54bc9-8gfjp	1m	15Mi
kube-system	weave-net-52qlp	3m	71Mi
kube-system	weave-net-gm5sg	2m	84Mi
kube-system	weave-net-hs5qq	2m	56Mi

También podemos cerrar la monitorización a un namespace específico:

```
$ kubectl top pods --namespace kube-system
```

NAME	CPU(cores)	MEMORY(bytes)
coredns-6955765f44-bzz62	5m	17Mi
coredns-6955765f44-klj6b	5m	13Mi
etcd-kubemaster	28m	324Mi
kube-apiserver-kubemaster	57m	275Mi
kube-controller-manager-kubemaster	24m	43Mi
kube-proxy-gg55k	1m	14Mi
kube-proxy-gnpx2	1m	20Mi
kube-proxy-jpvr8	1m	9Mi
kube-scheduler-kubemaster	8m	18Mi
metrics-server-f76f648c7-2gvk5	2m	11Mi
my-scheduler-789cb54bc9-8gfjp	1m	15Mi
weave-net-52qlp	2m	71Mi
weave-net-gm5sg	4m	84Mi
weave-net-hs5qq	2m	56Mi

Otra variante sería la monitorización de consumo de un único Pod:

```
$ kubectl top pod example-https
```

NAME	CPU(cores)	MEMORY(bytes)
example-https	1m	9Mi

Por otra parte, puede resultar útil la obtención de métricas a nivel de contenedores del Pod de

forma individual:

```
$ kubectl top pods example-https --containers
```

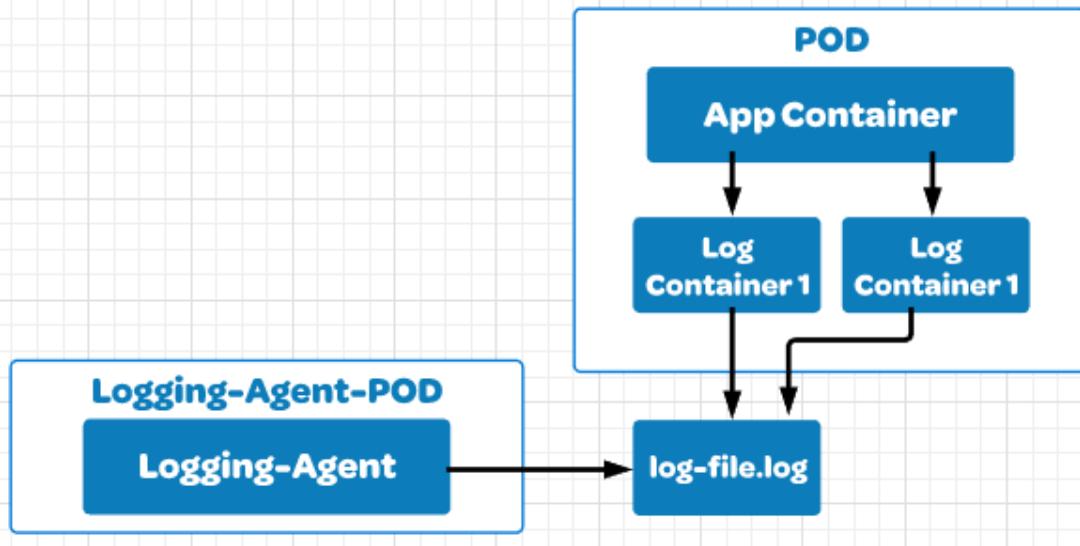
POD	NAME	CPU(cores)	MEMORY(bytes)
example-https	web-server	0m	5Mi
example-https	html-web	2m	4Mi

- En este caso observamos por separado las métricas de cada uno de los contenedores por separado

76.4. Cluster Logs

Cluster Logs

The log directory for containers is in `/var/log/containers`, which can potentially consume all the node's disk space. To manage this, use the sidecar technique with a logging agent.



Existen varias formas de administrar los registros que se pueden acumular tanto de las aplicaciones como de los componentes del sistema.

