

# Kubernetes IN ACTION

Marko Lukša

## ***Kubernetes resources covered in the book***

	<b>Resource (abbr.) [API version]</b>	<b>Description</b>	<b>Section</b>
	<b>Namespace*</b> (ns) [v1]	Enables organizing resources into non-overlapping groups (for example, per tenant)	3.7
<b>Deploying workloads</b>	<b>Pod</b> (po) [v1]	The basic deployable unit containing one or more processes in co-located containers	3.1
	<b>ReplicaSet</b> (rs) [apps/v1beta2**]	Keeps one or more pod replicas running	4.3
	<b>ReplicationController</b> (rc) [v1]	The older, less-powerful equivalent of a ReplicaSet	4.2
	<b>Job</b> [batch/v1]	Runs pods that perform a completable task	4.5
	<b>CronJob</b> [batch/v1beta1]	Runs a scheduled job once or periodically	4.6
	<b>DaemonSet</b> (ds) [apps/v1beta2**]	Runs one pod replica per node (on all nodes or only on those matching a node selector)	4.4
	<b>StatefulSet</b> (sts) [apps/v1beta1**]	Runs stateful pods with a stable identity	10.2
	<b>Deployment</b> (deploy) [apps/v1beta1**]	Declarative deployment and updates of pods	9.3
<b>Services</b>	<b>Service</b> (svc) [v1]	Exposes one or more pods at a single and stable IP address and port pair	5.1
	<b>Endpoints</b> (ep) [v1]	Defines which pods (or other servers) are exposed through a service	5.2.1
	<b>Ingress</b> (ing) [extensions/v1beta1]	Exposes one or more services to external clients through a single externally reachable IP address	5.4
<b>Config</b>	<b>ConfigMap</b> (cm) [v1]	A key-value map for storing non-sensitive config options for apps and exposing it to them	7.4
	<b>Secret</b> [v1]	Like a ConfigMap, but for sensitive data	7.5
<b>Storage</b>	<b>PersistentVolume*</b> (pv) [v1]	Points to persistent storage that can be mounted into a pod through a PersistentVolumeClaim	6.5
	<b>PersistentVolumeClaim</b> (pvc) [v1]	A request for and claim to a PersistentVolume	6.5
	<b>StorageClass*</b> (sc) [storage.k8s.io/v1]	Defines the type of dynamically-provisioned storage claimable in a PersistentVolumeClaim	6.6

\* Cluster-level resource (not namespaced)

\*\* Also in other API versions; listed version is the one used in this book

*(continues on inside back cover)*

## *Kubernetes in Action*



# *Kubernetes in Action*

MARKO LUKŠA



MANNING  
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit [www.manning.com](http://www.manning.com). The publisher offers discounts on this book when ordered in quantity. For more information, please contact


Special Sales Department  
Manning Publications Co.  
20 Baldwin Road  
PO Box 761  
Shelter Island, NY 11964  
Email: [orders@manning.com](mailto:orders@manning.com)

©2018 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

- © Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

 Manning Publications Co.  
20 Baldwin Road  
PO Box 761  
Shelter Island, NY 11964

Development editor: Elesha Hyde  
Review editor: Aleksandar Dragosavljević  
Technical development editor: Jeanne Boyarsky  
Project editor: Kevin Sullivan  
Copyeditor: Katie Petito  
Proofreader: Melody Dolab  
Technical proofreader: Antonio Magnaghi  
Illustrator: Chuck Larson  
Typesetter: Dennis Dalinnik  
Cover designer: Marija Tudor

ISBN: 9781617293726

Printed in the United States of America

1 2 3 4 5 6 7 8 9 10 – EBM – 22 21 20 19 18 17

*To my parents,  
who have always put their children's needs above their own*





# *brief contents*

---

## **PART 1 OVERVIEW**

- 1 ■ Introducing Kubernetes 1
- 2 ■ First steps with Docker and Kubernetes 25

## **PART 2 CORE CONCEPTS**

- 3 ■ Pods: running containers in Kubernetes 55
- 4 ■ Replication and other controllers: deploying managed pods 84
- 5 ■ Services: enabling clients to discover and talk to pods 120
- 6 ■ Volumes: attaching disk storage to containers 159
- 7 ■ ConfigMaps and Secrets: configuring applications 191
- 8 ■ Accessing pod metadata and other resources from applications 225
- 9 ■ Deployments: updating applications declaratively 250
- 10 ■ StatefulSets: deploying replicated stateful applications 280

**PART 3 BEYOND THE BASICS**

- 11 ■ Understanding Kubernetes internals 309
- 12 ■ Securing the Kubernetes API server 346
- 13 ■ Securing cluster nodes and the network 375
- 14 ■ Managing pods' computational resources 404
- 15 ■ Automatic scaling of pods and cluster nodes 437
- 16 ■ Advanced scheduling 457
- 17 ■ Best practices for developing apps 477
- 18 ■ Extending Kubernetes 508

# contents

---

<i>preface</i>	<i>xxi</i>
<i>acknowledgments</i>	<i>xxiii</i>
<i>about this book</i>	<i>xxv</i>
<i>about the author</i>	<i>xxix</i>
<i>about the cover illustration</i>	<i>xxx</i>

## PART 1 OVERVIEW

### **1** *Introducing Kubernetes* 1

- 1.1 Understanding the need for a system like Kubernetes 2
  - Moving from monolithic apps to microservices* 3 ■ *Providing a consistent environment to applications* 6 ■ *Moving to continuous delivery: DevOps and NoOps* 6
- 1.2 Introducing container technologies 7
  - Understanding what containers are* 8 ■ *Introducing the Docker container platform* 12 ■ *Introducing rkt—an alternative to Docker* 15
- 1.3 Introducing Kubernetes 16
  - Understanding its origins* 16 ■ *Looking at Kubernetes from the top of a mountain* 16 ■ *Understanding the architecture of a Kubernetes cluster* 18 ■ *Running an application in Kubernetes* 19
  - Understanding the benefits of using Kubernetes* 21
- 1.4 Summary 23

<b>2</b>	<b><i>First steps with Docker and Kubernetes</i></b>	<b>25</b>
2.1	Creating, running, and sharing a container image	26
	<i>Installing Docker and running a Hello World container</i>	26
	<i>Creating a trivial Node.js app</i>	28
	▪ <i>Creating a Dockerfile for the image</i>	29
	▪ <i>Building the container image</i>	29
	<i>Running the container image</i>	32
	▪ <i>Exploring the inside of a running container</i>	33
	▪ <i>Stopping and removing a container</i>	34
	▪ <i>Pushing the image to an image registry</i>	35
2.2	Setting up a Kubernetes cluster	36
	<i>Running a local single-node Kubernetes cluster with Minikube</i>	37
	<i>Using a hosted Kubernetes cluster with Google Kubernetes Engine</i>	38
	▪ <i>Setting up an alias and command-line completion for kubectl</i>	41
2.3	Running your first app on Kubernetes	42
	<i>Deploying your Node.js app</i>	42
	▪ <i>Accessing your web application</i>	45
	▪ <i>The logical parts of your system</i>	47
	<i>Horizontally scaling the application</i>	48
	▪ <i>Examining what nodes your app is running on</i>	51
	▪ <i>Introducing the Kubernetes dashboard</i>	52
2.4	Summary	53

## PART 2 CORE CONCEPTS

<b>3</b>	<b><i>Pods: running containers in Kubernetes</i></b>	<b>55</b>
3.1	Introducing pods	56
	<i>Understanding why we need pods</i>	56
	▪ <i>Understanding pods</i>	57
	<i>Organizing containers across pods properly</i>	58
3.2	Creating pods from YAML or JSON descriptors	61
	<i>Examining a YAML descriptor of an existing pod</i>	61
	▪ <i>Creating a simple YAML descriptor for a pod</i>	63
	▪ <i>Using kubectl create to create the pod</i>	65
	▪ <i>Viewing application logs</i>	65
	▪ <i>Sending requests to the pod</i>	66
3.3	Organizing pods with labels	67
	<i>Introducing labels</i>	68
	▪ <i>Specifying labels when creating a pod</i>	69
	<i>Modifying labels of existing pods</i>	70
3.4	Listing subsets of pods through label selectors	71
	<i>Listing pods using a label selector</i>	71
	▪ <i>Using multiple conditions in a label selector</i>	72

- 3.5 Using labels and selectors to constrain pod scheduling 73
  - Using labels for categorizing worker nodes* 74 ■ *Scheduling pods to specific nodes* 74 ■ *Scheduling to one specific node* 75
- 3.6 Annotating pods 75
  - Looking up an object's annotations* 75 ■ *Adding and modifying annotations* 76
- 3.7 Using namespaces to group resources 76
  - Understanding the need for namespaces* 77 ■ *Discovering other namespaces and their pods* 77 ■ *Creating a namespace* 78
  - Managing objects in other namespaces* 79 ■ *Understanding the isolation provided by namespaces* 79
- 3.8 Stopping and removing pods 80
  - Deleting a pod by name* 80 ■ *Deleting pods using label selectors* 80 ■ *Deleting pods by deleting the whole namespace* 80 ■ *Deleting all pods in a namespace, while keeping the namespace* 81 ■ *Deleting (almost) all resources in a namespace* 82
- 3.9 Summary 82

## 4 **Replication and other controllers: deploying managed pods** 84

- 4.1 Keeping pods healthy 85
  - Introducing liveness probes* 85 ■ *Creating an HTTP-based liveness probe* 86 ■ *Seeing a liveness probe in action* 87
  - Configuring additional properties of the liveness probe* 88
  - Creating effective liveness probes* 89
- 4.2 Introducing ReplicationControllers 90
  - The operation of a ReplicationController* 91 ■ *Creating a ReplicationController* 93 ■ *Seeing the ReplicationController in action* 94 ■ *Moving pods in and out of the scope of a ReplicationController* 98 ■ *Changing the pod template* 101
  - Horizontally scaling pods* 102 ■ *Deleting a ReplicationController* 103
- 4.3 Using ReplicaSets instead of ReplicationControllers 104
  - Comparing a ReplicaSet to a ReplicationController* 105
  - Defining a ReplicaSet* 105 ■ *Creating and examining a ReplicaSet* 106 ■ *Using the ReplicaSet's more expressive label selectors* 107 ■ *Wrapping up ReplicaSets* 108

- 4.4 Running exactly one pod on each node with DaemonSets 108
  - Using a DaemonSet to run a pod on every node* 109
  - Using a DaemonSet to run pods only on certain nodes* 109
- 4.5 Running pods that perform a single completable task 112
  - Introducing the Job resource* 112 ■ *Defining a Job resource* 113
  - Seeing a Job run a pod* 114 ■ *Running multiple pod instances in a Job* 114 ■ *Limiting the time allowed for a Job pod to complete* 116
- 4.6 Scheduling Jobs to run periodically or once in the future 116
  - Creating a CronJob* 116 ■ *Understanding how scheduled jobs are run* 117
- 4.7 Summary 118

## 5 *Services: enabling clients to discover and talk to pods* 120

- 5.1 Introducing services 121
  - Creating services* 122 ■ *Discovering services* 128
- 5.2 Connecting to services living outside the cluster 131
  - Introducing service endpoints* 131 ■ *Manually configuring service endpoints* 132 ■ *Creating an alias for an external service* 134
- 5.3 Exposing services to external clients 134
  - Using a NodePort service* 135 ■ *Exposing a service through an external load balancer* 138 ■ *Understanding the peculiarities of external connections* 141
- 5.4 Exposing services externally through an Ingress resource 142
  - Creating an Ingress resource* 144 ■ *Accessing the service through the Ingress* 145 ■ *Exposing multiple services through the same Ingress* 146 ■ *Configuring Ingress to handle TLS traffic* 147
- 5.5 Signaling when a pod is ready to accept connections 149
  - Introducing readiness probes* 149 ■ *Adding a readiness probe to a pod* 151 ■ *Understanding what real-world readiness probes should do* 153

- 5.6 Using a headless service for discovering individual pods 154
  - Creating a headless service* 154 ■ *Discovering pods through DNS* 155 ■ *Discovering all pods—even those that aren't ready* 156
- 5.7 Troubleshooting services 156
- 5.8 Summary 157

## 6 **Volumes: attaching disk storage to containers** 159

- 6.1 Introducing volumes 160
  - Explaining volumes in an example* 160 ■ *Introducing available volume types* 162
- 6.2 Using volumes to share data between containers 163
  - Using an emptyDir volume* 163 ■ *Using a Git repository as the starting point for a volume* 166
- 6.3 Accessing files on the worker node's filesystem 169
  - Introducing the hostPath volume* 169 ■ *Examining system pods that use hostPath volumes* 170
- 6.4 Using persistent storage 171
  - Using a GCE Persistent Disk in a pod volume* 171 ■ *Using other types of volumes with underlying persistent storage* 174
- 6.5 Decoupling pods from the underlying storage technology 176
  - Introducing PersistentVolumes and PersistentVolumeClaims* 176
  - Creating a PersistentVolume* 177 ■ *Claiming a PersistentVolume by creating a PersistentVolumeClaim* 179 ■ *Using a PersistentVolumeClaim in a pod* 181 ■ *Understanding the benefits of using PersistentVolumes and claims* 182 ■ *Recycling PersistentVolumes* 183
- 6.6 Dynamic provisioning of PersistentVolumes 184
  - Defining the available storage types through StorageClass resources* 185 ■ *Requesting the storage class in a PersistentVolumeClaim* 185 ■ *Dynamic provisioning without specifying a storage class* 187
- 6.7 Summary 190

## 7 *ConfigMaps and Secrets: configuring applications* 191

- 7.1 Configuring containerized applications 191
- 7.2 Passing command-line arguments to containers 192
  - Defining the command and arguments in Docker* 193
  - Overriding the command and arguments in Kubernetes* 195
- 7.3 Setting environment variables for a container 196
  - Specifying environment variables in a container definition* 197
  - Referring to other environment variables in a variable's value* 198
  - Understanding the drawback of hardcoding environment variables* 198
- 7.4 Decoupling configuration with a ConfigMap 198
  - Introducing ConfigMaps* 198 ■ *Creating a ConfigMap* 200
  - Passing a ConfigMap entry to a container as an environment variable* 202 ■ *Passing all entries of a ConfigMap as environment variables at once* 204 ■ *Passing a ConfigMap entry as a command-line argument* 204 ■ *Using a configMap volume to expose ConfigMap entries as files* 205 ■ *Updating an app's config without having to restart the app* 211
- 7.5 Using Secrets to pass sensitive data to containers 213
  - Introducing Secrets* 214 ■ *Introducing the default token Secret* 214 ■ *Creating a Secret* 216 ■ *Comparing ConfigMaps and Secrets* 217 ■ *Using the Secret in a pod* 218
  - Understanding image pull Secrets* 222
- 7.6 Summary 224

## 8 *Accessing pod metadata and other resources from applications* 225

- 8.1 Passing metadata through the Downward API 226
  - Understanding the available metadata* 226 ■ *Exposing metadata through environment variables* 227 ■ *Passing metadata through files in a downwardAPI volume* 230
- 8.2 Talking to the Kubernetes API server 233
  - Exploring the Kubernetes REST API* 234 ■ *Talking to the API server from within a pod* 238 ■ *Simplifying API server communication with ambassador containers* 243 ■ *Using client libraries to talk to the API server* 246
- 8.3 Summary 249



## 9 *Deployments: updating applications declaratively* 250

- 9.1 Updating applications running in pods 251
  - Deleting old pods and replacing them with new ones* 252
  - Spinning up new pods and then deleting the old ones* 252
- 9.2 Performing an automatic rolling update with a ReplicationController 254
  - Running the initial version of the app* 254 ■ *Performing a rolling update with kubectrl* 256 ■ *Understanding why kubectrl rolling-update is now obsolete* 260
- 9.3 Using Deployments for updating apps declaratively 261
  - Creating a Deployment* 262 ■ *Updating a Deployment* 264
  - Rolling back a deployment* 268 ■ *Controlling the rate of the rollout* 271 ■ *Pausing the rollout process* 273 ■ *Blocking rollouts of bad versions* 274
- 9.4 Summary 279

## 10 *StatefulSets: deploying replicated stateful applications* 280

- 10.1 Replicating stateful pods 281
  - Running multiple replicas with separate storage for each* 281
  - Providing a stable identity for each pod* 282
- 10.2 Understanding StatefulSets 284
  - Comparing StatefulSets with ReplicaSets* 284 ■ *Providing a stable network identity* 285 ■ *Providing stable dedicated storage to each stateful instance* 287 ■ *Understanding StatefulSet guarantees* 289
- 10.3 Using a StatefulSet 290
  - Creating the app and container image* 290 ■ *Deploying the app through a StatefulSet* 291 ■ *Playing with your pods* 295
- 10.4 Discovering peers in a StatefulSet 299
  - Implementing peer discovery through DNS* 301 ■ *Updating a StatefulSet* 302 ■ *Trying out your clustered data store* 303
- 10.5 Understanding how StatefulSets deal with node failures 304
  - Simulating a node's disconnection from the network* 304
  - Deleting the pod manually* 306
- 10.6 Summary 307

## PART 3 BEYOND THE BASICS

<b>11</b>	<b><i>Understanding Kubernetes internals</i></b>	<b>309</b>
11.1	Understanding the architecture	310
	<i>The distributed nature of Kubernetes components</i>	310
	<i>How Kubernetes uses etcd</i>	312
	▪ <i>What the API server does</i>	316
	<i>Understanding how the API server notifies clients of resource changes</i>	318
	▪ <i>Understanding the Scheduler</i>	319
	<i>Introducing the controllers running in the Controller Manager</i>	321
	<i>What the Kubelet does</i>	326
	▪ <i>The role of the Kubernetes Service Proxy</i>	327
	▪ <i>Introducing Kubernetes add-ons</i>	328
	▪ <i>Bringing it all together</i>	330
11.2	How controllers cooperate	330
	<i>Understanding which components are involved</i>	330
	▪ <i>The chain of events</i>	331
	▪ <i>Observing cluster events</i>	332
11.3	Understanding what a running pod is	333
11.4	Inter-pod networking	335
	<i>What the network must be like</i>	335
	▪ <i>Diving deeper into how networking works</i>	336
	▪ <i>Introducing the Container Network Interface</i>	338
11.5	How services are implemented	338
	<i>Introducing the kube-proxy</i>	339
	▪ <i>How kube-proxy uses iptables</i>	339
11.6	Running highly available clusters	341
	<i>Making your apps highly available</i>	341
	▪ <i>Making Kubernetes Control Plane components highly available</i>	342
11.7	Summary	345
<b>12</b>	<b><i>Securing the Kubernetes API server</i></b>	<b>346</b>
12.1	Understanding authentication	346
	<i>Users and groups</i>	347
	▪ <i>Introducing ServiceAccounts</i>	348
	<i>Creating ServiceAccounts</i>	349
	▪ <i>Assigning a ServiceAccount to a pod</i>	351
12.2	Securing the cluster with role-based access control	353
	<i>Introducing the RBAC authorization plugin</i>	353
	▪ <i>Introducing RBAC resources</i>	355
	▪ <i>Using Roles and RoleBindings</i>	358
	<i>Using ClusterRoles and ClusterRoleBindings</i>	362
	<i>Understanding default ClusterRoles and ClusterRoleBindings</i>	371
	<i>Granting authorization permissions wisely</i>	373
12.3	Summary	373

## 13 *Securing cluster nodes and the network* 375

- 13.1 Using the host node's namespaces in a pod 376
  - Using the node's network namespace in a pod* 376 ■ *Binding to a host port without using the host's network namespace* 377
  - Using the node's PID and IPC namespaces* 379
- 13.2 Configuring the container's security context 380
  - Running a container as a specific user* 381 ■ *Preventing a container from running as root* 382 ■ *Running pods in privileged mode* 382 ■ *Adding individual kernel capabilities to a container* 384 ■ *Dropping capabilities from a container* 385
  - Preventing processes from writing to the container's filesystem* 386
  - Sharing volumes when containers run as different users* 387
- 13.3 Restricting the use of security-related features in pods 389
  - Introducing the PodSecurityPolicy resource* 389 ■ *Understanding runAsUser, fsGroup, and supplementalGroups policies* 392
  - Configuring allowed, default, and disallowed capabilities* 394
  - Constraining the types of volumes pods can use* 395 ■ *Assigning different PodSecurityPolicies to different users and groups* 396
- 13.4 Isolating the pod network 399
  - Enabling network isolation in a namespace* 399 ■ *Allowing only some pods in the namespace to connect to a server pod* 400
  - Isolating the network between Kubernetes namespaces* 401
  - Isolating using CIDR notation* 402 ■ *Limiting the outbound traffic of a set of pods* 403
- 13.5 Summary 403

## 14 *Managing pods' computational resources* 404

- 14.1 Requesting resources for a pod's containers 405
  - Creating pods with resource requests* 405 ■ *Understanding how resource requests affect scheduling* 406 ■ *Understanding how CPU requests affect CPU time sharing* 411 ■ *Defining and requesting custom resources* 411
- 14.2 Limiting resources available to a container 412
  - Setting a hard limit for the amount of resources a container can use* 412 ■ *Exceeding the limits* 414 ■ *Understanding how apps in containers see limits* 415
- 14.3 Understanding pod QoS classes 417
  - Defining the QoS class for a pod* 417 ■ *Understanding which process gets killed when memory is low* 420

- 14.4 Setting default requests and limits for pods per namespace 421
  - Introducing the LimitRange resource* 421 ■ *Creating a LimitRange object* 422 ■ *Enforcing the limits* 423
  - Applying default resource requests and limits* 424
- 14.5 Limiting the total resources available in a namespace 425
  - Introducing the ResourceQuota object* 425 ■ *Specifying a quota for persistent storage* 427 ■ *Limiting the number of objects that can be created* 427 ■ *Specifying quotas for specific pod states and/or QoS classes* 429
- 14.6 Monitoring pod resource usage 430
  - Collecting and retrieving actual resource usages* 430 ■ *Storing and analyzing historical resource consumption statistics* 432
- 14.7 Summary 435

## 15 *Automatic scaling of pods and cluster nodes* 437

- 15.1 Horizontal pod autoscaling 438
  - Understanding the autoscaling process* 438 ■ *Scaling based on CPU utilization* 441 ■ *Scaling based on memory consumption* 448 ■ *Scaling based on other and custom metrics* 448 ■ *Determining which metrics are appropriate for autoscaling* 450 ■ *Scaling down to zero replicas* 450
- 15.2 Vertical pod autoscaling 451
  - Automatically configuring resource requests* 451 ■ *Modifying resource requests while a pod is running* 451
- 15.3 Horizontal scaling of cluster nodes 452
  - Introducing the Cluster Autoscaler* 452 ■ *Enabling the Cluster Autoscaler* 454 ■ *Limiting service disruption during cluster scale-down* 454
- 15.4 Summary 456

## 16 *Advanced scheduling* 457

- 16.1 Using taints and tolerations to repel pods from certain nodes 457
  - Introducing taints and tolerations* 458 ■ *Adding custom taints to a node* 460 ■ *Adding tolerations to pods* 460 ■ *Understanding what taints and tolerations can be used for* 461

- 16.2 Using node affinity to attract pods to certain nodes 462
  - Specifying hard node affinity rules* 463
  - *Prioritizing nodes when scheduling a pod* 465
- 16.3 Co-locating pods with pod affinity and anti-affinity 468
  - Using inter-pod affinity to deploy pods on the same node* 468
  - Deploying pods in the same rack, availability zone, or geographic region* 471
  - *Expressing pod affinity preferences instead of hard requirements* 472
  - *Scheduling pods away from each other with pod anti-affinity* 474
- 16.4 Summary 476

## 17 **Best practices for developing apps** 477

- 17.1 Bringing everything together 478
- 17.2 Understanding the pod's lifecycle 479
  - Applications must expect to be killed and relocated* 479
  - Rescheduling of dead or partially dead pods* 482
  - *Starting pods in a specific order* 483
  - *Adding lifecycle hooks* 485
  - Understanding pod shutdown* 489
- 17.3 Ensuring all client requests are handled properly 492
  - Preventing broken client connections when a pod is starting up* 492
  - Preventing broken connections during pod shut-down* 493
- 17.4 Making your apps easy to run and manage in Kubernetes 497
  - Making manageable container images* 497
  - *Properly tagging your images and using imagePullPolicy wisely* 497
  - Using multi-dimensional instead of single-dimensional labels* 498
  - Describing each resource through annotations* 498
  - *Providing information on why the process terminated* 498
  - *Handling application logs* 500
- 17.5 Best practices for development and testing 502
  - Running apps outside of Kubernetes during development* 502
  - Using Minikube in development* 503
  - *Versioning and auto-deploying resource manifests* 504
  - *Introducing Ksonnet as an alternative to writing YAML/JSON manifests* 505
  - *Employing Continuous Integration and Continuous Delivery (CI/CD)* 506
- 17.6 Summary 506

<b>18</b>	<b>Extending Kubernetes</b>	<b>508</b>
18.1	Defining custom API objects	508
	<i>Introducing CustomResourceDefinitions</i>	509
	<i>Automating custom resources with custom controllers</i>	513
	<i>Validating custom objects</i>	517
	<i>Providing a custom API server for your custom objects</i>	518
18.2	Extending Kubernetes with the Kubernetes Service Catalog	519
	<i>Introducing the Service Catalog</i>	520
	<i>Introducing the Service Catalog API server and Controller Manager</i>	521
	<i>Introducing Service Brokers and the OpenServiceBroker API</i>	522
	<i>Provisioning and using a service</i>	524
	<i>Unbinding and deprovisioning</i>	526
	<i>Understanding what the Service Catalog brings</i>	526
18.3	Platforms built on top of Kubernetes	527
	<i>Red Hat OpenShift Container Platform</i>	527
	<i>Deis Workflow and Helm</i>	530
18.4	Summary	533
appendix A	Using kubectl with multiple clusters	534
appendix B	Setting up a multi-node cluster with kubeadm	539
appendix C	Using other container runtimes	552
appendix D	Cluster Federation	556
	index	561

## *preface*

---

After working at Red Hat for a few years, in late 2014 I was assigned to a newly-established team called Cloud Enablement. Our task was to bring the company's range of middleware products to the OpenShift Container Platform, which was then being developed on top of Kubernetes. At that time, Kubernetes was still in its infancy—version 1.0 hadn't even been released yet.