Las aplicaciones que podemos encontrar en el mercado, así como el propio SystemLog pueden ayudarnos a comprender qué está pasando dentro del clúster si alguna aplicación sufre alguna anomalía

Los Logs son de vital importancia en cuanto al monitoreo y operaciones de debug

Los Logs de las diferentes aplicaciones, pueden ir creciendo con el tiempo y acumulándose, cuando tenemos despliegues de varios microservicios... La tarea de la gestión del Log puede llegar a complicarse si no tenemos buenas técnicas de gestión

Lo primero que tenemos que conocer, es que el directorio por defecto interno donde el contenedor almacena el log:

```
/var/log/containers
```

En cada nodo, dependiendo de los Pods que se estén operando, aparecerá dicha carpeta con distinto contenido

Vamos a crear un Pod que escribe dos archivos diferentes de log, utilizando dos formatos diferentes de salida, pero soltándolos en la misma localización.

Creamos el archivo **pod-with-2-flow-logs.yml** y le agregamos el siguiente contenido:

```
apiVersion: v1
kind: Pod
metadata:
  name: counter
spec:
  containers:
  - name: count
    image: busybox
    args:
      - /bin/sh
      - -c
      - >
        i=0;
        while true;
        do
          echo "$i: $(date)" >> /var/log/1.log;
          echo "$(date) INFO $i" >> /var/log/2.log;
          i=$((i+1));
          sleep 1;
        done
  volumeMounts:
  - name: varlog
    mountPath: /var/log
volumes:
- name: varlog
  emptyDir: {}
```

Ejecutamos la creación del archivo:

```
$ kubectl apply -f pod-with-2-flow-logs.yml
pod/counter created
```

Ahora, vamos a visualizar los logs internos del volumen que hemos montado dentro del contenedor, en la ruta **/var/log**:



```
$ kubectl exec counter -- ls /var/log
```

```
1.log  
2.log
```

- Observamos que aparecen 2 archivos de logs
- Cada archivo representa la salida de traza de los dos 2 echo que tenemos

A continuación, vamos a introducir un **patrón sidecar** en el contenedor, de manera que los logs nos los muestre por consola.

Primero, vamos a borrar el pod anterior con nombre **counter**:

```
$ kubectl delete pod counter
```

```
pod "counter" deleted
```

Editamos de nuevo el archivo **pod-with-2-flow-logs.yml** y le dejamos el siguiente contenido:



```

apiVersion: v1
kind: Pod
metadata:
  name: counter
spec:
  containers:
  - name: count
    image: busybox
    args:
      - /bin/sh
      - -c
      - >
        i=0;
        while true;
        do
          echo "$i: $(date)" >> /var/log/1.log;
          echo "$(date) INFO $i" >> /var/log/2.log;
          i=$((i+1));
          sleep 1;
        done
    volumeMounts:
    - name: varlog
      mountPath: /var/log
  - name: count-log-1
    image: busybox
    args: [/bin/sh, -c, 'tail -n+1 -f /var/log/1.log']
    volumeMounts:
    - name: varlog
      mountPath: /var/log
  - name: count-log-2
    image: busybox
    args: [/bin/sh, -c, 'tail -n+1 -f /var/log/2.log']
    volumeMounts:
    - name: varlog
      mountPath: /var/log
  volumes:
  - name: varlog
    emptyDir: {}

```

- Observamos ahora, que el Pod está compuesto por 3 contenedores
- Observamos que el contenedor **count-log-1** tiene como objetivo realizar un tail del contenido que vaya apareciendo en el archivo **/var/log/1.log**
- Por otra parte, el contenedor **count-log-2** tiene como objetivo, realizar un tail del contenido que vaya apareciendo en el archivo **/var/log/2.log**

Ejecutamos de nuevo la creación del Pod:

```
$ kubectl apply -f pod-with-2-flow-logs.yml
```

```
pod/counter created
```

Ahora, podemos obtener de manera separada los logs de cada archivo por la salida standard de consola.