Our team had to get to know the ins and outs of Kubernetes quickly to set a proper direction for our software and take advantage of everything Kubernetes had to offer. When faced with a problem, it was hard for us to tell if we were doing things wrong or merely hitting one of the early Kubernetes bugs.

Both Kubernetes and my understanding of it have come a long way since then. When I first started using it, most people hadn't even heard of Kubernetes. Now, virtually every software engineer knows about it, and it has become one of the fastest-growing and most-widely-adopted ways of running applications in both the cloud and on-premises datacenters.

In my first month of dealing with Kubernetes, I wrote a two-part blog post about how to run a JBoss WildFly application server cluster in OpenShift/Kubernetes. At the time, I never could have imagined that a simple blog post would ultimately lead the people at Manning to contact me about whether I would like to write a book about Kubernetes. Of course, I couldn't say no to such an offer, even though I was sure they'd approached other people as well and would ultimately pick someone else.

And yet, here we are. After more than a year and a half of writing and researching, the book is done. It's been an awesome journey. Writing a book about a technology is

absolutely the best way to get to know it in much greater detail than you'd learn as just a user. As my knowledge of Kubernetes has expanded during the process and Kubernetes itself has evolved, I've constantly gone back to previous chapters I've written and added additional information. I'm a perfectionist, so I'll never really be absolutely satisfied with the book, but I'm happy to hear that a lot of readers of the Manning Early Access Program (MEAP) have found it to be a great guide to Kubernetes.

My aim is to get the reader to understand the technology itself and teach them how to use the tooling to effectively and efficiently develop and deploy apps to Kubernetes clusters. In the book, I don't put much emphasis on how to actually set up and maintain a proper highly available Kubernetes cluster, but the last part should give readers a very solid understanding of what such a cluster consists of and should allow them to easily comprehend additional resources that deal with this subject.

I hope you'll enjoy reading it, and that it teaches you how to get the most out of the awesome system that is Kubernetes.



## *acknowledgments*

---

Before I started writing this book, I had no clue how many people would be involved in bringing it from a rough manuscript to a published piece of work. This means there are a lot of people to thank.

First, I'd like to thank Erin Twohey for approaching me about writing this book, and Michael Stephens from Manning, who had full confidence in my ability to write it from day one. His words of encouragement early on really motivated me and kept me motivated throughout the last year and a half.

I would also like to thank my initial development editor Andrew Warren, who helped me get my first chapter out the door, and Elesha Hyde, who took over from Andrew and worked with me all the way to the last chapter. Thank you for bearing with me, even though I'm a difficult person to deal with, as I tend to drop off the radar fairly regularly.

I would also like to thank Jeanne Boyarsky, who was the first reviewer to read and comment on my chapters while I was writing them. Jeanne and Elesha were instrumental in making the book as nice as it hopefully is. Without their comments, the book could never have received such good reviews from external reviewers and readers.

I'd like to thank my technical proofreader, Antonio Magnaghi, and of course all my external reviewers: Al Krinker, Alessandro Campeis, Alexander Myltsev, Csaba Sari, David DiMaria, Elias Rangel, Erisk Zelenka, Fabrizio Cucci, Jared Duncan, Keith Donaldson, Michael Bright, Paolo Antinori, Peter Perlepes, and Tiklu Ganguly. Their positive comments kept me going at times when I worried my writing was utterly awful and completely useless. On the other hand, their constructive criticism helped improve

sections that I'd quickly thrown together without enough effort. Thank you for pointing out the hard-to-understand sections and suggesting ways of improving the book. Also, thank you for asking the right questions, which made me realize I was wrong about two or three things in the initial versions of the manuscript.

I also need to thank readers who bought the early version of the book through Manning's MEAP program and voiced their comments in the online forum or reached out to me directly—especially Vimal Kansal, Paolo Patierno, and Roland Huß, who noticed quite a few inconsistencies and other mistakes. And I would like to thank everyone at Manning who has been involved in getting this book published. Before I finish, I also need to thank my colleague and high school friend Aleš Justin, who brought me to Red Hat, and my wonderful colleagues from the Cloud Enablement team. If I hadn't been at Red Hat or in the team, I wouldn't have been the one to write this book.

Lastly, I would like to thank my wife and my son, who were way too understanding and supportive over the last 18 months, while I was locked in my office instead of spending time with them.

Thank you all!

## *about this book*

---

*Kubernetes in Action* aims to make you a proficient user of Kubernetes. It teaches you virtually all the concepts you need to understand to effectively develop and run applications in a Kubernetes environment.

Before diving into Kubernetes, the book gives an overview of container technologies like Docker, including how to build containers, so that even readers who haven't used these technologies before can get up and running. It then slowly guides you through most of what you need to know about Kubernetes—from basic concepts to things hidden below the surface.

### ***Who should read this book***

The book focuses primarily on application developers, but it also provides an overview of managing applications from the operational perspective. It's meant for anyone interested in running and managing containerized applications on more than just a single server.

Both beginner and advanced software engineers who want to learn about container technologies and orchestrating multiple related containers at scale will gain the expertise necessary to develop, containerize, and run their applications in a Kubernetes environment.

No previous exposure to either container technologies or Kubernetes is required. The book explains the subject matter in a progressively detailed manner, and doesn't use any application source code that would be too hard for non-expert developers to understand.

Readers, however, should have at least a basic knowledge of programming, computer networking, and running basic commands in Linux, and an understanding of well-known computer protocols like HTTP.

### ***How this book is organized: a roadmap***

This book has three parts that cover 18 chapters.

Part 1 gives a short introduction to Docker and Kubernetes, how to set up a Kubernetes cluster, and how to run a simple application in it. It contains two chapters:

- Chapter 1 explains what Kubernetes is, how it came to be, and how it helps to solve today's problems of managing applications at scale.
- Chapter 2 is a hands-on tutorial on how to build a container image and run it in a Kubernetes cluster. It also explains how to run a local single-node Kubernetes cluster and a proper multi-node cluster in the cloud.

Part 2 introduces the key concepts you must understand to run applications in Kubernetes. The chapters are as follows:

- Chapter 3 introduces the fundamental building block in Kubernetes—the pod—and explains how to organize pods and other Kubernetes objects through labels.
- Chapter 4 teaches you how Kubernetes keeps applications healthy by automatically restarting containers. It also shows how to properly run managed pods, horizontally scale them, make them resistant to failures of cluster nodes, and run them at a predefined time in the future or periodically.
- Chapter 5 shows how pods can expose the service they provide to clients running both inside and outside the cluster. It also shows how pods running in the cluster can discover and access services, regardless of whether they live in or out of the cluster.
- Chapter 6 explains how multiple containers running in the same pod can share files and how you can manage persistent storage and make it accessible to pods.
- Chapter 7 shows how to pass configuration data and sensitive information like credentials to apps running inside pods.
- Chapter 8 describes how applications can get information about the Kubernetes environment they're running in and how they can talk to Kubernetes to alter the state of the cluster.
- Chapter 9 introduces the concept of a Deployment and explains the proper way of running and updating applications in a Kubernetes environment.
- Chapter 10 introduces a dedicated way of running stateful applications, which usually require a stable identity and state.

Part 3 dives deep into the internals of a Kubernetes cluster, introduces some additional concepts, and reviews everything you've learned in the first two parts from a higher perspective. This is the last group of chapters:

- Chapter 11 goes beneath the surface of Kubernetes and explains all the components that make up a Kubernetes cluster and what each of them does. It also

explains how pods communicate through the network and how services perform load balancing across multiple pods.

- Chapter 12 explains how to secure your Kubernetes API server, and by extension the cluster, using authentication and authorization.
- Chapter 13 teaches you how pods can access the node's resources and how a cluster administrator can prevent pods from doing that.
- Chapter 14 dives into constraining the computational resources each application is allowed to consume, configuring the applications' Quality of Service guarantees, and monitoring the resource usage of individual applications. It also teaches you how to prevent users from consuming too many resources.
- Chapter 15 discusses how Kubernetes can be configured to automatically scale the number of running replicas of your application, and how it can also increase the size of your cluster when your current number of cluster nodes can't accept any additional applications.
- Chapter 16 shows how to ensure pods are scheduled only to certain nodes or how to prevent them from being scheduled to others. It also shows how to make sure pods are scheduled together or how to prevent that from happening.
- Chapter 17 teaches you how you should develop your applications to make them good citizens of your cluster. It also gives you a few pointers on how to set up your development and testing workflows to reduce friction during development.
- Chapter 18 shows you how you can extend Kubernetes with your own custom objects and how others have done it and created enterprise-class application platforms.

As you progress through these chapters, you'll not only learn about the individual Kubernetes building blocks, but also progressively improve your knowledge of using the `kubectl` command-line tool.

### **About the code**

While this book doesn't contain a lot of actual source code, it does contain a lot of manifests of Kubernetes resources in YAML format and shell commands along with their outputs. All of this is formatted in a fixed-width font like this to separate it from ordinary text.

Shell commands are mostly **in bold**, to clearly separate them from their output, but sometimes only the most important parts of the command or parts of the command's output are in bold for emphasis. In most cases, the command output has been reformatted to make it fit into the limited space in the book. Also, because the Kubernetes CLI tool `kubectl` is constantly evolving, newer versions may print out more information than what's shown in the book. Don't be confused if they don't match exactly.

Listings sometimes include a line-continuation marker (**➞**) to show that a line of text wraps to the next line. They also include annotations, which highlight and explain the most important parts.

Within text paragraphs, some very common elements such as Pod, ReplicationController, ReplicaSet, DaemonSet, and so forth are set in regular font to avoid overproliferation of code font and help readability. In some places, “Pod” is capitalized to refer to the Pod resource, and lowercased to refer to the actual group of running containers.

All the samples in the book have been tested with Kubernetes version 1.8 running in Google Kubernetes Engine and in a local cluster run with Minikube. The complete source code and YAML manifests can be found at <https://github.com/luksa/kubernetes-in-action> or downloaded from the publisher’s website at [www.manning.com/books/kubernetes-in-action](http://www.manning.com/books/kubernetes-in-action).

## Book forum

Purchase of *Kubernetes in Action* includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the author and from other users. To access the forum, go to <https://forums.manning.com/forums/kubernetes-in-action>. You can also learn more about Manning’s forums and the rules of conduct at <https://forums.manning.com/forums/about>.

Manning’s commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the author can take place. It is not a commitment to any specific amount of participation on the part of the author, whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking the author some challenging questions lest his interest stray! The forum and the archives of previous discussions will be accessible from the publisher’s website as long as the book is in print.

## Other online resources

You can find a wide range of additional Kubernetes resources at the following locations:

- The Kubernetes website at <https://kubernetes.io>
- The Kubernetes Blog, which regularly posts interesting info (<http://blog.kubernetes.io>)
- The Kubernetes community’s Slack channel at <http://slack.k8s.io>
- The Kubernetes and Cloud Native Computing Foundation’s YouTube channels:
  - [https://www.youtube.com/channel/UCZ2bu0qutTOM0tHYa\\_jkIwg](https://www.youtube.com/channel/UCZ2bu0qutTOM0tHYa_jkIwg)
  - <https://www.youtube.com/channel/UCvqbFHWn-nwaWPjPUKpvTA>

To gain a deeper understanding of individual topics or even to help contribute to Kubernetes, you can also check out any of the Kubernetes Special Interest Groups (SIGs) at [https://github.com/kubernetes/kubernetes/wiki/Special-Interest-Groups-\(SIGs\)](https://github.com/kubernetes/kubernetes/wiki/Special-Interest-Groups-(SIGs)).

And, finally, as Kubernetes is open source, there’s a wealth of information available in the Kubernetes source code itself. You’ll find it at <https://github.com/kubernetes/kubernetes> and related repositories.

## *about the author*

---



Marko Lukša is a software engineer with more than 20 years of professional experience developing everything from simple web applications to full ERP systems, frameworks, and middleware software. He took his first steps in programming back in 1985, at the age of six, on a second-hand ZX Spectrum computer his father had bought for him. In primary school, he was the national champion in the Logo programming competition and attended summer coding camps, where he learned to program in Pascal. Since then, he has developed software in a wide range of programming languages.

In high school, he started building dynamic websites when the web was still relatively young. He then moved on to developing software for the healthcare and telecommunications industries at a local company, while studying computer science at the University of Ljubljana, Slovenia. Eventually, he ended up working for Red Hat, initially developing an open source implementation of the Google App Engine API, which utilized Red Hat's JBoss middleware products underneath. He also worked in or contributed to projects like CDI/Weld, Infinispan/JBoss Data-Grid, and others.

Since late 2014, he has been part of Red Hat's Cloud Enablement team, where his responsibilities include staying up-to-date on new developments in Kubernetes and related technologies and ensuring the company's middleware software utilizes the features of Kubernetes and OpenShift to their full potential.

## *about the cover illustration*

---

The figure on the cover of *Kubernetes in Action* is a “Member of the Divan,” the Turkish Council of State or governing body. The illustration is taken from a collection of costumes of the Ottoman Empire published on January 1, 1802, by William Miller of Old Bond Street, London. The title page is missing from the collection and we have been unable to track it down to date. The book’s table of contents identifies the figures in both English and French, and each illustration bears the names of two artists who worked on it, both of whom would no doubt be surprised to find their art gracing the front cover of a computer programming book ... 200 years later.

The collection was purchased by a Manning editor at an antiquarian flea market in the “Garage” on West 26th Street in Manhattan. The seller was an American based in Ankara, Turkey, and the transaction took place just as he was packing up his stand for the day. The Manning editor didn’t have on his person the substantial amount of cash that was required for the purchase, and a credit card and check were both politely turned down. With the seller flying back to Ankara that evening, the situation was getting hopeless. What was the solution? It turned out to be nothing more than an old-fashioned verbal agreement sealed with a handshake. The seller proposed that the money be transferred to him by wire, and the editor walked out with the bank information on a piece of paper and the portfolio of images under his arm. Needless to say, we transferred the funds the next day, and we remain grateful and impressed by this unknown person’s trust in one of us. It recalls something that might have happened a long time ago. We at Manning celebrate the inventiveness, the initiative, and, yes, the fun of the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought back to life by the pictures from this collection.



# *Introducing Kubernetes*

---

## **This chapter covers**

- Understanding how software development and deployment has changed over recent years
- Isolating applications and reducing environment differences using containers
- Understanding how containers and Docker are used by Kubernetes
- Making developers' and sysadmins' jobs easier with Kubernetes

Years ago, most software applications were big monoliths, running either as a single process or as a small number of processes spread across a handful of servers. These legacy systems are still widespread today. They have slow release cycles and are updated relatively infrequently. At the end of every release cycle, developers package up the whole system and hand it over to the ops team, who then deploys and monitors it. In case of hardware failures, the ops team manually migrates it to the remaining healthy servers.

Today, these big monolithic legacy applications are slowly being broken down into smaller, independently running components called microservices. Because

microservices are decoupled from each other, they can be developed, deployed, updated, and scaled individually. This enables you to change components quickly and as often as necessary to keep up with today's rapidly changing business requirements.

But with bigger numbers of deployable components and increasingly larger datacenters, it becomes increasingly difficult to configure, manage, and keep the whole system running smoothly. It's much harder to figure out where to put each of those components to achieve high resource utilization and thereby keep the hardware costs down. Doing all this manually is hard work. We need automation, which includes automatic scheduling of those components to our servers, automatic configuration, supervision, and failure-handling. This is where Kubernetes comes in.

Kubernetes enables developers to deploy their applications themselves and as often as they want, without requiring any assistance from the operations (ops) team. But Kubernetes doesn't benefit only developers. It also helps the ops team by automatically monitoring and rescheduling those apps in the event of a hardware failure. The focus for system administrators (sysadmins) shifts from supervising individual apps to mostly supervising and managing Kubernetes and the rest of the infrastructure, while Kubernetes itself takes care of the apps.

**NOTE** *Kubernetes* is Greek for pilot or helmsman (the person holding the ship's steering wheel). People pronounce Kubernetes in a few different ways. Many pronounce it as *Koo-ber-nay-tace*, while others pronounce it more like *Koo-ber-netties*. No matter which form you use, people will understand what you mean.

Kubernetes abstracts away the hardware infrastructure and exposes your whole datacenter as a single enormous computational resource. It allows you to deploy and run your software components without having to know about the actual servers underneath. When deploying a multi-component application through Kubernetes, it selects a server for each component, deploys it, and enables it to easily find and communicate with all the other components of your application.

This makes Kubernetes great for most on-premises datacenters, but where it starts to shine is when it's used in the largest datacenters, such as the ones built and operated by cloud providers. Kubernetes allows them to offer developers a simple platform for deploying and running any type of application, while not requiring the cloud provider's own sysadmins to know anything about the tens of thousands of apps running on their hardware.

With more and more big companies accepting the Kubernetes model as the best way to run apps, it's becoming the standard way of running distributed apps both in the cloud, as well as on local on-premises infrastructure.

## **1.1** *Understanding the need for a system like Kubernetes*

Before you start getting to know Kubernetes in detail, let's take a quick look at how the development and deployment of applications has changed in recent years. This change is both a consequence of splitting big monolithic apps into smaller microservices

and of the changes in the infrastructure that runs those apps. Understanding these changes will help you better see the benefits of using Kubernetes and container technologies such as Docker.

### 1.1.1 Moving from monolithic apps to microservices

Monolithic applications consist of components that are all tightly coupled together and have to be developed, deployed, and managed as one entity, because they all run as a single OS process. Changes to one part of the application require a redeployment of the whole application, and over time the lack of hard boundaries between the parts results in the increase of complexity and consequential deterioration of the quality of the whole system because of the unconstrained growth of inter-dependencies between these parts.

Running a monolithic application usually requires a small number of powerful servers that can provide enough resources for running the application. To deal with increasing loads on the system, you then either have to vertically scale the servers (also known as scaling up) by adding more CPUs, memory, and other server components, or scale the whole system horizontally, by setting up additional servers and running multiple copies (or replicas) of an application (scaling out). While scaling up usually doesn't require any changes to the app, it gets expensive relatively quickly and in practice always has an upper limit. Scaling out, on the other hand, is relatively cheap hardware-wise, but may require big changes in the application code and isn't always possible—certain parts of an application are extremely hard or next to impossible to scale horizontally (relational databases, for example). If any part of a monolithic application isn't scalable, the whole application becomes unscalable, unless you can split up the monolith somehow.

#### SPLITTING APPS INTO MICROSERVICES

These and other problems have forced us to start splitting complex monolithic applications into smaller independently deployable components called microservices. Each microservice runs as an independent process (see figure 1.1) and communicates with other microservices through simple, well-defined interfaces (APIs).

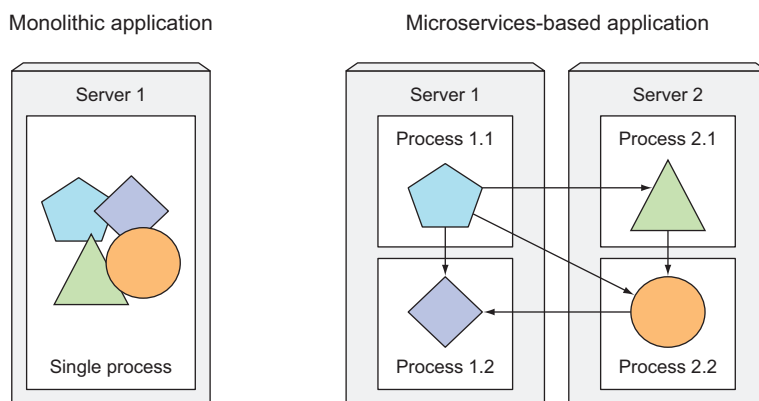


Figure 1.1 Components inside a monolithic application vs. standalone microservices

Microservices communicate through synchronous protocols such as HTTP, over which they usually expose RESTful (REpresentational State Transfer) APIs, or through asynchronous protocols such as AMQP (Advanced Message Queueing Protocol). These protocols are simple, well understood by most developers, and not tied to any specific programming language. Each microservice can be written in the language that's most appropriate for implementing that specific microservice.

Because each microservice is a standalone process with a relatively static external API, it's possible to develop and deploy each microservice separately. A change to one of them doesn't require changes or redeployment of any other service, provided that the API doesn't change or changes only in a backward-compatible way.

### SCALING MICROSERVICES

Scaling microservices, unlike monolithic systems, where you need to scale the system as a whole, is done on a per-service basis, which means you have the option of scaling only those services that require more resources, while leaving others at their original scale. Figure 1.2 shows an example. Certain components are replicated and run as multiple processes deployed on different servers, while others run as a single application process. When a monolithic application can't be scaled out because one of its parts is unscalable, splitting the app into microservices allows you to horizontally scale the parts that allow scaling out, and scale the parts that don't, vertically instead of horizontally.