Obtenemos el log del primer contenedor (archivo 1.log):

```
$ kubectl logs counter count-log-1
```

```
0: Sat Jan 11 13:06:32 UTC 2020  
1: Sat Jan 11 13:06:33 UTC 2020  
2: Sat Jan 11 13:06:34 UTC 2020  
3: Sat Jan 11 13:06:35 UTC 2020  
4: Sat Jan 11 13:06:36 UTC 2020  
5: Sat Jan 11 13:06:37 UTC 2020  
...
```



Obtenemos el log del segundo contenedor (archivo 2.log):

```
$ kubectl logs counter count-log-2
```

```
Sat Jan 11 13:06:32 UTC 2020 INFO 0  
Sat Jan 11 13:06:33 UTC 2020 INFO 1  
Sat Jan 11 13:06:34 UTC 2020 INFO 2  
Sat Jan 11 13:06:35 UTC 2020 INFO 3  
Sat Jan 11 13:06:36 UTC 2020 INFO 4  
...
```



Capítulo 77. Dashboard

Kubernetes dispone de un dashboard desde el cual podemos observar nuestra infraestructura, es un panel de control de propósito general, permite a los usuarios manejar aplicaciones que se están ejecutando en un clúster y poder realizar ciertos ajustes.

El proyecto está disponible en GitHub: <https://github.com/kubernetes/dashboard>



Capítulo 78. Lab: Dashboard Kubernetes

Mediante este laboratorio, pondremos en marcha el dashboard UI para kubernetes.

78.1. Instalación mediante manifiesto

Para poner en marcha el dashboard de kubernetes así como de forma conjunta, todos los elementos que paralelamente son necesarios, vamos a ejecutar la siguiente instrucción, que aplicará directamente el manifiesto en nuestro clúster:

```
$ kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.0.0-rc6/aio/deploy/recommended.yaml

namespace/kubernetes-dashboard created
serviceaccount/kubernetes-dashboard created
service/kubernetes-dashboard created
secret/kubernetes-dashboard-certs created
secret/kubernetes-dashboard-csrf created
secret/kubernetes-dashboard-key-holder created
configmap/kubernetes-dashboard-settings created
role.rbac.authorization.k8s.io/kubernetes-dashboard created
clusterrole.rbac.authorization.k8s.io/kubernetes-dashboard created
rolebinding.rbac.authorization.k8s.io/kubernetes-dashboard created
clusterrolebinding.rbac.authorization.k8s.io/kubernetes-dashboard created
deployment.apps/kubernetes-dashboard created
service/dashboard-metrics-scraper created
deployment.apps/dashboard-metrics-scraper created
```



Fondos Europeos



Todos los componentes del dashboard se han circunscrito a un dashboard específico que se ha creado, con nombre **namespace/kubernetes-dashboard**

Procedemos a listar los namespaces donde debería de aparecer entre otros, el correspondiente al dashboard de kubernetes:

```
$ kubectl get namespaces

NAME           STATUS  AGE
...
kubernetes-dashboard  Active  7m15s
...
```



Cofinanciado por
la Unión Europea



78.2. Activando el proxy de conexión

Una vez está el dashboard operando, procederemos a activar el proxy de conexión con el portal.

Ejecutamos en consola el siguiente comando:

```
$ kubectl proxy  
Starting to serve on 127.0.0.1:8001
```

Seguidamente, introducimos la siguiente URL en el navegador para acceder al portal:

```
http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/proxy/
```

Se mostrará una pantalla de login, donde se nos pedirá que nos identifiquemos mediante dos opciones:

- Mediante un Token
- Mediante un archivo de configuración Kubeconfig

En nuestro caso, utilizaremos la modalidad de autentificación mediante Token

78.3. Creando la cuenta de servicio

Vamos a proceder a crear una cuenta de servicio.

Creamos un archivo con nombre **kubernetes-dashboard-service-account.yml** y le agregamos el siguiente contenido:

```
apiVersion: v1  
kind: ServiceAccount  
metadata:  
  name: admin-user  
  namespace: kubernetes-dashboard
```

Ejecutamos la aplicación del manifiesto:

```
$ kubectl apply -f kubernetes-dashboard-service-account.yml  
serviceaccount/admin-user created
```

78.4. Configurando el ClusterRoleBinding

En la mayoría de los casos, después de haber llevado a cabo el aprovisionamiento del clúster, mediante alguna herramienta como kubeadm, kops o cualquiera otra, en el proceso de creación del clúster, se debe de haber creado de forma automática un ClusterRole con nombre **cluster-admin** que ya debe de existir.

Podemos usar dicho ClusterRole, con lo que únicamente necesitaríamos crear un elemento de tipo ClusterRoleBinding para el objeto ServiceAccount, que es la cuenta de servicio.