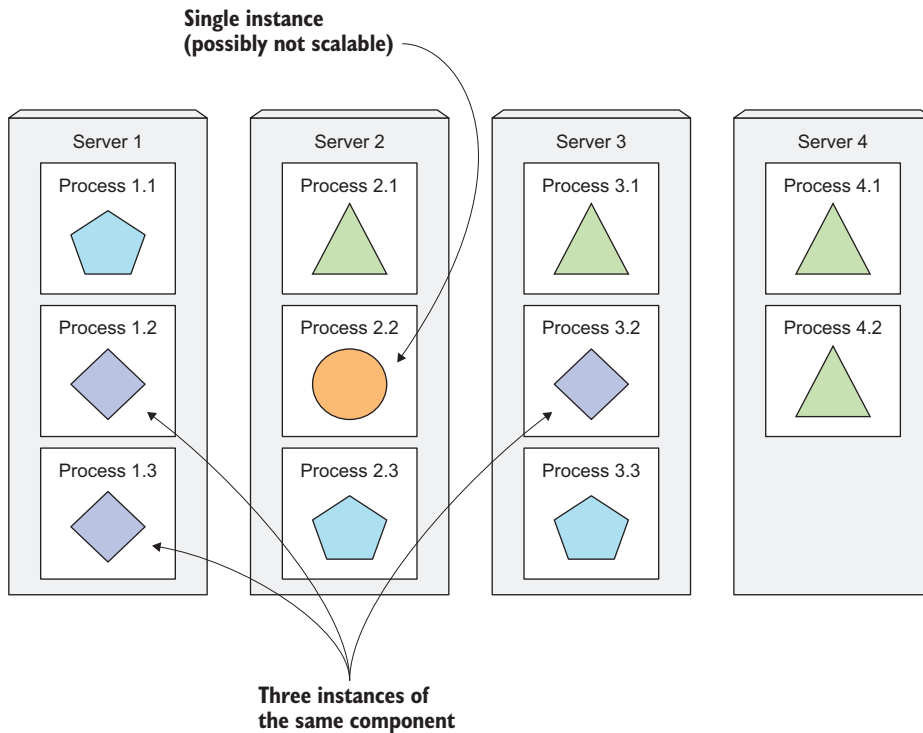


Figure 1.2 Each microservice can be scaled individually.

## DEPLOYING MICROSERVICES

As always, microservices also have drawbacks. When your system consists of only a small number of deployable components, managing those components is easy. It's trivial to decide where to deploy each component, because there aren't that many choices. When the number of those components increases, deployment-related decisions become increasingly difficult because not only does the number of deployment combinations increase, but the number of inter-dependencies between the components increases by an even greater factor.

Microservices perform their work together as a team, so they need to find and talk to each other. When deploying them, someone or something needs to configure all of them properly to enable them to work together as a single system. With increasing numbers of microservices, this becomes tedious and error-prone, especially when you consider what the ops/sysadmin teams need to do when a server fails.

Microservices also bring other problems, such as making it hard to debug and trace execution calls, because they span multiple processes and machines. Luckily, these problems are now being addressed with distributed tracing systems such as Zipkin.

## UNDERSTANDING THE DIVERGENCE OF ENVIRONMENT REQUIREMENTS

As I've already mentioned, components in a microservices architecture aren't only deployed independently, but are also developed that way. Because of their independence and the fact that it's common to have separate teams developing each component, nothing impedes each team from using different libraries and replacing them whenever the need arises. The divergence of dependencies between application components, like the one shown in figure 1.3, where applications require different versions of the same libraries, is inevitable.

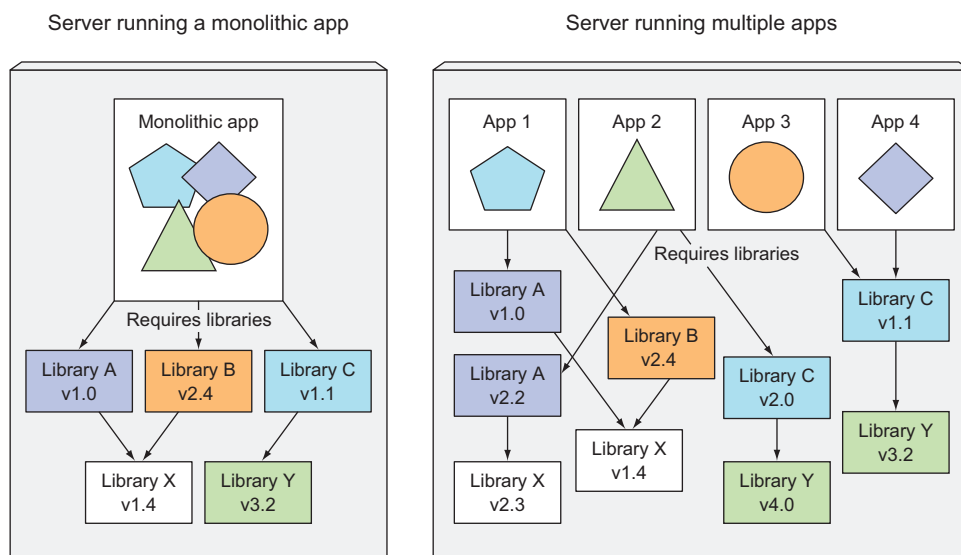


Figure 1.3 Multiple applications running on the same host may have conflicting dependencies.

Deploying dynamically linked applications that require different versions of shared libraries, and/or require other environment specifics, can quickly become a nightmare for the ops team who deploys and manages them on production servers. The bigger the number of components you need to deploy on the same host, the harder it will be to manage all their dependencies to satisfy all their requirements.

### **1.1.2** *Providing a consistent environment to applications*

Regardless of how many individual components you're developing and deploying, one of the biggest problems that developers and operations teams always have to deal with is the differences in the environments they run their apps in. Not only is there a huge difference between development and production environments, differences even exist between individual production machines. Another unavoidable fact is that the environment of a single production machine will change over time.

These differences range from hardware to the operating system to the libraries that are available on each machine. Production environments are managed by the operations team, while developers often take care of their development laptops on their own. The difference is how much these two groups of people know about system administration, and this understandably leads to relatively big differences between those two systems, not to mention that system administrators give much more emphasis on keeping the system up to date with the latest security patches, while a lot of developers don't care about that as much.

Also, production systems can run applications from multiple developers or development teams, which isn't necessarily true for developers' computers. A production system must provide the proper environment to all applications it hosts, even though they may require different, even conflicting, versions of libraries.

To reduce the number of problems that only show up in production, it would be ideal if applications could run in the exact same environment during development and in production so they have the exact same operating system, libraries, system configuration, networking environment, and everything else. You also don't want this environment to change too much over time, if at all. Also, if possible, you want the ability to add applications to the same server without affecting any of the existing applications on that server.

### **1.1.3** *Moving to continuous delivery: DevOps and NoOps*

In the last few years, we've also seen a shift in the whole application development process and how applications are taken care of in production. In the past, the development team's job was to create the application and hand it off to the operations team, who then deployed it, tended to it, and kept it running. But now, organizations are realizing it's better to have the same team that develops the application also take part in deploying it and taking care of it over its whole lifetime. This means the developer, QA, and operations teams now need to collaborate throughout the whole process. This practice is called DevOps.

### UNDERSTANDING THE BENEFITS

Having the developers more involved in running the application in production leads to them having a better understanding of both the users' needs and issues and the problems faced by the ops team while maintaining the app. Application developers are now also much more inclined to give users the app earlier and then use their feedback to steer further development of the app.

To release newer versions of applications more often, you need to streamline the deployment process. Ideally, you want developers to deploy the applications themselves without having to wait for the ops people. But deploying an application often requires an understanding of the underlying infrastructure and the organization of the hardware in the datacenter. Developers don't always know those details and, most of the time, don't even want to know about them.

### LETTING DEVELOPERS AND SYSADMINS DO WHAT THEY DO BEST

Even though developers and system administrators both work toward achieving the same goal of running a successful software application as a service to its customers, they have different individual goals and motivating factors. Developers love creating new features and improving the user experience. They don't normally want to be the ones making sure that the underlying operating system is up to date with all the security patches and things like that. They prefer to leave that up to the system administrators.

The ops team is in charge of the production deployments and the hardware infrastructure they run on. They care about system security, utilization, and other aspects that aren't a high priority for developers. The ops people don't want to deal with the implicit interdependencies of all the application components and don't want to think about how changes to either the underlying operating system or the infrastructure can affect the operation of the application as a whole, but they must.

Ideally, you want the developers to deploy applications themselves without knowing anything about the hardware infrastructure and without dealing with the ops team. This is referred to as *NoOps*. Obviously, you still need someone to take care of the hardware infrastructure, but ideally, without having to deal with peculiarities of each application running on it.

As you'll see, Kubernetes enables us to achieve all of this. By abstracting away the actual hardware and exposing it as a single platform for deploying and running apps, it allows developers to configure and deploy their applications without any help from the sysadmins and allows the sysadmins to focus on keeping the underlying infrastructure up and running, while not having to know anything about the actual applications running on top of it.

## 1.2 Introducing container technologies

In section 1.1 I presented a non-comprehensive list of problems facing today's development and ops teams. While you have many ways of dealing with them, this book will focus on how they're solved with Kubernetes.

Kubernetes uses Linux container technologies to provide isolation of running applications, so before we dig into Kubernetes itself, you need to become familiar with the basics of containers to understand what Kubernetes does itself, and what it offloads to container technologies like *Docker* or *rkt* (pronounced “rock-it”).

### 1.2.1 *Understanding what containers are*

In section 1.1.1 we saw how different software components running on the same machine will require different, possibly conflicting, versions of dependent libraries or have other different environment requirements in general.

When an application is composed of only smaller numbers of large components, it’s completely acceptable to give a dedicated Virtual Machine (VM) to each component and isolate their environments by providing each of them with their own operating system instance. But when these components start getting smaller and their numbers start to grow, you can’t give each of them their own VM if you don’t want to waste hardware resources and keep your hardware costs down. But it’s not only about wasting hardware resources. Because each VM usually needs to be configured and managed individually, rising numbers of VMs also lead to wasting human resources, because they increase the system administrators’ workload considerably.

#### **ISOLATING COMPONENTS WITH LINUX CONTAINER TECHNOLOGIES**

Instead of using virtual machines to isolate the environments of each microservice (or software processes in general), developers are turning to Linux container technologies. They allow you to run multiple services on the same host machine, while not only exposing a different environment to each of them, but also isolating them from each other, similarly to VMs, but with much less overhead.

A process running in a container runs inside the host’s operating system, like all the other processes (unlike VMs, where processes run in separate operating systems). But the process in the container is still isolated from other processes. To the process itself, it looks like it’s the only one running on the machine and in its operating system.

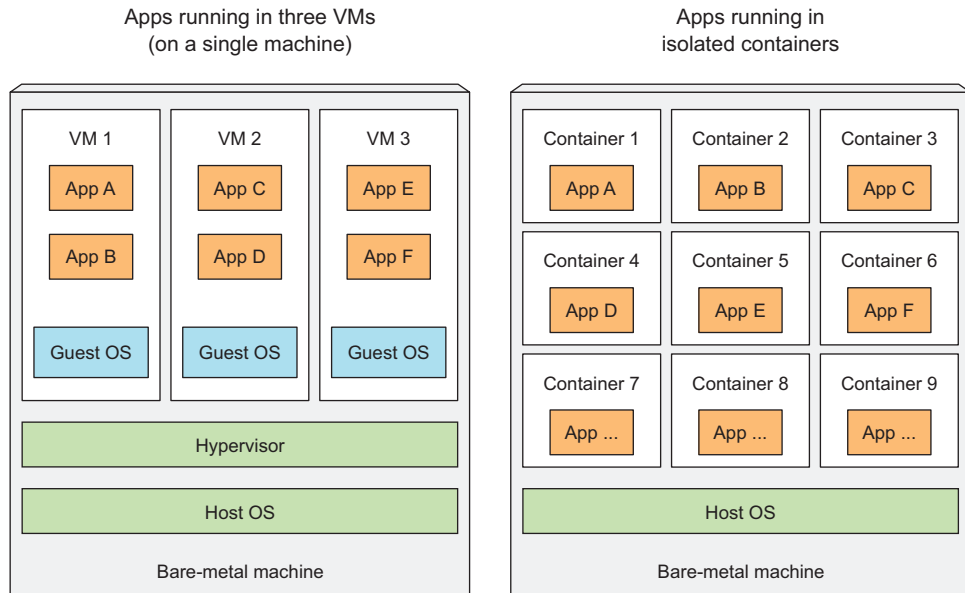
#### **COMPARING VIRTUAL MACHINES TO CONTAINERS**

Compared to VMs, containers are much more lightweight, which allows you to run higher numbers of software components on the same hardware, mainly because each VM needs to run its own set of system processes, which requires additional compute resources in addition to those consumed by the component’s own process. A container, on the other hand, is nothing more than a single isolated process running in the host OS, consuming only the resources that the app consumes and without the overhead of any additional processes.

Because of the overhead of VMs, you often end up grouping multiple applications into each VM because you don’t have enough resources to dedicate a whole VM to each app. When using containers, you can (and should) have one container for each



application, as shown in figure 1.4. The end-result is that you can fit many more applications on the same bare-metal machine.



**Figure 1.4** Using VMs to isolate groups of applications vs. isolating individual apps with containers

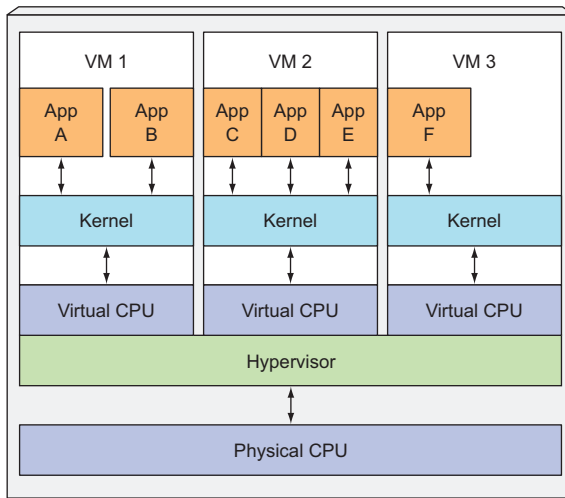
When you run three VMs on a host, you have three completely separate operating systems running on and sharing the same bare-metal hardware. Underneath those VMs is the host's OS and a hypervisor, which divides the physical hardware resources into smaller sets of virtual resources that can be used by the operating system inside each VM. Applications running inside those VMs perform system calls to the guest OS' kernel in the VM, and the kernel then performs x86 instructions on the host's physical CPU through the hypervisor.

**NOTE** Two types of hypervisors exist. Type 1 hypervisors don't use a host OS, while Type 2 do.

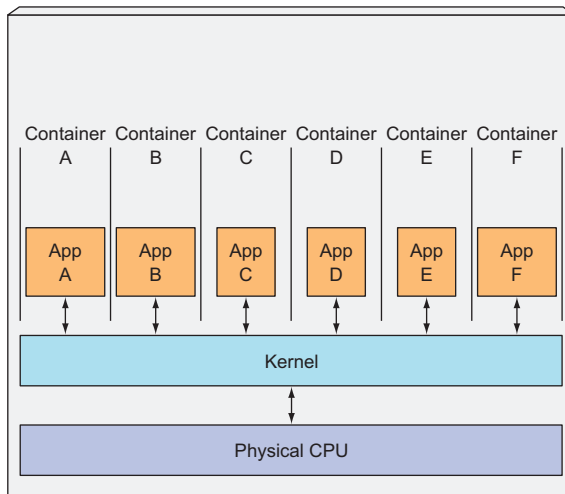
Containers, on the other hand, all perform system calls on the exact same kernel running in the host OS. This single kernel is the only one performing x86 instructions on the host's CPU. The CPU doesn't need to do any kind of virtualization the way it does with VMs (see figure 1.5).

The main benefit of virtual machines is the full isolation they provide, because each VM runs its own Linux kernel, while containers all call out to the same kernel, which can clearly pose a security risk. If you have a limited amount of hardware resources, VMs may only be an option when you have a small number of processes that

Apps running in multiple VMs



Apps running in isolated containers



**Figure 1.5** The difference between how apps in VMs use the CPU vs. how they use them in containers

you want to isolate. To run greater numbers of isolated processes on the same machine, containers are a much better choice because of their low overhead. Remember, each VM runs its own set of system services, while containers don't, because they all run in the same OS. That also means that to run a container, nothing needs to be booted up, as is the case in VMs. A process run in a container starts up immediately.

## INTRODUCING THE MECHANISMS THAT MAKE CONTAINER ISOLATION POSSIBLE

By this point, you're probably wondering how exactly containers can isolate processes if they're running on the same operating system. Two mechanisms make this possible. The first one, *Linux Namespaces*, makes sure each process sees its own personal view of the system (files, processes, network interfaces, hostname, and so on). The second one is *Linux Control Groups (cgroups)*, which limit the amount of resources the process can consume (CPU, memory, network bandwidth, and so on).

### ISOLATING PROCESSES WITH LINUX NAMESPACES

By default, each Linux system initially has one single namespace. All system resources, such as filesystems, process IDs, user IDs, network interfaces, and others, belong to the single namespace. But you can create additional namespaces and organize resources across them. When running a process, you run it inside one of those namespaces. The process will only see resources that are inside the same namespace. Well, multiple kinds of namespaces exist, so a process doesn't belong to one namespace, but to one namespace of each kind.

The following kinds of namespaces exist:

- Mount (mnt)
- Process ID (pid)
- Network (net)
- Inter-process communication (ipc)
- UTS
- User ID (user)

Each namespace kind is used to isolate a certain group of resources. For example, the UTS namespace determines what hostname and domain name the process running inside that namespace sees. By assigning two different UTS namespaces to a pair of processes, you can make them see different local hostnames. In other words, to the two processes, it will appear as though they are running on two different machines (at least as far as the hostname is concerned).

Likewise, what Network namespace a process belongs to determines which network interfaces the application running inside the process sees. Each network interface belongs to exactly one namespace, but can be moved from one namespace to another. Each container uses its own Network namespace, and therefore each container sees its own set of network interfaces.

This should give you a basic idea of how namespaces are used to isolate applications running in containers from each other.

### LIMITING RESOURCES AVAILABLE TO A PROCESS

The other half of container isolation deals with limiting the amount of system resources a container can consume. This is achieved with *cgroups*, a Linux kernel feature that limits the resource usage of a process (or a group of processes). A process can't use more than the configured amount of CPU, memory, network bandwidth,

and so on. This way, processes cannot hog resources reserved for other processes, which is similar to when each process runs on a separate machine.

### **1.2.2** *Introducing the Docker container platform*

While container technologies have been around for a long time, they've become more widely known with the rise of the Docker container platform. Docker was the first container system that made containers easily portable across different machines. It simplified the process of packaging up not only the application but also all its libraries and other dependencies, even the whole OS file system, into a simple, portable package that can be used to provision the application to any other machine running Docker.

When you run an application packaged with Docker, it sees the exact filesystem contents that you've bundled with it. It sees the same files whether it's running on your development machine or a production machine, even if the production server is running a completely different Linux OS. The application won't see anything from the server it's running on, so it doesn't matter if the server has a completely different set of installed libraries compared to your development machine.

For example, if you've packaged up your application with the files of the whole Red Hat Enterprise Linux (RHEL) operating system, the application will believe it's running inside RHEL, both when you run it on your development computer that runs Fedora and when you run it on a server running Debian or some other Linux distribution. Only the kernel may be different.

This is similar to creating a VM image by installing an operating system into a VM, installing the app inside it, and then distributing the whole VM image around and running it. Docker achieves the same effect, but instead of using VMs to achieve app isolation, it uses Linux container technologies mentioned in the previous section to provide (almost) the same level of isolation that VMs do. Instead of using big monolithic VM images, it uses container images, which are usually smaller.

A big difference between Docker-based container images and VM images is that container images are composed of layers, which can be shared and reused across multiple images. This means only certain layers of an image need to be downloaded if the other layers were already downloaded previously when running a different container image that also contains the same layers.

#### **UNDERSTANDING DOCKER CONCEPTS**

Docker is a platform for packaging, distributing, and running applications. As we've already stated, it allows you to package your application together with its whole environment. This can be either a few libraries that the app requires or even all the files that are usually available on the filesystem of an installed operating system. Docker makes it possible to transfer this package to a central repository from which it can then be transferred to any computer running Docker and executed there (for the most part, but not always, as we'll soon explain).

Three main concepts in Docker comprise this scenario:

- **Images**—A Docker-based container image is something you package your application and its environment into. It contains the filesystem that will be available to the application and other metadata, such as the path to the executable that should be executed when the image is run.
- **Registries**—A Docker Registry is a repository that stores your Docker images and facilitates easy sharing of those images between different people and computers. When you build your image, you can either run it on the computer you’ve built it on, or you can *push* (upload) the image to a registry and then *pull* (download) it on another computer and run it there. Certain registries are public, allowing anyone to pull images from it, while others are private, only accessible to certain people or machines.
- **Containers**—A Docker-based container is a regular Linux container created from a Docker-based container image. A running container is a process running on the host running Docker, but it’s completely isolated from both the host and all other processes running on it. The process is also resource-constrained, meaning it can only access and use the amount of resources (CPU, RAM, and so on) that are allocated to it.

#### BUILDING, DISTRIBUTING, AND RUNNING A DOCKER IMAGE

Figure 1.6 shows all three concepts and how they relate to each other. The developer first builds an image and then pushes it to a registry. The image is thus available to anyone who can access the registry. They can then pull the image to any other machine running Docker and run the image. Docker creates an isolated container based on the image and runs the binary executable specified as part of the image.

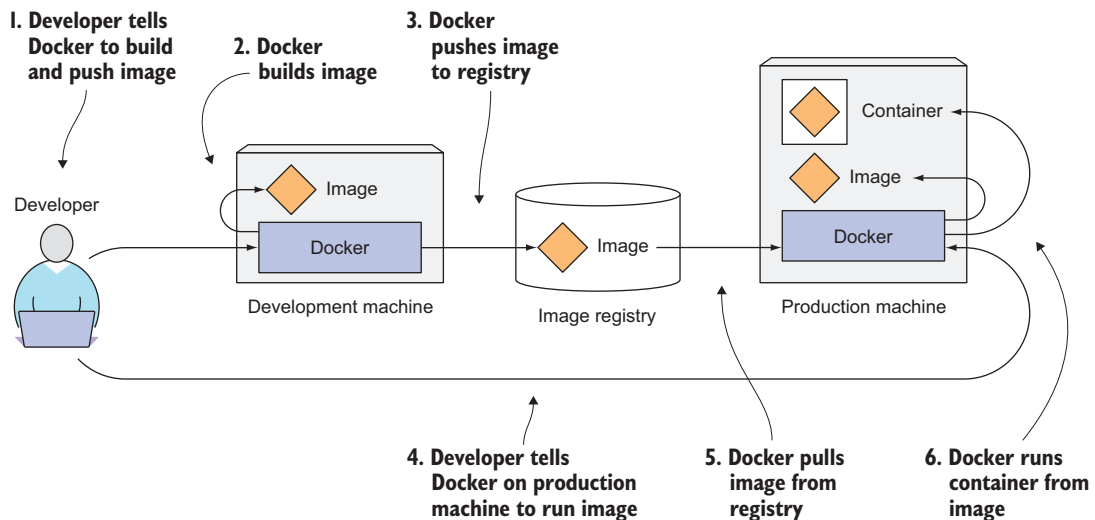
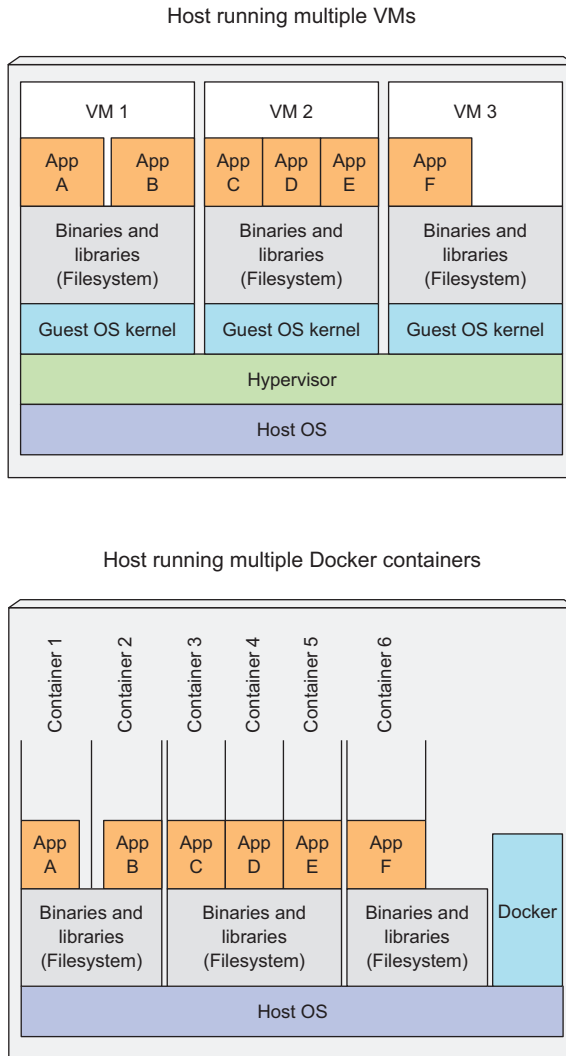


Figure 1.6 Docker images, registries, and containers

**COMPARING VIRTUAL MACHINES AND DOCKER CONTAINERS**

I've explained how Linux containers are generally like virtual machines, but much more lightweight. Now let's look at how Docker containers specifically compare to virtual machines (and how Docker images compare to VM images). Figure 1.7 again shows the same six applications running both in VMs and as Docker containers.



**Figure 1.7** Running six apps on three VMs vs. running them in Docker containers

You'll notice that apps A and B have access to the same binaries and libraries both when running in a VM and when running as two separate containers. In the VM, this is obvious, because both apps see the same filesystem (that of the VM). But we said

that each container has its own isolated filesystem. How can both app A and app B share the same files?

#### UNDERSTANDING IMAGE LAYERS

I've already said that Docker images are composed of layers. Different images can contain the exact same layers because every Docker image is built on top of another image and two different images can both use the same parent image as their base. This speeds up the distribution of images across the network, because layers that have already been transferred as part of the first image don't need to be transferred again when transferring the other image.

But layers don't only make distribution more efficient, they also help reduce the storage footprint of images. Each layer is only stored once. Two containers created from two images based on the same base layers can therefore read the same files, but if one of them writes over those files, the other one doesn't see those changes. Therefore, even if they share files, they're still isolated from each other. This works because container image layers are read-only. When a container is run, a new writable layer is created on top of the layers in the image. When the process in the container writes to a file located in one of the underlying layers, a copy of the whole file is created in the top-most layer and the process writes to the copy.

#### UNDERSTANDING THE PORTABILITY LIMITATIONS OF CONTAINER IMAGES

In theory, a container image can be run on any Linux machine running Docker, but one small caveat exists—one related to the fact that all containers running on a host use the host's Linux kernel. If a containerized application requires a specific kernel version, it may not work on every machine. If a machine runs a different version of the Linux kernel or doesn't have the same kernel modules available, the app can't run on it.

While containers are much more lightweight compared to VMs, they impose certain constraints on the apps running inside them. VMs have no such constraints, because each VM runs its own kernel.

And it's not only about the kernel. It should also be clear that a containerized app built for a specific hardware architecture can only run on other machines that have the same architecture. You can't containerize an application built for the x86 architecture and expect it to run on an ARM-based machine because it also runs Docker. You still need a VM for that.

### 1.2.3 Introducing *rkt*—an alternative to Docker

Docker was the first container platform that made containers mainstream. I hope I've made it clear that Docker itself doesn't provide process isolation. The actual isolation of containers is done at the Linux kernel level using kernel features such as Linux Namespaces and cgroups. Docker only makes it easy to use those features.

After the success of Docker, the Open Container Initiative (OCI) was born to create open industry standards around container formats and runtime. Docker is part of that initiative, as is *rkt* (pronounced "rock-it"), which is another Linux container engine.

Like Docker, rkt is a platform for running containers. It puts a strong emphasis on security, composability, and conforming to open standards. It uses the OCI container image format and can even run regular Docker container images.

This book focuses on using Docker as the container runtime for Kubernetes, because it was initially the only one supported by Kubernetes. Recently, Kubernetes has also started supporting rkt, as well as others, as the container runtime.

The reason I mention rkt at this point is so you don't make the mistake of thinking Kubernetes is a container orchestration system made specifically for Docker-based containers. In fact, over the course of this book, you'll realize that the essence of Kubernetes isn't orchestrating containers. It's much more. Containers happen to be the best way to run apps on different cluster nodes. With that in mind, let's finally dive into the core of what this book is all about—Kubernetes.

## 1.3 *Introducing Kubernetes*

We've already shown that as the number of deployable application components in your system grows, it becomes harder to manage them all. Google was probably the first company that realized it needed a much better way of deploying and managing their software components and their infrastructure to scale globally. It's one of only a few companies in the world that runs hundreds of thousands of servers and has had to deal with managing deployments on such a massive scale. This has forced them to develop solutions for making the development and deployment of thousands of software components manageable and cost-efficient.

### 1.3.1 *Understanding its origins*

Through the years, Google developed an internal system called *Borg* (and later a new system called *Omega*), that helped both application developers and system administrators manage those thousands of applications and services. In addition to simplifying the development and management, it also helped them achieve a much higher utilization of their infrastructure, which is important when your organization is that large. When you run hundreds of thousands of machines, even tiny improvements in utilization mean savings in the millions of dollars, so the incentives for developing such a system are clear.

After having kept Borg and Omega secret for a whole decade, in 2014 Google introduced Kubernetes, an open-source system based on the experience gained through Borg, Omega, and other internal Google systems.

### 1.3.2 *Looking at Kubernetes from the top of a mountain*

Kubernetes is a software system that allows you to easily deploy and manage containerized applications on top of it. It relies on the features of Linux containers to run heterogeneous applications without having to know any internal details of these applications and without having to manually deploy these applications on each host. Because these apps run in containers, they don't affect other apps running on the



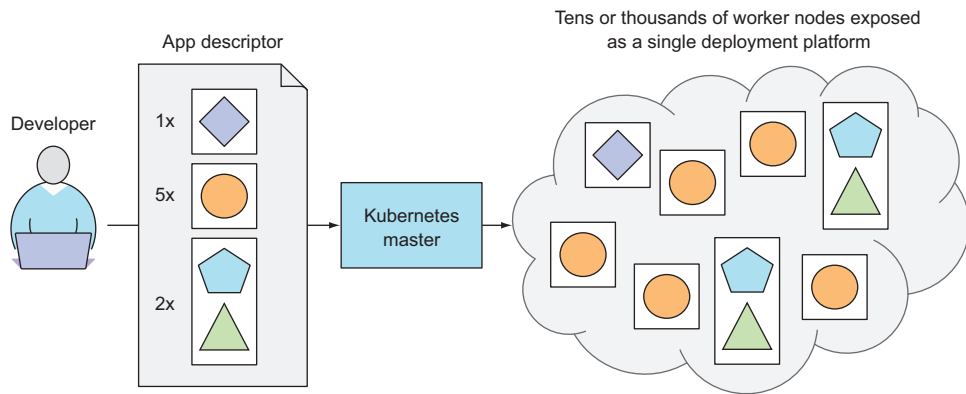
same server, which is critical when you run applications for completely different organizations on the same hardware. This is of paramount importance for cloud providers, because they strive for the best possible utilization of their hardware while still having to maintain complete isolation of hosted applications.

Kubernetes enables you to run your software applications on thousands of computer nodes as if all those nodes were a single, enormous computer. It abstracts away the underlying infrastructure and, by doing so, simplifies development, deployment, and management for both development and the operations teams.

Deploying applications through Kubernetes is always the same, whether your cluster contains only a couple of nodes or thousands of them. The size of the cluster makes no difference at all. Additional cluster nodes simply represent an additional amount of resources available to deployed apps.

#### UNDERSTANDING THE CORE OF WHAT KUBERNETES DOES

Figure 1.8 shows the simplest possible view of a Kubernetes system. The system is composed of a master node and any number of worker nodes. When the developer submits a list of apps to the master, Kubernetes deploys them to the cluster of worker nodes. What node a component lands on doesn't (and shouldn't) matter—neither to the developer nor to the system administrator.



**Figure 1.8** Kubernetes exposes the whole datacenter as a single deployment platform.

The developer can specify that certain apps must run together and Kubernetes will deploy them on the same worker node. Others will be spread around the cluster, but they can talk to each other in the same way, regardless of where they're deployed.

#### HELPING DEVELOPERS FOCUS ON THE CORE APP FEATURES

Kubernetes can be thought of as an operating system for the cluster. It relieves application developers from having to implement certain infrastructure-related services into their apps; instead they rely on Kubernetes to provide these services. This includes things such as service discovery, scaling, load-balancing, self-healing, and even leader

election. Application developers can therefore focus on implementing the actual features of the applications and not waste time figuring out how to integrate them with the infrastructure.

#### HELPING OPS TEAMS ACHIEVE BETTER RESOURCE UTILIZATION

Kubernetes will run your containerized app somewhere in the cluster, provide information to its components on how to find each other, and keep all of them running. Because your application doesn't care which node it's running on, Kubernetes can relocate the app at any time, and by mixing and matching apps, achieve far better resource utilization than is possible with manual scheduling.

### 1.3.3 Understanding the architecture of a Kubernetes cluster

We've seen a bird's-eye view of Kubernetes' architecture. Now let's take a closer look at what a Kubernetes cluster is composed of. At the hardware level, a Kubernetes cluster is composed of many nodes, which can be split into two types:

- The *master* node, which hosts the *Kubernetes Control Plane* that controls and manages the whole Kubernetes system
- *Worker nodes* that run the actual applications you deploy

Figure 1.9 shows the components running on these two sets of nodes. I'll explain them next.

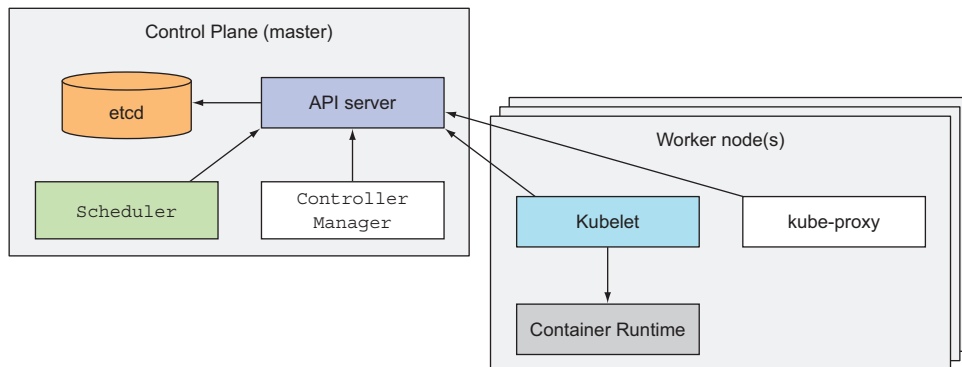


Figure 1.9 The components that make up a Kubernetes cluster

#### THE CONTROL PLANE

The Control Plane is what controls the cluster and makes it function. It consists of multiple components that can run on a single master node or be split across multiple nodes and replicated to ensure high availability. These components are

- The *Kubernetes API Server*, which you and the other Control Plane components communicate with

- The *Scheduler*, which schedules your apps (assigns a worker node to each deployable component of your application)
- The *Controller Manager*, which performs cluster-level functions, such as replicating components, keeping track of worker nodes, handling node failures, and so on
- *etcd*, a reliable distributed data store that persistently stores the cluster configuration.

The components of the Control Plane hold and control the state of the cluster, but they don't run your applications. This is done by the (worker) nodes.

#### THE NODES

The worker nodes are the machines that run your containerized applications. The task of running, monitoring, and providing services to your applications is done by the following components:

- Docker, rkt, or another *container runtime*, which runs your containers
- The *Kubelet*, which talks to the API server and manages containers on its node
- The *Kubernetes Service Proxy* (*kube-proxy*), which load-balances network traffic between application components

We'll explain all these components in detail in chapter 11. I'm not a fan of explaining how things work before first explaining *what* something does and teaching people to use it. It's like learning to drive a car. You don't want to know what's under the hood. You first want to learn how to drive it from point A to point B. Only after you learn how to do that do you become interested in how a car makes that possible. After all, knowing what's under the hood may someday help you get the car moving again after it breaks down and leaves you stranded at the side of the road.

### 1.3.4 Running an application in Kubernetes

To run an application in Kubernetes, you first need to package it up into one or more container images, push those images to an image registry, and then post a description of your app to the Kubernetes API server.

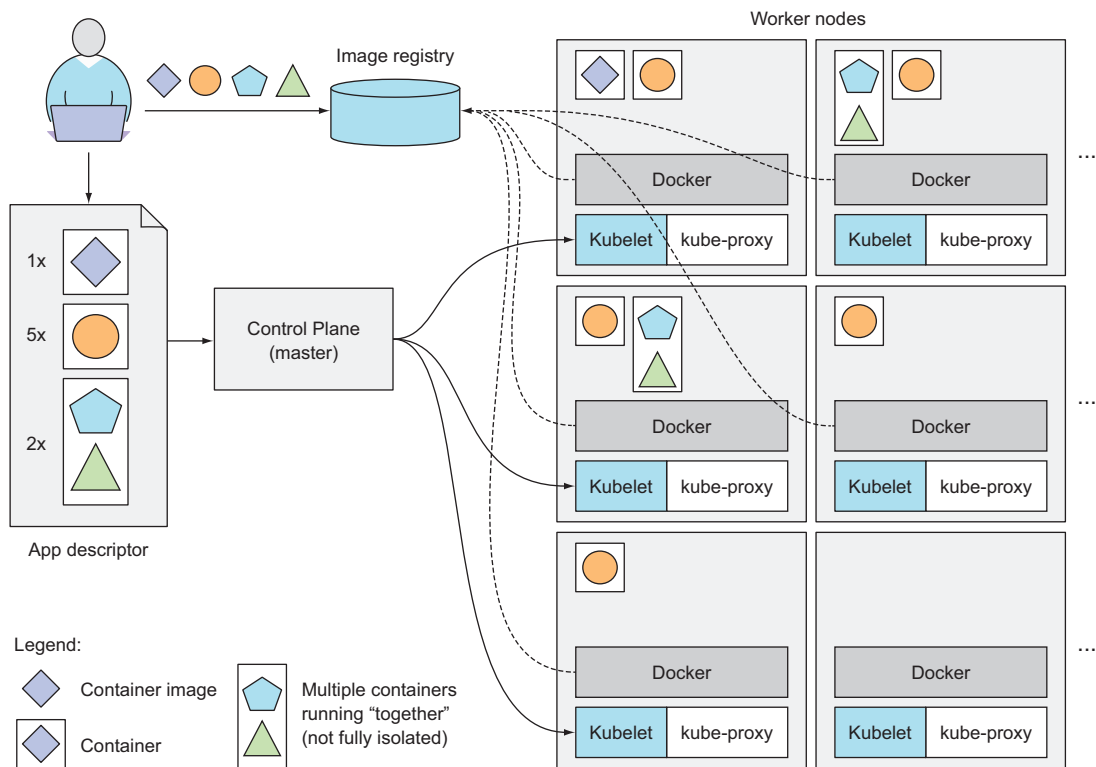
The description includes information such as the container image or images that contain your application components, how those components are related to each other, and which ones need to be run co-located (together on the same node) and which don't. For each component, you can also specify how many copies (or *replicas*) you want to run. Additionally, the description also includes which of those components provide a service to either internal or external clients and should be exposed through a single IP address and made discoverable to the other components.

#### UNDERSTANDING HOW THE DESCRIPTION RESULTS IN A RUNNING CONTAINER

When the API server processes your app's description, the Scheduler schedules the specified groups of containers onto the available worker nodes based on computational resources required by each group and the unallocated resources on each node

at that moment. The Kubelet on those nodes then instructs the Container Runtime (Docker, for example) to pull the required container images and run the containers.

Examine figure 1.10 to gain a better understanding of how applications are deployed in Kubernetes. The app descriptor lists four containers, grouped into three sets (these sets are called *pods*; we'll explain what they are in chapter 3). The first two pods each contain only a single container, whereas the last one contains two. That means both containers need to run co-located and shouldn't be isolated from each other. Next to each pod, you also see a number representing the number of replicas of each pod that need to run in parallel. After submitting the descriptor to Kubernetes, it will schedule the specified number of replicas of each pod to the available worker nodes. The Kubelets on the nodes will then tell Docker to pull the container images from the image registry and run the containers.



**Figure 1.10** A basic overview of the Kubernetes architecture and an application running on top of it

### KEEPING THE CONTAINERS RUNNING

Once the application is running, Kubernetes continuously makes sure that the deployed state of the application always matches the description you provided. For example, if

you specify that you always want five instances of a web server running, Kubernetes will always keep exactly five instances running. If one of those instances stops working properly, like when its process crashes or when it stops responding, Kubernetes will restart it automatically.

Similarly, if a whole worker node dies or becomes inaccessible, Kubernetes will select new nodes for all the containers that were running on the node and run them on the newly selected nodes.

#### **SCALING THE NUMBER OF COPIES**

While the application is running, you can decide you want to increase or decrease the number of copies, and Kubernetes will spin up additional ones or stop the excess ones, respectively. You can even leave the job of deciding the optimal number of copies to Kubernetes. It can automatically keep adjusting the number, based on real-time metrics, such as CPU load, memory consumption, queries per second, or any other metric your app exposes.

#### **HITTING A MOVING TARGET**

We've said that Kubernetes may need to move your containers around the cluster. This can occur when the node they were running on has failed or because they were evicted from a node to make room for other containers. If the container is providing a service to external clients or other containers running in the cluster, how can they use the container properly if it's constantly moving around the cluster? And how can clients connect to containers providing a service when those containers are replicated and spread across the whole cluster?

To allow clients to easily find containers that provide a specific service, you can tell Kubernetes which containers provide the same service and Kubernetes will expose all of them at a single static IP address and expose that address to all applications running in the cluster. This is done through environment variables, but clients can also look up the service IP through good old DNS. The kube-proxy will make sure connections to the service are load balanced across all the containers that provide the service. The IP address of the service stays constant, so clients can always connect to its containers, even when they're moved around the cluster.

### **1.3.5 Understanding the benefits of using Kubernetes**

If you have Kubernetes deployed on all your servers, the ops team doesn't need to deal with deploying your apps anymore. Because a containerized application already contains all it needs to run, the system administrators don't need to install anything to deploy and run the app. On any node where Kubernetes is deployed, Kubernetes can run the app immediately without any help from the sysadmins.

#### **SIMPLIFYING APPLICATION DEPLOYMENT**

Because Kubernetes exposes all its worker nodes as a single deployment platform, application developers can start deploying applications on their own and don't need to know anything about the servers that make up the cluster.

In essence, all the nodes are now a single bunch of computational resources that are waiting for applications to consume them. A developer doesn't usually care what kind of server the application is running on, as long as the server can provide the application with adequate system resources.

Certain cases do exist where the developer does care what kind of hardware the application should run on. If the nodes are heterogeneous, you'll find cases when you want certain apps to run on nodes with certain capabilities and run other apps on others. For example, one of your apps may require being run on a system with SSDs instead of HDDs, while other apps run fine on HDDs. In such cases, you obviously want to ensure that particular app is always scheduled to a node with an SSD.

Without using Kubernetes, the sysadmin would select one specific node that has an SSD and deploy the app there. But when using Kubernetes, instead of selecting a specific node where your app should be run, it's more appropriate to tell Kubernetes to only choose among nodes with an SSD. You'll learn how to do that in chapter 3.

#### **ACHIEVING BETTER UTILIZATION OF HARDWARE**

By setting up Kubernetes on your servers and using it to run your apps instead of running them manually, you've decoupled your app from the infrastructure. When you tell Kubernetes to run your application, you're letting it choose the most appropriate node to run your application on based on the description of the application's resource requirements and the available resources on each node.

By using containers and not tying the app down to a specific node in your cluster, you're allowing the app to freely move around the cluster at any time, so the different app components running on the cluster can be mixed and matched to be packed tightly onto the cluster nodes. This ensures the node's hardware resources are utilized as best as possible.

The ability to move applications around the cluster at any time allows Kubernetes to utilize the infrastructure much better than what you can achieve manually. Humans aren't good at finding optimal combinations, especially when the number of all possible options is huge, such as when you have many application components and many server nodes they can be deployed on. Computers can obviously perform this work much better and faster than humans.

#### **HEALTH CHECKING AND SELF-HEALING**

Having a system that allows moving an application across the cluster at any time is also valuable in the event of server failures. As your cluster size increases, you'll deal with failing computer components ever more frequently.

Kubernetes monitors your app components and the nodes they run on and automatically reschedules them to other nodes in the event of a node failure. This frees the ops team from having to migrate app components manually and allows the team to immediately focus on fixing the node itself and returning it to the pool of available hardware resources instead of focusing on relocating the app.

If your infrastructure has enough spare resources to allow normal system operation even without the failed node, the ops team doesn't even need to react to the failure

immediately, such as at 3 a.m. They can sleep tight and deal with the failed node during regular work hours.

#### **AUTOMATIC SCALING**

Using Kubernetes to manage your deployed applications also means the ops team doesn't need to constantly monitor the load of individual applications to react to sudden load spikes. As previously mentioned, Kubernetes can be told to monitor the resources used by each application and to keep adjusting the number of running instances of each application.

If Kubernetes is running on cloud infrastructure, where adding additional nodes is as easy as requesting them through the cloud provider's API, Kubernetes can even automatically scale the whole cluster size up or down based on the needs of the deployed applications.

#### **SIMPLIFYING APPLICATION DEVELOPMENT**

The features described in the previous section mostly benefit the operations team. But what about the developers? Does Kubernetes bring anything to their table? It definitely does.

If you turn back to the fact that apps run in the same environment both during development and in production, this has a big effect on when bugs are discovered. We all agree the sooner you discover a bug, the easier it is to fix it, and fixing it requires less work. It's the developers who do the fixing, so this means less work for them.

Then there's the fact that developers don't need to implement features that they would usually implement. This includes discovery of services and/or peers in a clustered application. Kubernetes does this instead of the app. Usually, the app only needs to look up certain environment variables or perform a DNS lookup. If that's not enough, the application can query the Kubernetes API server directly to get that and/or other information. Querying the Kubernetes API server like that can even save developers from having to implement complicated mechanisms such as leader election.

As a final example of what Kubernetes brings to the table, you also need to consider the increase in confidence developers will feel knowing that when a new version of their app is going to be rolled out, Kubernetes can automatically detect if the new version is bad and stop its rollout immediately. This increase in confidence usually accelerates the continuous delivery of apps, which benefits the whole organization.