Creamos un archivo con nombre **kubernetes-dashboard-cluster-role-binding.yml** y le agregamos el siguiente contenido:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: admin-user
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: admin-user
  namespace: kubernetes-dashboard
```

Ejecutamos la aplicación del manifiesto:

```
$ kubectl apply -f kubernetes-dashboard-cluster-role-binding.yml
clusterrolebinding.rbac.authorization.k8s.io/admin-user created
```

78.5. Obteniendo el Token de acceso

A continuación, vamos a visualizar el token de acceso que está presente en un objeto de Kubernetes de tipo Secret.

Dicho Secret está circunscrito al namespace kubernetes-dashboard, por lo que vamos a ejecutar el siguiente comando en consola:



```
$ kubectl -n kubernetes-dashboard describe secret $(kubectl -n kubernetes-dashboard get secret | grep admin-user | awk '{print $1}')
```

```
Name: admin-user-token-z2n74
Namespace: kubernetes-dashboard
Labels: <none>
Annotations: kubernetes.io/service-account.name: admin-user
kubernetes.io/service-account.uid: 511b6d05-1de5-432d-8609-5d243cadba62
```

```
Type: kubernetes.io/service-account-token
```

```
Data
```

```
====
```

```
ca.crt: 1025 bytes
```

```
namespace: 20 bytes
```

```
token:
```

```
eyJhbGciOiJSUzI1NiIsImtpZCI6Imx6XzZ3eDI0VThkVzE3cGIUQUdZc1RSc0IOSmhFNmhqMkNPeHR3RDhCeVEifQ.eyJpc3MiOiJrdWJlcmt5ldG'ZpY2VhY2NvdW50Iiwia3ViZXJuZXRIcy5pbj9zZXJ2aWNlYWNjb3VudC9uYW1lc3BhY2UiOijrdWJlcmt5ldGVzLWRhc2hib2FyZCIIsImt1YmVybmc'vc2VydmljZWFljY291bnQvc2Vjcmv0Lm5hbWUiOijhZG1pbi11c2VylXRva2VuLXoybjc0Iiwia3ViZXJuZXRIcy5pbj9zZXJ2aWNlYWNjb3VudC9zZLWFjY291bnQubmFtZSI6ImFkbWluLXVzZXIiLCJrdWJlcmt5ldGVzLmlvL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UtYWNjb3VudC51aWQiOiiMTfGU1LTQzMmQtODYwOS01ZDI0M2NhZGjhNjIiLCJzdWIiOijzeXN0ZW06c2VydmljZWFljY291bnQ6a3ViZXJuZXRIcy1kYXNoYm9hcmQ6YWRta'tShNI2VF4xYoVQ4KS9Zwa1JN0gZm-xeAkfl2twXgoT4ws63Rz1y5r-3ZryvUMB9nxa_-k-D6AZdonHd-2Hgh_AS1RUUBiEwW250izoycUZuFfnvOPjjdnocjypYh-X33tNndBQiQHNW1BcpjXcWljYvcI3BIdGQkkfntKO45CMYFJFxS_6AQ3F6yt2EL59rsNp6Fg83NHlcUgIOHbviNXkbQNc4wPzKtS5mdEYIJ22wkuU721Bq-iEKT3w8_bU4iUMv0QAhLqGEXBxh2iVMikXNE_8KsSZbypai_wEY9hoc6kmQJn9uicloSAckI_k6XzpAIjpoQ
```



Copiamos al completo el token y lo introducimos en la pantalla de login del dashboard para finalmente acceder.

Capítulo 79. Glosario de términos

- **Clúster:** Conjunto de máquinas físicas o virtuales que son utilizados por kubernetes
- **Pod:** Es la unidad mínima de kubernetes, realmente es un contenedor en jerga Docker
- **Label y Selectors:** Son pares de claves y valores, las cuales se pueden aplicar a pods, services, replication controllers, etc. Y con ellos podremos identificarlos para poderlos gestionar
- **Node:** Es el servidor, ya sea virtual o físico que aloja el sistema de kubernetes y donde vamos a desplegar nuestros pods (contenedores), también llamados minions
- **Replication Controller:** Es el responsable de gestionar la vida de los pods y el encargado de mantener arrancados los pods que se le hayan indicado en la configuración Permite esclar de forma muy sencilla los sistemas y maneja la recreación de un pod cuando ocurre algún tipo de fallo
- **Replica Sets:** Es la nueva generación del Replication Controller, con nuevas funcionalidades, una de las funcionalidades, es que nos permite desplegar pods en función de los labels y selectores
- **Deployments:** Es donde se especifican la cantidad de réplicas de pods que tendremos en el sistema. Se trata de una funcionalidad más avanzada que los Replication Controller y muy parecida a los Replication Sets, pero con otras características