## **1.4 Summary**

In this introductory chapter, you've seen how applications have changed in recent years and how they can now be harder to deploy and manage. We've introduced Kubernetes and shown how it, together with Docker and other container platforms, helps deploy and manage applications and the infrastructure they run on. You've learned that

- Monolithic apps are easier to deploy, but harder to maintain over time and sometimes impossible to scale.

- Microservices-based application architectures allow easier development of each component, but are harder to deploy and configure to work as a single system.
- Linux containers provide much the same benefits as virtual machines, but are far more lightweight and allow for much better hardware utilization.
- Docker improved on existing Linux container technologies by allowing easier and faster provisioning of containerized apps together with their OS environments.
- Kubernetes exposes the whole datacenter as a single computational resource for running applications.
- Developers can deploy apps through Kubernetes without assistance from sysadmins.
- Sysadmins can sleep better by having Kubernetes deal with failed nodes automatically.

In the next chapter, you'll get your hands dirty by building an app and running it in Docker and then in Kubernetes.



# *First steps with Docker and Kubernetes*

---

## **This chapter covers**

- Creating, running, and sharing a container image with Docker
- Running a single-node Kubernetes cluster locally
- Setting up a Kubernetes cluster on Google Kubernetes Engine
- Setting up and using the `kubectl` command-line client
- Deploying an app on Kubernetes and scaling it horizontally

Before you start learning about Kubernetes concepts in detail, let's see how to create a simple application, package it into a container image, and run it in a managed Kubernetes cluster (in Google Kubernetes Engine) or in a local single-node cluster. This should give you a slightly better overview of the whole Kubernetes system and will make it easier to follow the next few chapters, where we'll go over the basic building blocks and concepts in Kubernetes.

## 2.1 *Creating, running, and sharing a container image*

As you’ve already learned in the previous chapter, running applications in Kubernetes requires them to be packaged into container images. We’ll do a basic introduction to using Docker in case you haven’t used it yet. In the next few sections you’ll

- 1 Install Docker and run your first “Hello world” container
- 2 Create a trivial Node.js app that you’ll later deploy in Kubernetes
- 3 Package the app into a container image so you can then run it as an isolated container
- 4 Run a container based on the image
- 5 Push the image to Docker Hub so that anyone anywhere can run it

### 2.1.1 *Installing Docker and running a Hello World container*

First, you’ll need to install Docker on your Linux machine. If you don’t use Linux, you’ll need to start a Linux virtual machine (VM) and run Docker inside that VM. If you’re using a Mac or Windows and install Docker per instructions, Docker will set up a VM for you and run the Docker daemon inside that VM. The Docker client executable will be available on your host OS, and will communicate with the daemon inside the VM.

To install Docker, follow the instructions at <http://docs.docker.com/engine/installation/> for your specific operating system. After completing the installation, you can use the Docker client executable to run various Docker commands. For example, you could try pulling and running an existing image from Docker Hub, the public Docker registry, which contains ready-to-use container images for many well-known software packages. One of them is the `busybox` image, which you’ll use to run a simple `echo "Hello world"` command.

#### **RUNNING A HELLO WORLD CONTAINER**

If you’re unfamiliar with `busybox`, it’s a single executable that combines many of the standard UNIX command-line tools, such as `echo`, `ls`, `gzip`, and so on. Instead of the `busybox` image, you could also use any other full-fledged OS container image such as Fedora, Ubuntu, or other similar images, as long as it includes the `echo` executable.

How do you run the `busybox` image? You don’t need to download or install anything. Use the `docker run` command and specify what image to download and run and (optionally) what command to execute, as shown in the following listing.

#### **Listing 2.1 Running a Hello world container with Docker**

```
$ docker run busybox echo "Hello world"
Unable to find image 'busybox:latest' locally
latest: Pulling from docker.io/busybox
9a163e0b8d13: Pull complete
fef924a0204a: Pull complete
Digest: sha256:97473e34e311e6c1b3f61f2a721d038d1e5eef17d98d1353a513007cf46ca6bd
Status: Downloaded newer image for docker.io/busybox:latest
Hello world
```

This doesn't look that impressive, but when you consider that the whole “app” was downloaded and executed with a single command, without you having to install that app or anything else, you'll agree it's awesome. In your case, the app was a single executable (busybox), but it might as well have been an incredibly complex app with many dependencies. The whole process of setting up and running the app would have been exactly the same. What's also important is that the app was executed inside a container, completely isolated from all the other processes running on your machine.

### UNDERSTANDING WHAT HAPPENS BEHIND THE SCENES

Figure 2.1 shows exactly what happened when you performed the `docker run` command. First, Docker checked to see if the `busybox:latest` image was already present on your local machine. It wasn't, so Docker pulled it from the Docker Hub registry at <http://docker.io>. After the image was downloaded to your machine, Docker created a container from that image and ran the command inside it. The `echo` command printed the text to STDOUT and then the process terminated and the container stopped.

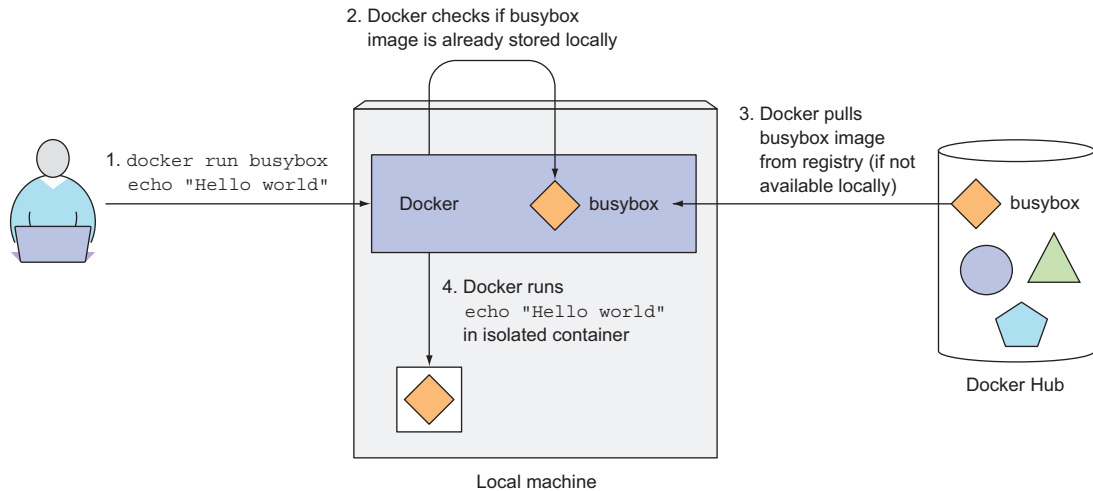


Figure 2.1 Running `echo "Hello world"` in a container based on the `busybox` container image

### RUNNING OTHER IMAGES

Running other existing container images is much the same as how you ran the `busybox` image. In fact, it's often even simpler, because you usually don't need to specify what command to execute, the way you did in the example (`echo "Hello world"`). The command that should be executed is usually baked into the image itself, but you can override it if you want. After searching or browsing through the publicly available images on <http://hub.docker.com> or another public registry, you tell Docker to run the image like this:

```
$ docker run <image>
```

### VERSIONING CONTAINER IMAGES

All software packages get updated, so more than a single version of a package usually exists. Docker supports having multiple versions or variants of the same image under the same name. Each variant must have a unique tag. When referring to images without explicitly specifying the tag, Docker will assume you're referring to the so-called *latest* tag. To run a different version of the image, you may specify the tag along with the image name like this:

```
$ docker run <image>:<tag>
```

#### 2.1.2 *Creating a trivial Node.js app*

Now that you have a working Docker setup, you're going to create an app. You'll build a trivial Node.js web application and package it into a container image. The application will accept HTTP requests and respond with the hostname of the machine it's running in. This way, you'll see that an app running inside a container sees its own hostname and not that of the host machine, even though it's running on the host like any other process. This will be useful later, when you deploy the app on Kubernetes and scale it out (scale it horizontally; that is, run multiple instances of the app). You'll see your HTTP requests hitting different instances of the app.

Your app will consist of a single file called `app.js` with the contents shown in the following listing.

##### Listing 2.2 A simple Node.js app: `app.js`

```
const http = require('http');
const os = require('os');

console.log("Kubia server starting...");

var handler = function(request, response) {
  console.log("Received request from " + request.connection.remoteAddress);
  response.writeHead(200);
  response.end("You've hit " + os.hostname() + "\n");
};

var www = http.createServer(handler);
www.listen(8080);
```

It should be clear what this code does. It starts up an HTTP server on port 8080. The server responds with an HTTP response status code 200 OK and the text "You've hit <hostname>" to every request. The request handler also logs the client's IP address to the standard output, which you'll need later.

**NOTE** The returned hostname is the server's actual hostname, not the one the client sends in the HTTP request's `Host` header.

You could now download and install Node.js and test your app directly, but this isn't necessary, because you'll use Docker to package the app into a container image and

enable it to be run anywhere without having to download or install anything (except Docker, which does need to be installed on the machine you want to run the image on).

### 2.1.3 Creating a Dockerfile for the image

To package your app into an image, you first need to create a file called `Dockerfile`, which will contain a list of instructions that Docker will perform when building the image. The `Dockerfile` needs to be in the same directory as the `app.js` file and should contain the commands shown in the following listing.

#### Listing 2.3 A Dockerfile for building a container image for your app

```
FROM node:7
ADD app.js /app.js
ENTRYPOINT ["node", "app.js"]
```

The `FROM` line defines the container image you'll use as a starting point (the base image you're building on top of). In your case, you're using the `node` container image, tag 7. In the second line, you're adding your `app.js` file from your local directory into the root directory in the image, under the same name (`app.js`). Finally, in the third line, you're defining what command should be executed when somebody runs the image. In your case, the command is `node app.js`.

#### Choosing a base image

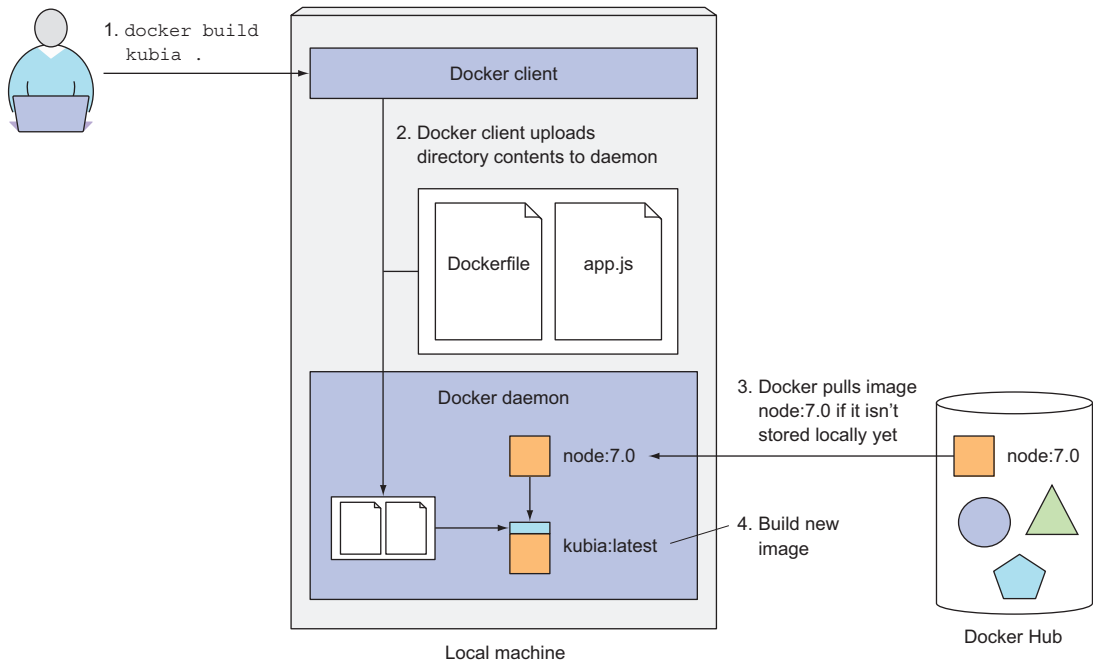
You may wonder why we chose this specific image as your base. Because your app is a Node.js app, you need your image to contain the `node` binary executable to run the app. You could have used any image that contains that binary, or you could have even used a Linux distro base image such as `fedora` or `ubuntu` and installed Node.js into the container at image build time. But because the `node` image is made specifically for running Node.js apps, and includes everything you need to run your app, you'll use that as the base image.

### 2.1.4 Building the container image

Now that you have your `Dockerfile` and the `app.js` file, you have everything you need to build your image. To build it, run the following Docker command:

```
$ docker build -t kubia .
```

Figure 2.2 shows what happens during the build process. You're telling Docker to build an image called `kubia` based on the contents of the current directory (note the dot at the end of the build command). Docker will look for the `Dockerfile` in the directory and build the image based on the instructions in the file.



**Figure 2.2** Building a new container image from a Dockerfile

### UNDERSTANDING HOW AN IMAGE IS BUILT

The build process isn't performed by the Docker client. Instead, the contents of the whole directory are uploaded to the Docker daemon and the image is built there. The client and daemon don't need to be on the same machine at all. If you're using Docker on a non-Linux OS, the client is on your host OS, but the daemon runs inside a VM. Because all the files in the build directory are uploaded to the daemon, if it contains many large files and the daemon isn't running locally, the upload may take longer.

**TIP** Don't include any unnecessary files in the build directory, because they'll slow down the build process—especially when the Docker daemon is on a remote machine.

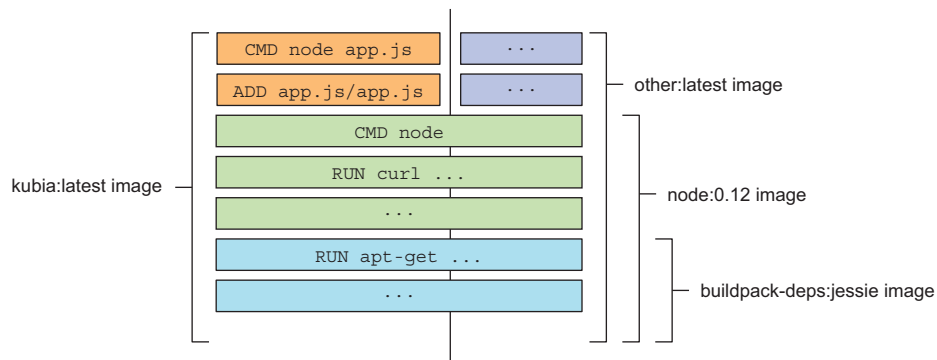
During the build process, Docker will first pull the base image (`node:7`) from the public image repository (Docker Hub), unless the image has already been pulled and is stored on your machine.

### UNDERSTANDING IMAGE LAYERS

An image isn't a single, big, binary blob, but is composed of multiple layers, which you may have already noticed when running the `busybox` example (there were multiple `Pull complete` lines—one for each layer). Different images may share several layers,

which makes storing and transferring images much more efficient. For example, if you create multiple images based on the same base image (such as `node:7` in the example), all the layers comprising the base image will be stored only once. Also, when pulling an image, Docker will download each layer individually. Several layers may already be stored on your machine, so Docker will only download those that aren't.

You may think that each Dockerfile creates only a single new layer, but that's not the case. When building an image, a new layer is created for each individual command in the Dockerfile. During the build of your image, after pulling all the layers of the base image, Docker will create a new layer on top of them and add the `app.js` file into it. Then it will create yet another layer that will specify the command that should be executed when the image is run. This last layer will then be tagged as `kubia:latest`. This is shown in figure 2.3, which also shows how a different image called `other:latest` would use the same layers of the Node.js image as your own image does.



**Figure 2.3** Container images are composed of layers that can be shared among different images.

When the build process completes, you have a new image stored locally. You can see it by telling Docker to list all locally stored images, as shown in the following listing.

#### Listing 2.4 Listing locally stored images

```
$ docker images
REPOSITORY    TAG       IMAGE ID       CREATED        VIRTUAL SIZE
kubia         latest   d30ecc7419e7   1 minute ago   637.1 MB
...
```

#### COMPARING BUILDING IMAGES WITH A DOCKERFILE VS. MANUALLY

Dockerfiles are the usual way of building container images with Docker, but you could also build the image manually by running a container from an existing image, executing commands in the container, exiting the container, and committing the final state as a new image. This is exactly what happens when you build from a Dockerfile, but it's performed automatically and is repeatable, which allows you to make changes to

the Dockerfile and rebuild the image any time, without having to manually retype all the commands again.

### 2.1.5 *Running the container image*

You can now run your image with the following command:

```
$ docker run --name kubia-container -p 8080:8080 -d kubia
```

This tells Docker to run a new container called `kubia-container` from the `kubia` image. The container will be detached from the console (`-d` flag), which means it will run in the background. Port 8080 on the local machine will be mapped to port 8080 inside the container (`-p 8080:8080` option), so you can access the app through <http://localhost:8080>.

If you're not running the Docker daemon on your local machine (if you're using a Mac or Windows, the daemon is running inside a VM), you'll need to use the host-name or IP of the VM running the daemon instead of `localhost`. You can look it up through the `DOCKER_HOST` environment variable.

#### ACCESSING YOUR APP

Now try to access your application at <http://localhost:8080> (be sure to replace `localhost` with the hostname or IP of the Docker host if necessary):

```
$ curl localhost:8080
You've hit 44d76963e8e1
```

That's the response from your app. Your tiny application is now running inside a container, isolated from everything else. As you can see, it's returning `44d76963e8e1` as its hostname, and not the actual hostname of your host machine. The hexadecimal number is the ID of the Docker container.

#### LISTING ALL RUNNING CONTAINERS

Let's list all running containers in the following listing, so you can examine the list (I've edited the output to make it more readable—imagine the last two lines as the continuation of the first two).

##### Listing 2.5 Listing running containers

```
$ docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED        ...
44d76963e8e1   kubia:latest   "/bin/sh -c 'node ap    6 minutes ago  ...

... STATUS                PORTS                NAMES
... Up 6 minutes          0.0.0.0:8080->8080/tcp kubia-container
```

A single container is running. For each container, Docker prints out its ID and name, the image used to run the container, and the command that's executing inside the container.



**GETTING ADDITIONAL INFORMATION ABOUT A CONTAINER**

The `docker ps` command only shows the most basic information about the containers. To see additional information, you can use `docker inspect`:

```
$ docker inspect kubia-container
```

Docker will print out a long JSON containing low-level information about the container.

**2.1.6 Exploring the inside of a running container**

What if you want to see what the environment is like inside the container? Because multiple processes can run inside the same container, you can always run an additional process in it to see what's inside. You can even run a shell, provided that the shell's binary executable is available in the image.

**RUNNING A SHELL INSIDE AN EXISTING CONTAINER**

The Node.js image on which you've based your image contains the bash shell, so you can run the shell inside the container like this:

```
$ docker exec -it kubia-container bash
```

This will run bash inside the existing `kubia-container` container. The bash process will have the same Linux namespaces as the main container process. This allows you to explore the container from within and see how Node.js and your app see the system when running inside the container. The `-it` option is shorthand for two options:

- `-i`, which makes sure STDIN is kept open. You need this for entering commands into the shell.
- `-t`, which allocates a pseudo terminal (TTY).

You need both if you want to use the shell like you're used to. (If you leave out the first one, you can't type any commands, and if you leave out the second one, the command prompt won't be displayed and some commands will complain about the `TERM` variable not being set.)

**EXPLORING THE CONTAINER FROM WITHIN**

Let's see how to use the shell in the following listing to see the processes running in the container.

**Listing 2.6 Listing processes from inside a container**

```
root@44d76963e8e1:/# ps aux
USER  PID  %CPU  %MEM    VSZ   RSS TTY  STAT  START  TIME  COMMAND
root    1   0.0   0.1 676380 16504 ?    S1   12:31   0:00 node app.js
root   10   0.0   0.0  20216  1924 ?    Ss   12:31   0:00 bash
root   19   0.0   0.0  17492  1136 ?    R+   12:38   0:00 ps aux
```

You see only three processes. You don't see any other processes from the host OS.

**UNDERSTANDING THAT PROCESSES IN A CONTAINER RUN IN THE HOST OPERATING SYSTEM**

If you now open another terminal and list the processes on the host OS itself, you will, among all other host processes, also see the processes running in the container, as shown in listing 2.7.

**NOTE** If you're using a Mac or Windows, you'll need to log into the VM where the Docker daemon is running to see these processes.

**Listing 2.7 A container's processes run in the host OS**

```
$ ps aux | grep app.js
USER  PID %CPU %MEM    VSZ   RSS TTY  STAT  START  TIME  COMMAND
root  382  0.0  0.1 676380 16504 ?    Sl   12:31  0:00  node app.js
```

This proves that processes running in the container are running in the host OS. If you have a keen eye, you may have noticed that the processes have different IDs inside the container vs. on the host. The container is using its own PID Linux namespace and has a completely isolated process tree, with its own sequence of numbers.

**THE CONTAINER'S FILESYSTEM IS ALSO ISOLATED**

Like having an isolated process tree, each container also has an isolated filesystem. Listing the contents of the root directory inside the container will only show the files in the container and will include all the files that are in the image plus any files that are created while the container is running (log files and similar), as shown in the following listing.

**Listing 2.8 A container has its own complete filesystem**

```
root@44d76963e8e1:/# ls /
app.js  boot  etc   lib   media  opt   root  sbin  sys  usr
bin     dev   home  lib64 mnt    proc  run   srv   tmp  var
```

It contains the `app.js` file and other system directories that are part of the `node:7` base image you're using. To exit the container, you exit the shell by running the `exit` command and you'll be returned to your host machine (like logging out of an `ssh` session, for example).

**TIP** Entering a running container like this is useful when debugging an app running in a container. When something's wrong, the first thing you'll want to explore is the actual state of the system your application sees. Keep in mind that an application will not only see its own unique filesystem, but also processes, users, hostname, and network interfaces.

**2.1.7 Stopping and removing a container**

To stop your app, you tell Docker to stop the `kubia-container` container:

```
$ docker stop kubia-container
```

This will stop the main process running in the container and consequently stop the container, because no other processes are running inside the container. The container itself still exists and you can see it with `docker ps -a`. The `-a` option prints out all the containers, those running and those that have been stopped. To truly remove a container, you need to remove it with the `docker rm` command:

```
$ docker rm kubia-container
```

This deletes the container. All its contents are removed and it can't be started again.

### 2.1.8 Pushing the image to an image registry

The image you've built has so far only been available on your local machine. To allow you to run it on any other machine, you need to push the image to an external image registry. For the sake of simplicity, you won't set up a private image registry and will instead push the image to Docker Hub (<http://hub.docker.com>), which is one of the publicly available registries. Other widely used such registries are Quay.io and the Google Container Registry.

Before you do that, you need to re-tag your image according to Docker Hub's rules. Docker Hub will allow you to push an image if the image's repository name starts with your Docker Hub ID. You create your Docker Hub ID by registering at <http://hub.docker.com>. I'll use my own ID (luksa) in the following examples. Please change every occurrence with your own ID.

#### TAGGING AN IMAGE UNDER AN ADDITIONAL TAG

Once you know your ID, you're ready to rename your image, currently tagged as `kubia`, to `luksa/kubia` (replace `luksa` with your own Docker Hub ID):

```
$ docker tag kubia luksa/kubia
```

This doesn't rename the tag; it creates an additional tag for the same image. You can confirm this by listing the images stored on your system with the `docker images` command, as shown in the following listing.