- **Namespaces:** Son agrupaciones, en ellos podremos diferenciar espacios de trabajo para diferentes situaciones. Podríamos utilizar un namespace para producción y otro para desarrollo, y cada namespace tendría sus propios pods, replication controllers, etc.
- **Volumes:** Es el acceso a un sistema de almacenamiento
- **Secrets:** Es donde se guarda la información confidencial como usuarios y passwords, para poder acceder a los recursos
- **Service:** Es la política de acceso a los pods, se puede definir como la abstracción que define un conjunto de pods y la lógica para poder acceder a ellos

Capítulo 80. Helm

La palabra **Helm** en inglés, una de las referencias es: Caso, un mecanismo de dirección del buque, la rueda de la nave... El timón.

Helm es una herramienta para gestionar aplicaciones de Kubernetes.

Helm nos ayuda a "timonear" Kubernetes usando las **cartas de navegación**, llamadas en inglés **Helm Charts**.

La función principal de Helm consiste en:

- Definir
- Instalar
- Actualizar aplicaciones complejas de Kubernetes

Helm es mantenido por la **CNCF** en colaboración con Microsoft, Google, Bitnami y la comunidad de Helm.

La CNCF: <https://www.cncf.io/>

La Cloud Native Computing Foundation (CNCF) aloja componentes críticos de la infraestructura tecnológica global.



CNCF reúne a los principales desarrolladores, usuarios finales y proveedores del mundo y organiza las mayores conferencias de desarrolladores de código abierto.

CNCF es parte de la Fundación Linux sin fines de lucro.

80.1. Helm Charts

Con Helm Charts es posible llevar a cabo las siguientes operaciones con una aplicación:

- Crear
- Versionar
- Publicar

Cuando usamos Helm Charts es como si tuviéramos un asistente de optimización que facilita la administración e instalación de las aplicaciones en un clúster de Kubernetes, así como el proceso de empaquetamiento.

80.2. Arquitectura de Helm Charts

Helm Charts se divide en dos vertientes:

- Helm
 - Actúa como cliente

- Tiller
 - Actúa como servidor de Helm

80.3. Arquitectura de Helm Charts (Tiller)

Tiller es el componente que se encarga de la gestión de los Charts, específicamente en nuestras instalaciones.

Tiller interactúa directamente con el API de Kubernetes para realizar operaciones de gestión de recursos Kubernetes:

- Instalar
- Actualizar
- Consultar
- Eliminar

También almacena los objetos de cada release o distribución.



80.4. Arquitectura de Helm Charts (Helm)

Helm por su parte, se va a ejecutar directamente en el equipo elegido para la ejecución.

Los paquetes Helm están compuestos de una descripción del paquete y de archivos contenedores de manifiestos de Kubernetes.



80.5. Beneficios de utilizar Helm

Al crear las **cartas de navegación** de Helm, los Helm Charts sirven para describir incluso las aplicaciones más complejas.

Ofrecen una **instalación repetible** de la aplicación, manteniendo un **único punto de control**.

Las actualizaciones de Helm Charts son sencillas y más fáciles de utilizar para los desarrolladores.

Los Helm Charts buscan ser fáciles de versionar, compartir y alojar a través de tecnologías cloud.

Al desplegar una aplicación nueva, es posible tener que revertir la acción por cualquier razón.

El proceso de **rollback** con Helm Charts es sencillo, se usa para retroceder la versión de una publicación anterior en caso de que el proceso no haya sido satisfactorio



80.5.1. Estado actual del proyecto

Helm es otra solución que ha aparecido en el mundo de los contenedores para hacer más efectiva la adopción de los mismos.

Pero debemos de tener en cuenta que aún es un proyecto muy reciente y está en proceso de adopción.

Podemos consultar la hoja de ruta en la siguiente URL: <https://github.com/helm/helm/wiki/Roadmap>