**Listing 2.9** A container image can have multiple tags

```
$ docker images | head
REPOSITORY          TAG          IMAGE ID          CREATED           VIRTUAL SIZE
luksa/kubia         latest      d30ecc7419e7     About an hour ago 654.5 MB
kubia               latest      d30ecc7419e7     About an hour ago 654.5 MB
docker.io/node      7.0        04c0ca2a8dad     2 days ago       654.5 MB
...
```

As you can see, both `kubia` and `luksa/kubia` point to the same image ID, so they're in fact one single image with two tags.

### PUSHING THE IMAGE TO DOCKER HUB

Before you can push the image to Docker Hub, you need to log in under your user ID with the `docker login` command. Once you're logged in, you can finally push the `yourid/kubia` image to Docker Hub like this:

```
$ docker push luksa/kubia
```

### RUNNING THE IMAGE ON A DIFFERENT MACHINE

After the push to Docker Hub is complete, the image will be available to everyone. You can now run the image on any machine running Docker by executing the following command:

```
$ docker run -p 8080:8080 -d luksa/kubia
```

It doesn't get much simpler than that. And the best thing about this is that your application will have the exact same environment every time and everywhere it's run. If it ran fine on your machine, it should run as well on every other Linux machine. No need to worry about whether the host machine has Node.js installed or not. In fact, even if it does, your app won't use it, because it will use the one installed inside the image.

## 2.2 *Setting up a Kubernetes cluster*

Now that you have your app packaged inside a container image and made available through Docker Hub, you can deploy it in a Kubernetes cluster instead of running it in Docker directly. But first, you need to set up the cluster itself.

Setting up a full-fledged, multi-node Kubernetes cluster isn't a simple task, especially if you're not well-versed in Linux and networking administration. A proper Kubernetes install spans multiple physical or virtual machines and requires the networking to be set up properly, so that all the containers running inside the Kubernetes cluster can connect to each other through the same flat networking space.

A long list of methods exists for installing a Kubernetes cluster. These methods are described in detail in the documentation at <http://kubernetes.io>. We're not going to list all of them here, because the list keeps evolving, but Kubernetes can be run on your local development machine, your own organization's cluster of machines, on cloud providers providing virtual machines (Google Compute Engine, Amazon EC2, Microsoft Azure, and so on), or by using a managed Kubernetes cluster such as Google Kubernetes Engine (previously known as Google Container Engine).

In this chapter, we'll cover two simple options for getting your hands on a running Kubernetes cluster. You'll see how to run a single-node Kubernetes cluster on your local machine and how to get access to a hosted cluster running on Google Kubernetes Engine (GKE).

A third option, which covers installing a cluster with the `kubeadm` tool, is explained in appendix B. The instructions there show you how to set up a three-node Kubernetes

cluster using virtual machines, but I suggest you try it only after reading the first 11 chapters of the book.

Another option is to install Kubernetes on Amazon’s AWS (Amazon Web Services). For this, you can look at the `kops` tool, which is built on top of `kubeadm` mentioned in the previous paragraph, and is available at <http://github.com/kubernetes/kops>. It helps you deploy production-grade, highly available Kubernetes clusters on AWS and will eventually support other platforms as well (Google Kubernetes Engine, VMware, vSphere, and so on).

### 2.2.1 Running a local single-node Kubernetes cluster with Minikube

The simplest and quickest path to a fully functioning Kubernetes cluster is by using Minikube. Minikube is a tool that sets up a single-node cluster that’s great for both testing Kubernetes and developing apps locally.

Although we can’t show certain Kubernetes features related to managing apps on multiple nodes, the single-node cluster should be enough for exploring most topics discussed in this book.

#### INSTALLING MINIKUBE

Minikube is a single binary that needs to be downloaded and put onto your path. It’s available for OSX, Linux, and Windows. To install it, the best place to start is to go to the Minikube repository on GitHub (<http://github.com/kubernetes/minikube>) and follow the instructions there.

For example, on OSX and Linux, Minikube can be downloaded and set up with a single command. For OSX, this is what the command looks like:

```
$ curl -Lo minikube https://storage.googleapis.com/minikube/releases/  
➡ v0.23.0/minikube-darwin-amd64 && chmod +x minikube && sudo mv minikube  
➡ /usr/local/bin/
```

On Linux, you download a different release (replace “darwin” with “linux” in the URL). On Windows, you can download the file manually, rename it to `minikube.exe`, and put it onto your path. Minikube runs Kubernetes inside a VM run through either VirtualBox or KVM, so you also need to install one of them before you can start the Minikube cluster.

#### STARTING A KUBERNETES CLUSTER WITH MINIKUBE

Once you have Minikube installed locally, you can immediately start up the Kubernetes cluster with the command in the following listing.

##### Listing 2.10 Starting a Minikube virtual machine

```
$ minikube start  
Starting local Kubernetes cluster...  
Starting VM...  
SSH-ing files into VM...  
...  
Kubectl is now configured to use the cluster.
```

Starting the cluster takes more than a minute, so don't interrupt the command before it completes.

### INSTALLING THE KUBERNETES CLIENT (KUBECTL)

To interact with Kubernetes, you also need the `kubectl` CLI client. Again, all you need to do is download it and put it on your path. The latest stable release for OSX, for example, can be downloaded and installed with the following command:

```
$ curl -LO https://storage.googleapis.com/kubernetes-release/release
➤ /$(curl -s https://storage.googleapis.com/kubernetes-release/release
➤ /stable.txt)/bin/darwin/amd64/kubectl
➤ && chmod +x kubectl
➤ && sudo mv kubectl /usr/local/bin/
```

To download `kubectl` for Linux or Windows, replace `darwin` in the URL with either `linux` or `windows`.

**NOTE** If you'll be using multiple Kubernetes clusters (for example, both Minikube and GKE), refer to appendix A for information on how to set up and switch between different `kubectl` contexts.

### CHECKING TO SEE THE CLUSTER IS UP AND KUBECTL CAN TALK TO IT

To verify your cluster is working, you can use the `kubectl cluster-info` command shown in the following listing.

#### Listing 2.11 Displaying cluster information

```
$ kubectl cluster-info
Kubernetes master is running at https://192.168.99.100:8443
KubeDNS is running at https://192.168.99.100:8443/api/v1/proxy/...
kubernetes-dashboard is running at https://192.168.99.100:8443/api/v1/...
```

This shows the cluster is up. It shows the URLs of the various Kubernetes components, including the API server and the web console.

**TIP** You can run `minikube ssh` to log into the Minikube VM and explore it from the inside. For example, you may want to see what processes are running on the node.

## 2.2.2 Using a hosted Kubernetes cluster with Google Kubernetes Engine

If you want to explore a full-fledged multi-node Kubernetes cluster instead, you can use a managed Google Kubernetes Engine (GKE) cluster. This way, you don't need to manually set up all the cluster nodes and networking, which is usually too much for someone making their first steps with Kubernetes. Using a managed solution such as GKE makes sure you don't end up with a misconfigured, non-working, or partially working cluster.

**SETTING UP A GOOGLE CLOUD PROJECT AND DOWNLOADING THE NECESSARY CLIENT BINARIES**

Before you can set up a new Kubernetes cluster, you need to set up your GKE environment. Because the process may change, I'm not listing the exact instructions here. To get started, please follow the instructions at <https://cloud.google.com/container-engine/docs/before-you-begin>.

Roughly, the whole procedure includes

- 1 Signing up for a Google account, in the unlikely case you don't have one already.
- 2 Creating a project in the Google Cloud Platform Console.
- 3 Enabling billing. This does require your credit card info, but Google provides a 12-month free trial. And they're nice enough to not start charging automatically after the free trial is over.)
- 4 Enabling the Kubernetes Engine API.
- 5 Downloading and installing Google Cloud SDK. (This includes the *gcloud* command-line tool, which you'll need to create a Kubernetes cluster.)
- 6 Installing the *kubectl* command-line tool with *gcloud* components `install kubectl`.

**NOTE** Certain operations (the one in step 2, for example) may take a few minutes to complete, so relax and grab a coffee in the meantime.

**CREATING A KUBERNETES CLUSTER WITH THREE NODES**

After completing the installation, you can create a Kubernetes cluster with three worker nodes using the command shown in the following listing.

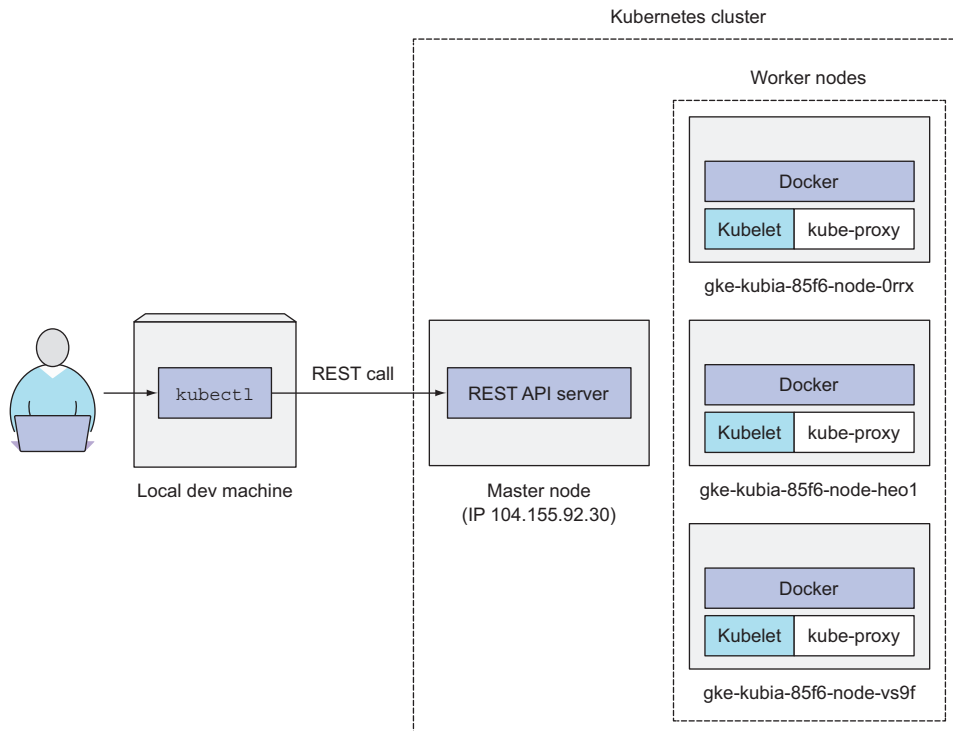
**Listing 2.12 Creating a three-node cluster in GKE**

```
$ gcloud container clusters create kubia --num-nodes 3
➡ --machine-type f1-micro
Creating cluster kubia...done.
Created [https://container.googleapis.com/v1/projects/kubia1-
1227/zones/europe-west1-d/clusters/kubia].
kubeconfig entry generated for kubia.
NAME     ZONE     MST_VER  MASTER_IP    TYPE      NODE_VER  NUM_NODES  STATUS
kubia    eu-wld  1.5.3    104.155.92.30 f1-micro  1.5.3     3          RUNNING
```

You should now have a running Kubernetes cluster with three worker nodes as shown in figure 2.4. You're using three nodes to help better demonstrate features that apply to multiple nodes. You can use a smaller number of nodes, if you want.

**GETTING AN OVERVIEW OF YOUR CLUSTER**

To give you a basic idea of what your cluster looks like and how to interact with it, see figure 2.4. Each node runs Docker, the Kubelet and the kube-proxy. You'll interact with the cluster through the *kubectl* command line client, which issues REST requests to the Kubernetes API server running on the master node.



**Figure 2.4** How you're interacting with your three-node Kubernetes cluster

### CHECKING IF THE CLUSTER IS UP BY LISTING CLUSTER NODES

You'll use the `kubectl` command now to list all the nodes in your cluster, as shown in the following listing.

#### Listing 2.13 Listing cluster nodes with `kubectl`

```
$ kubectl get nodes
NAME                                STATUS    AGE    VERSION
gke-kubia-85f6-node-0rrx           Ready    1m     v1.5.3
gke-kubia-85f6-node-heo1           Ready    1m     v1.5.3
gke-kubia-85f6-node-vs9f           Ready    1m     v1.5.3
```

The `kubectl get` command can list all kinds of Kubernetes objects. You'll use it constantly, but it usually shows only the most basic information for the listed objects.

**TIP** You can log into one of the nodes with `gcloud compute ssh <node-name>` to explore what's running on the node.



### RETRIEVING ADDITIONAL DETAILS OF AN OBJECT

To see more detailed information about an object, you can use the `kubectl describe` command, which shows much more:

```
$ kubectl describe node gke-kubia-85f6-node-0rrx
```

I'm omitting the actual output of the `describe` command, because it's fairly wide and would be completely unreadable here in the book. The output shows the node's status, its CPU and memory data, system information, containers running on the node, and much more.

In the previous `kubectl describe` example, you specified the name of the node explicitly, but you could also have performed a simple `kubectl describe node` without typing the node's name and it would print out a detailed description of all the nodes.

**TIP** Running the `describe` and `get` commands without specifying the name of the object comes in handy when only one object of a given type exists, so you don't waste time typing or copy/pasting the object's name.

While we're talking about reducing keystrokes, let me give you additional advice on how to make working with `kubectl` much easier, before we move on to running your first app in Kubernetes.

### 2.2.3 Setting up an alias and command-line completion for `kubectl`

You'll use `kubectl` often. You'll soon realize that having to type the full command every time is a real pain. Before you continue, take a minute to make your life easier by setting up an alias and tab completion for `kubectl`.

#### CREATING AN ALIAS

Throughout the book, I'll always be using the full name of the `kubectl` executable, but you may want to add a short alias such as `k`, so you won't have to type `kubectl` every time. If you haven't used aliases yet, here's how you define one. Add the following line to your `~/ .bashrc` or equivalent file:

```
alias k=kubectl
```

**NOTE** You may already have the `k` executable if you used `gcloud` to set up the cluster.

#### CONFIGURING TAB COMPLETION FOR KUBECTL

Even with a short alias such as `k`, you'll still need to type way more than you'd like. Luckily, the `kubectl` command can also output shell completion code for both the `bash` and `zsh` shell. It doesn't enable tab completion of only command names, but also of the actual object names. For example, instead of having to write the whole node name in the previous example, all you'd need to type is

```
$ kubectl desc<TAB> no<TAB> gke-ku<TAB>
```

To enable tab completion in bash, you'll first need to install a package called `bash-completion` and then run the following command (you'll probably also want to add it to `~/.bashrc` or equivalent):

```
$ source <(kubectl completion bash)
```

But there's one caveat. When you run the preceding command, tab completion will only work when you use the full `kubectl` name (it won't work when you use the `k` alias). To fix this, you need to transform the output of the `kubectl completion` command a bit:

```
$ source <(kubectl completion bash | sed s/kubectl/k/g)
```

**NOTE** Unfortunately, as I'm writing this, shell completion doesn't work for aliases on MacOS. You'll have to use the full `kubectl` command name if you want completion to work.

Now you're all set to start interacting with your cluster without having to type too much. You can finally run your first app on Kubernetes.

## 2.3 *Running your first app on Kubernetes*

Because this may be your first time, you'll use the simplest possible way of running an app on Kubernetes. Usually, you'd prepare a JSON or YAML manifest, containing a description of all the components you want to deploy, but because we haven't talked about the types of components you can create in Kubernetes yet, you'll use a simple one-line command to get something running.

### 2.3.1 *Deploying your Node.js app*

The simplest way to deploy your app is to use the `kubectl run` command, which will create all the necessary components without having to deal with JSON or YAML. This way, we don't need to dive into the structure of each object yet. Try to run the image you created and pushed to Docker Hub earlier. Here's how to run it in Kubernetes:

```
$ kubectl run kuba --image=luksa/kuba --port=8080 --generator=run/v1
replicationcontroller "kuba" created
```

The `--image=luksa/kuba` part obviously specifies the container image you want to run, and the `--port=8080` option tells Kubernetes that your app is listening on port 8080. The last flag (`--generator`) does require an explanation, though. Usually, you won't use it, but you're using it here so Kubernetes creates a *ReplicationController* instead of a *Deployment*. You'll learn what *ReplicationControllers* are later in the chapter, but we won't talk about *Deployments* until chapter 9. That's why I don't want `kubectl` to create a *Deployment* yet.

As the previous command's output shows, a *ReplicationController* called `kuba` has been created. As already mentioned, we'll see what that is later in the chapter. For

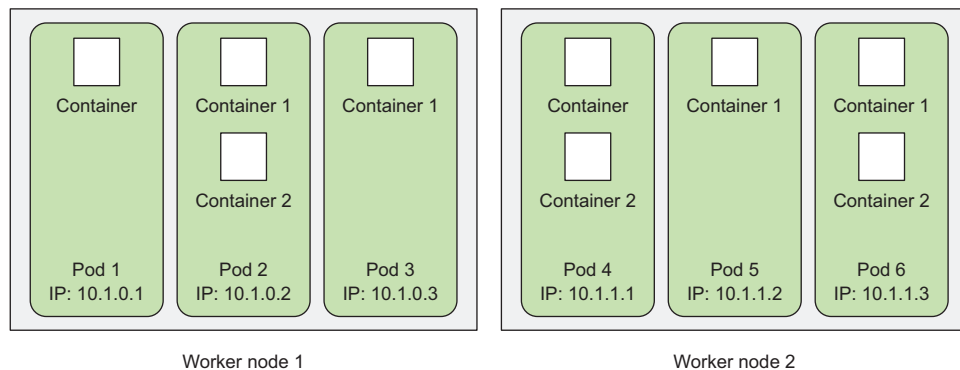
now, let's start from the bottom and focus on the container you created (you can assume a container has been created, because you specified a container image in the run command).

### INTRODUCING PODS

You may be wondering if you can see your container in a list showing all the running containers. Maybe something such as `kubectl get containers`? Well, that's not exactly how Kubernetes works. It doesn't deal with individual containers directly. Instead, it uses the concept of multiple co-located containers. This group of containers is called a Pod.

A pod is a group of one or more tightly related containers that will always run together on the same worker node and in the same Linux namespace(s). Each pod is like a separate logical machine with its own IP, hostname, processes, and so on, running a single application. The application can be a single process, running in a single container, or it can be a main application process and additional supporting processes, each running in its own container. All the containers in a pod will appear to be running on the same logical machine, whereas containers in other pods, even if they're running on the same worker node, will appear to be running on a different one.

To better understand the relationship between containers, pods, and nodes, examine figure 2.5. As you can see, each pod has its own IP and contains one or more containers, each running an application process. Pods are spread out across different worker nodes.



**Figure 2.5** The relationship between containers, pods, and physical worker nodes

### LISTING PODS

Because you can't list individual containers, since they're not standalone Kubernetes objects, can you list pods instead? Yes, you can. Let's see how to tell `kubectl` to list pods in the following listing.

**Listing 2.14** Listing pods

```
$ kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
kubia-4jfyf   0/1     Pending   0           1m
```

This is your pod. Its status is still `Pending` and the pod's single container is shown as not ready yet (this is what the `0/1` in the `READY` column means). The reason why the pod isn't running yet is because the worker node the pod has been assigned to is downloading the container image before it can run it. When the download is finished, the pod's container will be created and then the pod will transition to the `Running` state, as shown in the following listing.

**Listing 2.15** Listing pods again to see if the pod's status has changed

```
$ kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
kubia-4jfyf   1/1     Running   0           5m
```

To see more information about the pod, you can also use the `kubectl describe pod` command, like you did earlier for one of the worker nodes. If the pod stays stuck in the `Pending` status, it might be that Kubernetes can't pull the image from the registry. If you're using your own image, make sure it's marked as public on Docker Hub. To make sure the image can be pulled successfully, try pulling the image manually with the `docker pull` command on another machine.

**UNDERSTANDING WHAT HAPPENED BEHIND THE SCENES**

To help you visualize what transpired, look at figure 2.6. It shows both steps you had to perform to get a container image running inside Kubernetes. First, you built the image and pushed it to Docker Hub. This was necessary because building the image on your local machine only makes it available on your local machine, but you needed to make it accessible to the Docker daemons running on your worker nodes.

When you ran the `kubectl` command, it created a new `ReplicationController` object in the cluster by sending a `REST HTTP` request to the Kubernetes API server. The `ReplicationController` then created a new pod, which was then scheduled to one of the worker nodes by the `Scheduler`. The `Kubelet` on that node saw that the pod was scheduled to it and instructed `Docker` to pull the specified image from the registry because the image wasn't available locally. After downloading the image, `Docker` created and ran the container.

The other two nodes are displayed to show context. They didn't play any role in the process, because the pod wasn't scheduled to them.

**DEFINITION** The term scheduling means assigning the pod to a node. The pod is run immediately, not at a time in the future as the term might lead you to believe.

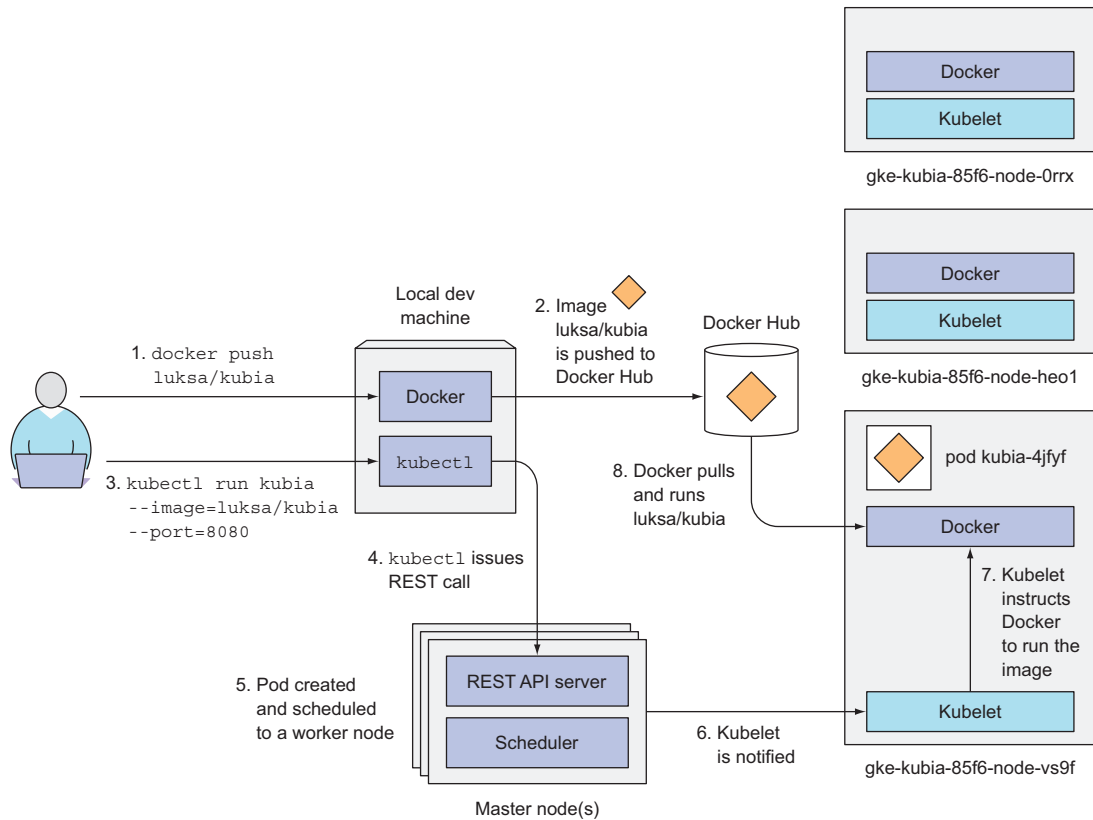


Figure 2.6 Running the luksa/kubia container image in Kubernetes

### 2.3.2 Accessing your web application

With your pod running, how do you access it? We mentioned that each pod gets its own IP address, but this address is internal to the cluster and isn't accessible from outside of it. To make the pod accessible from the outside, you'll expose it through a Service object. You'll create a special service of type `LoadBalancer`, because if you create a regular service (a `ClusterIP` service), like the pod, it would also only be accessible from inside the cluster. By creating a `LoadBalancer`-type service, an external load balancer will be created and you can connect to the pod through the load balancer's public IP.

#### CREATING A SERVICE OBJECT

To create the service, you'll tell Kubernetes to expose the ReplicationController you created earlier:

```
$ kubectl expose rc kubia --type=LoadBalancer --name kubia-http
service "kubia-http" exposed
```

**NOTE** We’re using the abbreviation `rc` instead of `replicationcontroller`. Most resource types have an abbreviation like this so you don’t have to type the full name (for example, `po` for pods, `svc` for services, and so on).

#### LISTING SERVICES

The `expose` command’s output mentions a service called `kubia-http`. Services are objects like Pods and Nodes, so you can see the newly created Service object by running the `kubectl get services` command, as shown in the following listing.

#### Listing 2.16 Listing Services

```
$ kubectl get services
```

NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	10.3.240.1	<none>	443/TCP	34m
kubia-http	10.3.246.185	<pending>	8080:31348/TCP	4s

The list shows two services. Ignore the `kubernetes` service for now and take a close look at the `kubia-http` service you created. It doesn’t have an external IP address yet, because it takes time for the load balancer to be created by the cloud infrastructure Kubernetes is running on. Once the load balancer is up, the external IP address of the service should be displayed. Let’s wait a while and list the services again, as shown in the following listing.

#### Listing 2.17 Listing services again to see if an external IP has been assigned

```
$ kubectl get svc
```

NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	10.3.240.1	<none>	443/TCP	35m
kubia-http	10.3.246.185	104.155.74.57	8080:31348/TCP	1m

Aha, there’s the external IP. Your application is now accessible at <http://104.155.74.57:8080> from anywhere in the world.

**NOTE** Minikube doesn’t support `LoadBalancer` services, so the service will never get an external IP. But you can access the service anyway through its external port. How to do that is described in the next section’s tip.

#### ACCESSING YOUR SERVICE THROUGH ITS EXTERNAL IP

You can now send requests to your pod through the service’s external IP and port:

```
$ curl 104.155.74.57:8080
You've hit kubia-4jfyf
```

Woohoo! Your app is now running somewhere in your three-node Kubernetes cluster (or a single-node cluster if you’re using Minikube). If you don’t count the steps required to set up the whole cluster, all it took was two simple commands to get your app running and to make it accessible to users across the world.

**TIP** When using Minikube, you can get the IP and port through which you can access the service by running `minikube service kuba-http`.

If you look closely, you'll see that the app is reporting the name of the pod as its hostname. As already mentioned, each pod behaves like a separate independent machine with its own IP address and hostname. Even though the application is running in the worker node's operating system, to the app it appears as though it's running on a separate machine dedicated to the app itself—no other processes are running alongside it.

### 2.3.3 The logical parts of your system

Until now, I've mostly explained the actual physical components of your system. You have three worker nodes, which are VMs running Docker and the Kubelet, and you have a master node that controls the whole system. Honestly, we don't know if a single master node is hosting all the individual components of the Kubernetes Control Plane or if they're split across multiple nodes. It doesn't really matter, because you're only interacting with the API server, which is accessible at a single endpoint.

Besides this physical view of the system, there's also a separate, logical view of it. I've already mentioned Pods, ReplicationControllers, and Services. All of them will be explained in the next few chapters, but let's quickly look at how they fit together and what roles they play in your little setup.

#### UNDERSTANDING HOW THE REPLICATIONCONTROLLER, THE POD, AND THE SERVICE FIT TOGETHER

As I've already explained, you're not creating and working with containers directly. Instead, the basic building block in Kubernetes is the pod. But, you didn't really create any pods either, at least not directly. By running the `kubectl run` command you created a ReplicationController, and this ReplicationController is what created the actual Pod object. To make that pod accessible from outside the cluster, you told Kubernetes to expose all the pods managed by that ReplicationController as a single Service. A rough picture of all three elements is presented in figure 2.7.

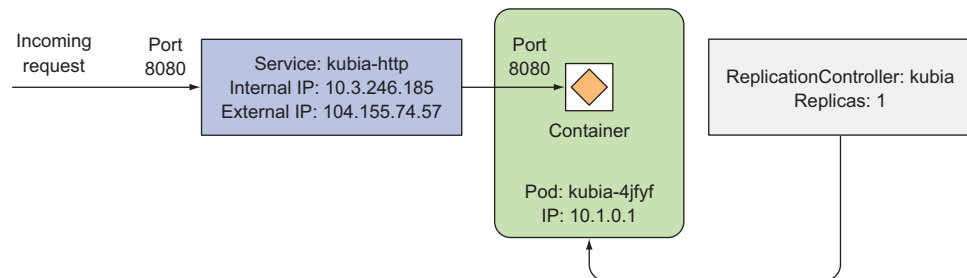


Figure 2.7 Your system consists of a ReplicationController, a Pod, and a Service.

**UNDERSTANDING THE POD AND ITS CONTAINER**

The main and most important component in your system is the pod. It contains only a single container, but generally a pod can contain as many containers as you want. Inside the container is your Node.js process, which is bound to port 8080 and is waiting for HTTP requests. The pod has its own unique private IP address and hostname.

**UNDERSTANDING THE ROLE OF THE REPLICATIONCONTROLLER**

The next component is the kubernetes ReplicationController. It makes sure there's always exactly one instance of your pod running. Generally, ReplicationControllers are used to replicate pods (that is, create multiple copies of a pod) and keep them running. In your case, you didn't specify how many pod replicas you want, so the ReplicationController created a single one. If your pod were to disappear for any reason, the ReplicationController would create a new pod to replace the missing one.

**UNDERSTANDING WHY YOU NEED A SERVICE**

The third component of your system is the kubernetes-http service. To understand why you need services, you need to learn a key detail about pods. They're ephemeral. A pod may disappear at any time—because the node it's running on has failed, because someone deleted the pod, or because the pod was evicted from an otherwise healthy node. When any of those occurs, a missing pod is replaced with a new one by the ReplicationController, as described previously. This new pod gets a different IP address from the pod it's replacing. This is where services come in—to solve the problem of ever-changing pod IP addresses, as well as exposing multiple pods at a single constant IP and port pair.

When a service is created, it gets a static IP, which never changes during the lifetime of the service. Instead of connecting to pods directly, clients should connect to the service through its constant IP address. The service makes sure one of the pods receives the connection, regardless of where the pod is currently running (and what its IP address is).

Services represent a static location for a group of one or more pods that all provide the same service. Requests coming to the IP and port of the service will be forwarded to the IP and port of one of the pods belonging to the service at that moment.

**2.3.4 Horizontally scaling the application**

You now have a running application, monitored and kept running by a ReplicationController and exposed to the world through a service. Now let's make additional magic happen.

One of the main benefits of using Kubernetes is the simplicity with which you can scale your deployments. Let's see how easy it is to scale up the number of pods. You'll increase the number of running instances to three.

Your pod is managed by a ReplicationController. Let's see it with the `kubectl get` command:

```
$ kubectl get replicationcontrollers
NAME          DESIRED    CURRENT    AGE
kubia         1          1          17m
```



### Listing all the resource types with `kubectl get`

You've been using the same basic `kubectl get` command to list things in your cluster. You've used this command to list Node, Pod, Service and ReplicationController objects. You can get a list of all the possible object types by invoking `kubectl get` without specifying the type. You can then use those types with various `kubectl` commands such as `get`, `describe`, and so on. The list also shows the abbreviations I mentioned earlier.

The list shows a single ReplicationController called `kubia`. The `DESIRED` column shows the number of pod replicas you want the ReplicationController to keep, whereas the `CURRENT` column shows the actual number of pods currently running. In your case, you wanted to have a single replica of the pod running, and exactly one replica is currently running.

### INCREASING THE DESIRED REPLICA COUNT

To scale up the number of replicas of your pod, you need to change the desired replica count on the ReplicationController like this:

```
$ kubectl scale rc kubia --replicas=3
replicationcontroller "kubia" scaled
```

You've now told Kubernetes to make sure three instances of your pod are always running. Notice that you didn't instruct Kubernetes what action to take. You didn't tell it to add two more pods. You only set the new desired number of instances and let Kubernetes determine what actions it needs to take to achieve the requested state.

This is one of the most fundamental Kubernetes principles. Instead of telling Kubernetes exactly what actions it should perform, you're only declaratively changing the desired state of the system and letting Kubernetes examine the current actual state and reconcile it with the desired state. This is true across all of Kubernetes.

### SEEING THE RESULTS OF THE SCALE-OUT

Back to your replica count increase. Let's list the ReplicationControllers again to see the updated replica count:

```
$ kubectl get rc
NAME          DESIRED  CURRENT  READY  AGE
kubia         3        3        2      17m
```

Because the actual number of pods has already been increased to three (as evident from the `CURRENT` column), listing all the pods should now show three pods instead of one:

```
$ kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
kubia-hczji   1/1     Running   0          7s
kubia-iq9y6   0/1     Pending   0          7s
kubia-4jfyf   1/1     Running   0          18m
```

As you can see, three pods exist instead of one. Two are already running, one is still pending, but should be ready in a few moments, as soon as the container image is downloaded and the container is started.

As you can see, scaling an application is incredibly simple. Once your app is running in production and a need to scale the app arises, you can add additional instances with a single command without having to install and run additional copies manually.

Keep in mind that the app itself needs to support being scaled horizontally. Kubernetes doesn't magically make your app scalable; it only makes it trivial to scale the app up or down.

### SEEING REQUESTS HIT ALL THREE PODS WHEN HITTING THE SERVICE

Because you now have multiple instances of your app running, let's see what happens if you hit the service URL again. Will you always hit the same app instance or not?

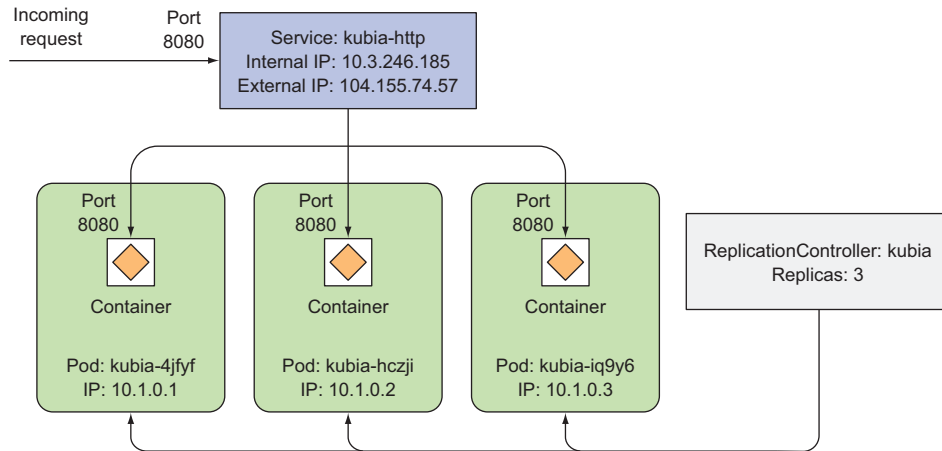
```
$ curl 104.155.74.57:8080
You've hit kubia-hczji
$ curl 104.155.74.57:8080
You've hit kubia-iq9y6
$ curl 104.155.74.57:8080
You've hit kubia-iq9y6
$ curl 104.155.74.57:8080
You've hit kubia-4jfyf
```

Requests are hitting different pods randomly. This is what services in Kubernetes do when more than one pod instance backs them. They act as a load balancer standing in front of multiple pods. When there's only one pod, services provide a static address for the single pod. Whether a service is backed by a single pod or a group of pods, those pods come and go as they're moved around the cluster, which means their IP addresses change, but the service is always there at the same address. This makes it easy for clients to connect to the pods, regardless of how many exist and how often they change location.

### VISUALIZING THE NEW STATE OF YOUR SYSTEM

Let's visualize your system again to see what's changed from before. Figure 2.8 shows the new state of your system. You still have a single service and a single ReplicationController, but you now have three instances of your pod, all managed by the ReplicationController. The service no longer sends all requests to a single pod, but spreads them across all three pods as shown in the experiment with `curl` in the previous section.

As an exercise, you can now try spinning up additional instances by increasing the ReplicationController's replica count even further and then scaling back down.



**Figure 2.8** Three instances of a pod managed by the same ReplicationController and exposed through a single service IP and port.

### 2.3.5 Examining what nodes your app is running on

You may be wondering what nodes your pods have been scheduled to. In the Kubernetes world, what node a pod is running on isn't that important, as long as it gets scheduled to a node that can provide the CPU and memory the pod needs to run properly.

Regardless of the node they're scheduled to, all the apps running inside containers have the same type of OS environment. Each pod has its own IP and can talk to any other pod, regardless of whether that other pod is also running on the same node or on a different one. Each pod is provided with the requested amount of computational resources, so whether those resources are provided by one node or another doesn't make any difference.

#### DISPLAYING THE POD IP AND THE POD'S NODE WHEN LISTING PODS

If you've been paying close attention, you probably noticed that the `kubectl get pods` command doesn't even show any information about the nodes the pods are scheduled to. This is because it's usually not an important piece of information.

But you can request additional columns to display using the `-o wide` option. When listing pods, this option shows the pod's IP and the node the pod is running on:

```
$ kubectl get pods -o wide
NAME          READY   STATUS    RESTARTS   AGE   IP        NODE
kuba-hczji    1/1     Running   0           7s    10.1.0.2  gke-kuba-85...
```

**INSPECTING OTHER DETAILS OF A POD WITH KUBECTL DESCRIBE**

You can also see the node by using the `kubectl describe` command, which shows many other details of the pod, as shown in the following listing.

**Listing 2.18 Describing a pod with `kubectl describe`**

```
$ kubectl describe pod kubia-hczji
Name:          kubia-hczji
Namespace:     default
Node:          gke-kubia-85f6-node-vs9f/10.132.0.3
Start Time:    Fri, 29 Apr 2016 14:12:33 +0200
Labels:        run=kubia
Status:        Running
IP:            10.1.0.2
Controllers:   ReplicationController/kubia
Containers:    ...
Conditions:
  Type           Status
  Ready          True
Volumes:        ...
Events:         ...
```

← Here's the node the pod has been scheduled to.

This shows, among other things, the node the pod has been scheduled to, the time when it was started, the image(s) it's running, and other useful information.

**2.3.6 Introducing the Kubernetes dashboard**

Before we wrap up this initial hands-on chapter, let's look at another way of exploring your Kubernetes cluster.

Up to now, you've only been using the `kubectl` command-line tool. If you're more into graphical web user interfaces, you'll be glad to hear that Kubernetes also comes with a nice (but still evolving) web dashboard.

The dashboard allows you to list all the Pods, ReplicationControllers, Services, and other objects deployed in your cluster, as well as to create, modify, and delete them. Figure 2.9 shows the dashboard.

Although you won't use the dashboard in this book, you can open it up any time to quickly see a graphical view of what's deployed in your cluster after you create or modify objects through `kubectl`.

**ACCESSING THE DASHBOARD WHEN RUNNING KUBERNETES IN GKE**

If you're using Google Kubernetes Engine, you can find out the URL of the dashboard through the `kubectl cluster-info` command, which we already introduced:

```
$ kubectl cluster-info | grep dashboard
kubernetes-dashboard is running at https://104.155.108.191/api/v1/proxy/
➡ namespaces/kube-system/services/kubernetes-dashboard
```

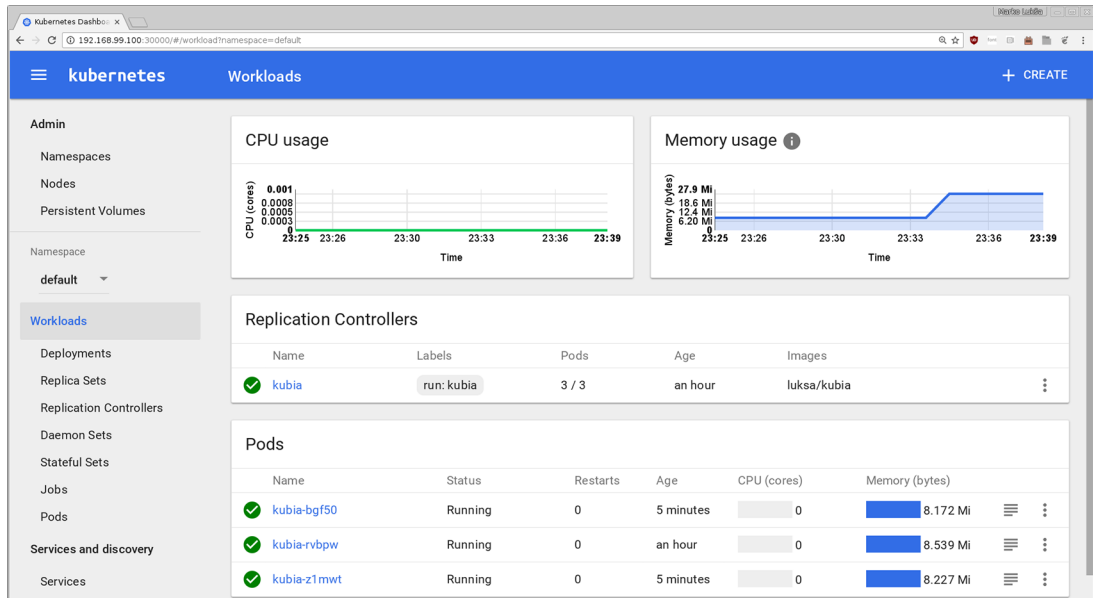


Figure 2.9 Screenshot of the Kubernetes web-based dashboard

If you open this URL in a browser, you're presented with a username and password prompt. You'll find the username and password by running the following command:

```
$ gcloud container clusters describe kubia | grep -E "(username|password):"
password: 32nEngreEJ632A12
username: admin
```

The username and password  
for the dashboard

#### ACCESSING THE DASHBOARD WHEN USING MINIKUBE

To open the dashboard in your browser when using Minikube to run your Kubernetes cluster, run the following command:

```
$ minikube dashboard
```

The dashboard will open in your default browser. Unlike with GKE, you won't need to enter any credentials to access it.

## 2.4 Summary

Hopefully, this initial hands-on chapter has shown you that Kubernetes isn't a complicated platform to use, and you're ready to learn in depth about all the things it can provide. After reading this chapter, you should now know how to

- Pull and run any publicly available container image
- Package your apps into container images and make them available to anyone by pushing the images to a remote image registry

- Enter a running container and inspect its environment
- Set up a multi-node Kubernetes cluster on Google Kubernetes Engine
- Configure an alias and tab completion for the `kubectl` command-line tool
- List and inspect Nodes, Pods, Services, and ReplicationControllers in a Kubernetes cluster
- Run a container in Kubernetes and make it accessible from outside the cluster
- Have a basic sense of how Pods, ReplicationControllers, and Services relate to one another
- Scale an app horizontally by changing the ReplicationController's replica count
- Access the web-based Kubernetes dashboard on both Minikube and GKE

# *Pods: running containers in Kubernetes*

---

## **This chapter covers**

- Creating, running, and stopping pods
- Organizing pods and other resources with labels
- Performing an operation on all pods with a specific label
- Using namespaces to split pods into non-overlapping groups
- Scheduling pods onto specific types of worker nodes

The previous chapter should have given you a rough picture of the basic components you create in Kubernetes and at least an outline of what they do. Now, we'll start reviewing all types of Kubernetes objects (or *resources*) in greater detail, so you'll understand when, how, and why to use each of them. We'll start with pods, because they're the central, most important, concept in Kubernetes. Everything else either manages, exposes, or is used by pods.

### 3.1 Introducing pods

You’ve already learned that a pod is a co-located group of containers and represents the basic building block in Kubernetes. Instead of deploying containers individually, you always deploy and operate on a pod of containers. We’re not implying that a pod always includes more than one container—it’s common for pods to contain only a single container. The key thing about pods is that when a pod does contain multiple containers, all of them are always run on a single worker node—it never spans multiple worker nodes, as shown in figure 3.1.

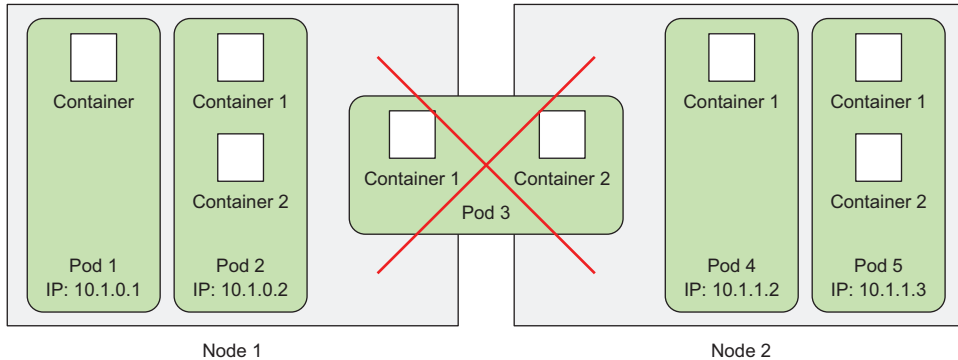


Figure 3.1 All containers of a pod run on the same node. A pod never spans two nodes.

#### 3.1.1 Understanding why we need pods

But why do we even need pods? Why can’t we use containers directly? Why would we even need to run multiple containers together? Can’t we put all our processes into a single container? We’ll answer those questions now.

##### UNDERSTANDING WHY MULTIPLE CONTAINERS ARE BETTER THAN ONE CONTAINER RUNNING MULTIPLE PROCESSES

Imagine an app consisting of multiple processes that either communicate through *IPC* (Inter-Process Communication) or through locally stored files, which requires them to run on the same machine. Because in Kubernetes you always run processes in containers and each container is much like an isolated machine, you may think it makes sense to run multiple processes in a single container, but you shouldn’t do that.

Containers are designed to run only a single process per container (unless the process itself spawns child processes). If you run multiple unrelated processes in a single container, it is your responsibility to keep all those processes running, manage their logs, and so on. For example, you’d have to include a mechanism for automatically restarting individual processes if they crash. Also, all those processes would log to the same standard output, so you’d have a hard time figuring out what process logged what.



Therefore, you need to run each process in its own container. That's how Docker and Kubernetes are meant to be used.

### 3.1.2 Understanding pods

Because you're not supposed to group multiple processes into a single container, it's obvious you need another higher-level construct that will allow you to bind containers together and manage them as a single unit. This is the reasoning behind pods.

A pod of containers allows you to run closely related processes together and provide them with (almost) the same environment as if they were all running in a single container, while keeping them somewhat isolated. This way, you get the best of both worlds. You can take advantage of all the features containers provide, while at the same time giving the processes the illusion of running together.

#### UNDERSTANDING THE PARTIAL ISOLATION BETWEEN CONTAINERS OF THE SAME POD

In the previous chapter, you learned that containers are completely isolated from each other, but now you see that you want to isolate groups of containers instead of individual ones. You want containers inside each group to share certain resources, although not all, so that they're not fully isolated. Kubernetes achieves this by configuring Docker to have all containers of a pod share the same set of Linux namespaces instead of each container having its own set.

Because all containers of a pod run under the same Network and UTS namespaces (we're talking about Linux namespaces here), they all share the same hostname and network interfaces. Similarly, all containers of a pod run under the same IPC namespace and can communicate through IPC. In the latest Kubernetes and Docker versions, they can also share the same PID namespace, but that feature isn't enabled by default.

**NOTE** When containers of the same pod use separate PID namespaces, you only see the container's own processes when running `ps aux` in the container.

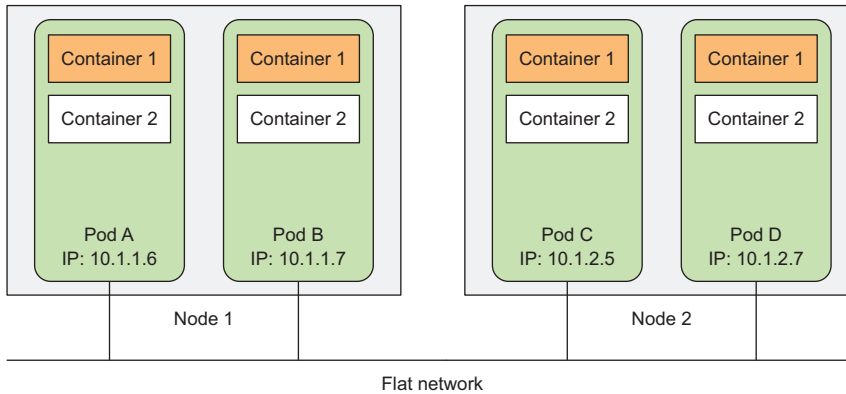
But when it comes to the filesystem, things are a little different. Because most of the container's filesystem comes from the container image, by default, the filesystem of each container is fully isolated from other containers. However, it's possible to have them share file directories using a Kubernetes concept called a *Volume*, which we'll talk about in chapter 6.

#### UNDERSTANDING HOW CONTAINERS SHARE THE SAME IP AND PORT SPACE

One thing to stress here is that because containers in a pod run in the same Network namespace, they share the same IP address and port space. This means processes running in containers of the same pod need to take care not to bind to the same port numbers or they'll run into port conflicts. But this only concerns containers in the same pod. Containers of different pods can never run into port conflicts, because each pod has a separate port space. All the containers in a pod also have the same loopback network interface, so a container can communicate with other containers in the same pod through localhost.

**INTRODUCING THE FLAT INTER-POD NETWORK**

All pods in a Kubernetes cluster reside in a single flat, shared, network-address space (shown in figure 3.2), which means every pod can access every other pod at the other pod's IP address. No NAT (Network Address Translation) gateways exist between them. When two pods send network packets between each other, they'll each see the actual IP address of the other as the source IP in the packet.



**Figure 3.2** Each pod gets a routable IP address and all other pods see the pod under that IP address.

Consequently, communication between pods is always simple. It doesn't matter if two pods are scheduled onto a single or onto different worker nodes; in both cases the containers inside those pods can communicate with each other across the flat NAT-less network, much like computers on a local area network (LAN), regardless of the actual inter-node network topology. Like a computer on a LAN, each pod gets its own IP address and is accessible from all other pods through this network established specifically for pods. This is usually achieved through an additional software-defined network layered on top of the actual network.

To sum up what's been covered in this section: pods are logical hosts and behave much like physical hosts or VMs in the non-container world. Processes running in the same pod are like processes running on the same physical or virtual machine, except that each process is encapsulated in a container.

### 3.1.3 **Organizing containers across pods properly**

You should think of pods as separate machines, but where each one hosts only a certain app. Unlike the old days, when we used to cram all sorts of apps onto the same host, we don't do that with pods. Because pods are relatively lightweight, you can have as many as you need without incurring almost any overhead. Instead of stuffing everything into a single pod, you should organize apps into multiple pods, where each one contains only tightly related components or processes.

Having said that, do you think a multi-tier application consisting of a frontend application server and a backend database should be configured as a single pod or as two pods?

#### **SPLITTING MULTI-TIER APPS INTO MULTIPLE PODS**

Although nothing is stopping you from running both the frontend server and the database in a single pod with two containers, it isn't the most appropriate way. We've said that all containers of the same pod always run co-located, but do the web server and the database really need to run on the same machine? The answer is obviously no, so you don't want to put them into a single pod. But is it wrong to do so regardless? In a way, it is.

If both the frontend and backend are in the same pod, then both will always be run on the same machine. If you have a two-node Kubernetes cluster and only this single pod, you'll only be using a single worker node and not taking advantage of the computational resources (CPU and memory) you have at your disposal on the second node. Splitting the pod into two would allow Kubernetes to schedule the frontend to one node and the backend to the other node, thereby improving the utilization of your infrastructure.

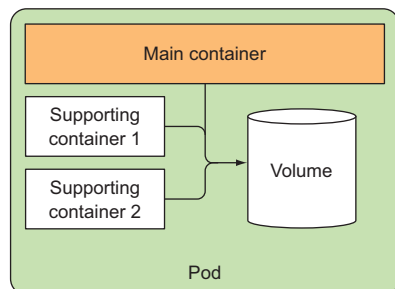
#### **SPLITTING INTO MULTIPLE PODS TO ENABLE INDIVIDUAL SCALING**

Another reason why you shouldn't put them both into a single pod is scaling. A pod is also the basic unit of scaling. Kubernetes can't horizontally scale individual containers; instead, it scales whole pods. If your pod consists of a frontend and a backend container, when you scale up the number of instances of the pod to, let's say, two, you end up with two frontend containers and two backend containers.

Usually, frontend components have completely different scaling requirements than the backends, so we tend to scale them individually. Not to mention the fact that backends such as databases are usually much harder to scale compared to (stateless) frontend web servers. If you need to scale a container individually, this is a clear indication that it needs to be deployed in a separate pod.

#### **UNDERSTANDING WHEN TO USE MULTIPLE CONTAINERS IN A POD**

The main reason to put multiple containers into a single pod is when the application consists of one main process and one or more complementary processes, as shown in figure 3.3.



**Figure 3.3** Pods should contain tightly coupled containers, usually a main container and containers that support the main one.

For example, the main container in a pod could be a web server that serves files from a certain file directory, while an additional container (a sidecar container) periodically downloads content from an external source and stores it in the web server's directory. In chapter 6 you'll see that you need to use a Kubernetes Volume that you mount into both containers.

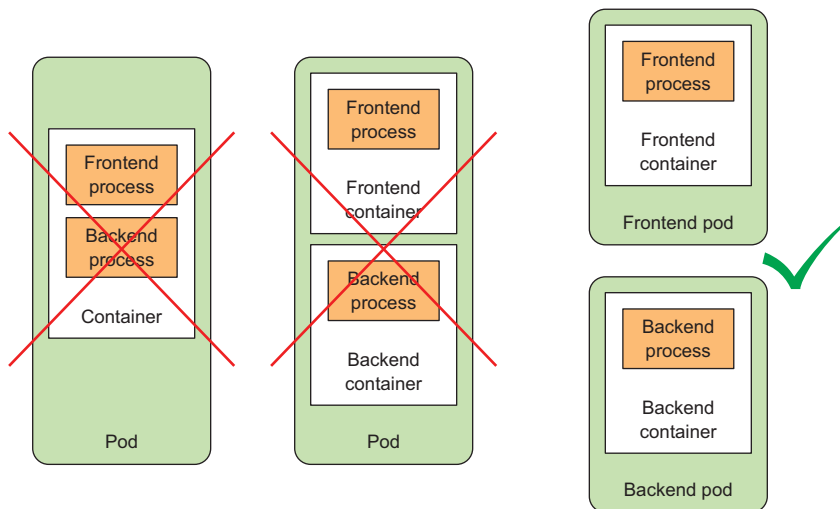
Other examples of sidecar containers include log rotators and collectors, data processors, communication adapters, and others.

#### DECIDING WHEN TO USE MULTIPLE CONTAINERS IN A POD

To recap how containers should be grouped into pods—when deciding whether to put two containers into a single pod or into two separate pods, you always need to ask yourself the following questions:

- Do they need to be run together or can they run on different hosts?
- Do they represent a single whole or are they independent components?
- Must they be scaled together or individually?

Basically, you should always gravitate toward running containers in separate pods, unless a specific reason requires them to be part of the same pod. Figure 3.4 will help you memorize this.



**Figure 3.4** A container shouldn't run multiple processes. A pod shouldn't contain multiple containers if they don't need to run on the same machine.

Although pods can contain multiple containers, to keep things simple for now, you'll only be dealing with single-container pods in this chapter. You'll see how multiple containers are used in the same pod later, in chapter 6.

## 3.2 Creating pods from YAML or JSON descriptors

Pods and other Kubernetes resources are usually created by posting a JSON or YAML manifest to the Kubernetes REST API endpoint. Also, you can use other, simpler ways of creating resources, such as the `kubectl run` command you used in the previous chapter, but they usually only allow you to configure a limited set of properties, not all. Additionally, defining all your Kubernetes objects from YAML files makes it possible to store them in a version control system, with all the benefits it brings.

To configure all aspects of each type of resource, you'll need to know and understand the Kubernetes API object definitions. You'll get to know most of them as you learn about each resource type throughout this book. We won't explain every single property, so you should also refer to the Kubernetes API reference documentation at <http://kubernetes.io/docs/reference/> when creating objects.

### 3.2.1 Examining a YAML descriptor of an existing pod

You already have some existing pods you created in the previous chapter, so let's look at what a YAML definition for one of those pods looks like. You'll use the `kubectl get` command with the `-o yaml` option to get the whole YAML definition of the pod, as shown in the following listing.

**Listing 3.1 Full YAML of a deployed pod**

```
$ kubectl get po kubia-zxzij -o yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubernetes.io/created-by: ...
  creationTimestamp: 2016-03-18T12:37:50Z
  generateName: kubia-
  labels:
    run: kubia
  name: kubia-zxzij
  namespace: default
  resourceVersion: "294"
  selfLink: /api/v1/namespaces/default/pods/kubia-zxzij
  uid: 3a564dc0-ed06-11e5-ba3b-42010af00004
spec:
  containers:
  - image: luksa/kubia
    imagePullPolicy: IfNotPresent
    name: kubia
    ports:
    - containerPort: 8080
      protocol: TCP
    resources:
      requests:
        cpu: 100m
```

Kubernetes API version used  
in this YAML descriptor

Type of Kubernetes  
object/resource

Pod metadata (name,  
labels, annotations,  
and so on)

Pod specification/  
contents (list of  
pod's containers,  
volumes, and so on)

```

    terminationMessagePath: /dev/termination-log
    volumeMounts:
    - mountPath: /var/run/secrets/kubernetes.io/serviceaccount
      name: default-token-kvcqa
      readOnly: true
  dnsPolicy: ClusterFirst
  nodeName: gke-kubia-e8fe08b8-node-txje
  restartPolicy: Always
  serviceAccount: default
  serviceAccountName: default
  terminationGracePeriodSeconds: 30
  volumes:
  - name: default-token-kvcqa
    secret:
      secretName: default-token-kvcqa
status:
  conditions:
  - lastProbeTime: null
    lastTransitionTime: null
    status: "True"
    type: Ready
  containerStatuses:
  - containerID: docker://f0276994322d247ba...
    image: luksa/kubia
    imageID: docker://4c325bcc6b40c110226b89fe...
    lastState: {}
    name: kubia
    ready: true
    restartCount: 0
    state:
      running:
        startedAt: 2016-03-18T12:46:05Z
  hostIP: 10.132.0.4
  phase: Running
  podIP: 10.0.2.3
  startTime: 2016-03-18T12:44:32Z

```

Pod specification/  
contents (list of  
pod's containers,  
volumes, and so on)

Detailed status  
of the pod and  
its containers

I know this looks complicated, but it becomes simple once you understand the basics and know how to distinguish between the important parts and the minor details. Also, you can take comfort in the fact that when creating a new pod, the YAML you need to write is much shorter, as you'll see later.

#### INTRODUCING THE MAIN PARTS OF A POD DEFINITION

The pod definition consists of a few parts. First, there's the Kubernetes API version used in the YAML and the type of resource the YAML is describing. Then, three important sections are found in almost all Kubernetes resources:

- *Metadata* includes the name, namespace, labels, and other information about the pod.
- *Spec* contains the actual description of the pod's contents, such as the pod's containers, volumes, and other data.

- *Status* contains the current information about the running pod, such as what condition the pod is in, the description and status of each container, and the pod's internal IP and other basic info.

Listing 3.1 showed a full description of a running pod, including its status. The *status* part contains read-only runtime data that shows the state of the resource at a given moment. When creating a new pod, you never need to provide the *status* part.

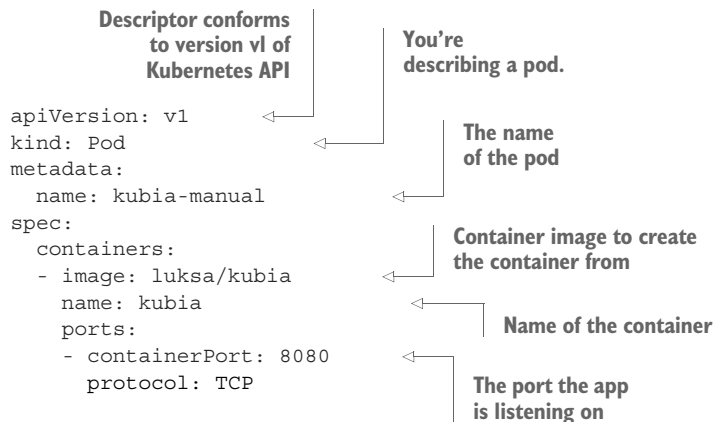
The three parts described previously show the typical structure of a Kubernetes API object. As you'll see throughout the book, all other objects have the same anatomy. This makes understanding new objects relatively easy.

Going through all the individual properties in the previous YAML doesn't make much sense, so, instead, let's see what the most basic YAML for creating a pod looks like.

### 3.2.2 Creating a simple YAML descriptor for a pod

You're going to create a file called `kubia-manual.yaml` (you can create it in any directory you want), or download the book's code archive, where you'll find the file inside the `Chapter03` directory. The following listing shows the entire contents of the file.

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



I'm sure you'll agree this is much simpler than the definition in listing 3.1. Let's examine this descriptor in detail. It conforms to the v1 version of the Kubernetes API. The type of resource you're describing is a pod, with the name `kubia-manual`. The pod consists of a single container based on the `luksa/kubia` image. You've also given a name to the container and indicated that it's listening on port 8080.

#### SPECIFYING CONTAINER PORTS

Specifying ports in the pod definition is purely informational. Omitting them has no effect on whether clients can connect to the pod through the port or not. If the con-

tainer is accepting connections through a port bound to the 0.0.0.0 address, other pods can always connect to it, even if the port isn't listed in the pod spec explicitly. But it makes sense to define the ports explicitly so that everyone using your cluster can quickly see what ports each pod exposes. Explicitly defining ports also allows you to assign a name to each port, which can come in handy, as you'll see later in the book.

### Using `kubectl explain` to discover possible API object fields

When preparing a manifest, you can either turn to the Kubernetes reference documentation at <http://kubernetes.io/docs/api> to see which attributes are supported by each API object, or you can use the `kubectl explain` command.

For example, when creating a pod manifest from scratch, you can start by asking `kubectl` to explain pods:

```
$ kubectl explain pods
DESCRIPTION:
Pod is a collection of containers that can run on a host. This resource
    is created by clients and scheduled onto hosts.

FIELDS:
  kind          <string>
    Kind is a string value representing the REST resource this object
    represents...

  metadata      <Object>
    Standard object's metadata...

  spec          <Object>
    Specification of the desired behavior of the pod...

  status        <Object>
    Most recently observed status of the pod. This data may not be up to
    date...
```

`Kubectl` prints out the explanation of the object and lists the attributes the object can contain. You can then drill deeper to find out more about each attribute. For example, you can examine the `spec` attribute like this:

```
$ kubectl explain pod.spec
RESOURCE: spec <Object>

DESCRIPTION:
  Specification of the desired behavior of the pod...
  podSpec is a description of a pod.

FIELDS:
  hostPID      <boolean>
    Use the host's pid namespace. Optional: Default to false.

  ...

  volumes      <[]Object>
    List of volumes that can be mounted by containers belonging to the
    pod.
```



```
Containers <[]Object> -required-
List of containers belonging to the pod. Containers cannot currently
Be added or removed. There must be at least one container in a pod.
Cannot be updated. More info:
http://releases.k8s.io/release-1.4/docs/user-guide/containers.md
```

### 3.2.3 Using kubectl create to create the pod

To create the pod from your YAML file, use the `kubectl create` command:

```
$ kubectl create -f kubia-manual.yaml
pod "kubia-manual" created
```

The `kubectl create -f` command is used for creating any resource (not only pods) from a YAML or JSON file.

#### RETRIEVING THE WHOLE DEFINITION OF A RUNNING POD

After creating the pod, you can ask Kubernetes for the full YAML of the pod. You'll see it's similar to the YAML you saw earlier. You'll learn about the additional fields appearing in the returned definition in the next sections. Go ahead and use the following command to see the full descriptor of the pod:

```
$ kubectl get po kubia-manual -o yaml
```

If you're more into JSON, you can also tell `kubectl` to return JSON instead of YAML like this (this works even if you used YAML to create the pod):

```
$ kubectl get po kubia-manual -o json
```

#### SEEING YOUR NEWLY CREATED POD IN THE LIST OF PODS

Your pod has been created, but how do you know if it's running? Let's list pods to see their statuses:

```
$ kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
kubia-manual  1/1     Running   0           32s
kubia-zxzijs  1/1     Running   0           1d
```

There's your `kubia-manual` pod. Its status shows that it's running. If you're like me, you'll probably want to confirm that's true by talking to the pod. You'll do that in a minute. First, you'll look at the app's log to check for any errors.

### 3.2.4 Viewing application logs

Your little Node.js application logs to the process's standard output. Containerized applications usually log to the standard output and standard error stream instead of

writing their logs to files. This is to allow users to view logs of different applications in a simple, standard way.

The container runtime (Docker in your case) redirects those streams to files and allows you to get the container's log by running

```
$ docker logs <container id>
```

You could use `ssh` to log into the node where your pod is running and retrieve its logs with `docker logs`, but Kubernetes provides an easier way.

#### RETRIEVING A POD'S LOG WITH KUBECTL LOGS

To see your pod's log (more precisely, the container's log) you run the following command on your local machine (no need to `ssh` anywhere):

```
$ kubectl logs kuba-manual
Kuba server starting...
```

You haven't sent any web requests to your Node.js app, so the log only shows a single log statement about the server starting up. As you can see, retrieving logs of an application running in Kubernetes is incredibly simple if the pod only contains a single container.

**NOTE** Container logs are automatically rotated daily and every time the log file reaches 10MB in size. The `kubectl logs` command only shows the log entries from the last rotation.

#### SPECIFYING THE CONTAINER NAME WHEN GETTING LOGS OF A MULTI-CONTAINER POD

If your pod includes multiple containers, you have to explicitly specify the container name by including the `-c <container name>` option when running `kubectl logs`. In your `kuba-manual` pod, you set the container's name to `kuba`, so if additional containers exist in the pod, you'd have to get its logs like this:

```
$ kubectl logs kuba-manual -c kuba
Kuba server starting...
```

Note that you can only retrieve container logs of pods that are still in existence. When a pod is deleted, its logs are also deleted. To make a pod's logs available even after the pod is deleted, you need to set up centralized, cluster-wide logging, which stores all the logs into a central store. Chapter 17 explains how centralized logging works.

### 3.2.5 *Sending requests to the pod*

The pod is now running—at least that's what `kubectl get` and your app's log say. But how do you see it in action? In the previous chapter, you used the `kubectl expose` command to create a service to gain access to the pod externally. You're not going to do that now, because a whole chapter is dedicated to services, and you have other ways of connecting to a pod for testing and debugging purposes. One of them is through *port forwarding*.

### FORWARDING A LOCAL NETWORK PORT TO A PORT IN THE POD

When you want to talk to a specific pod without going through a service (for debugging or other reasons), Kubernetes allows you to configure port forwarding to the pod. This is done through the `kubectl port-forward` command. The following command will forward your machine's local port 8888 to port 8080 of your `kubia-manual` pod:

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

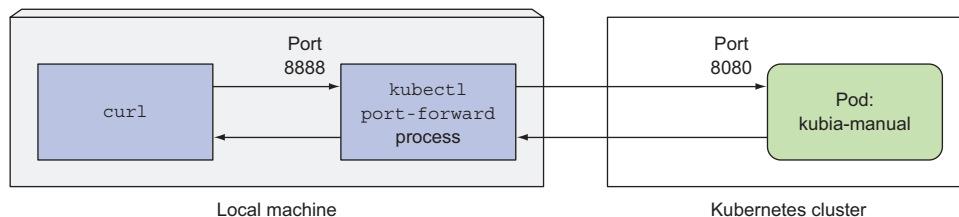
The port forwarder is running and you can now connect to your pod through the local port.

### CONNECTING TO THE POD THROUGH THE PORT FORWARDER

In a different terminal, you can now use `curl` to send an HTTP request to your pod through the `kubectl port-forward` proxy running on `localhost:8888`:

```
$ curl localhost:8888
You've hit kubia-manual
```

Figure 3.5 shows an overly simplified view of what happens when you send the request. In reality, a couple of additional components sit between the `kubectl` process and the pod, but they aren't relevant right now.



**Figure 3.5** A simplified view of what happens when you use `curl` with `kubectl port-forward`

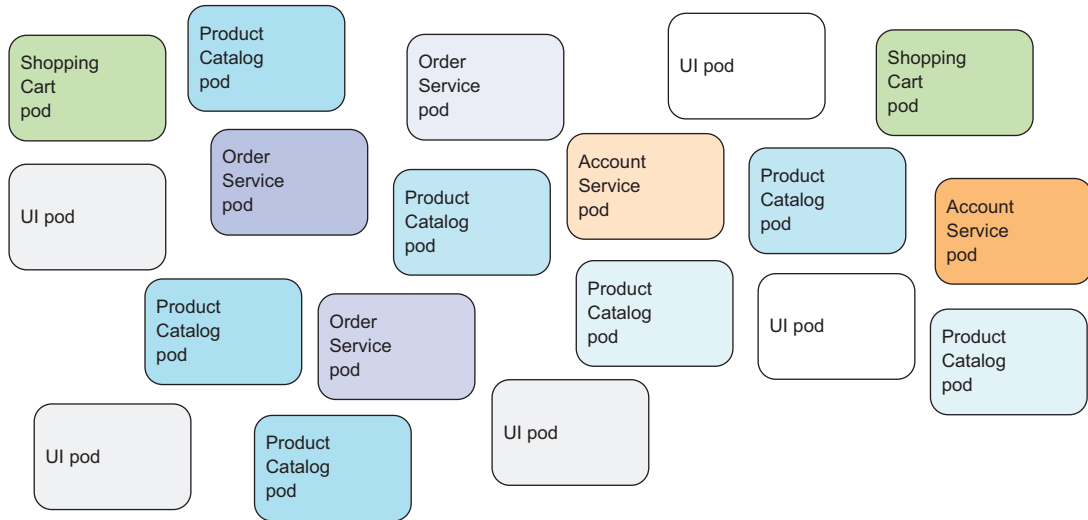
Using port forwarding like this is an effective way to test an individual pod. You'll learn about other similar methods throughout the book.

## 3.3 Organizing pods with labels

At this point, you have two pods running in your cluster. When deploying actual applications, most users will end up running many more pods. As the number of pods increases, the need for categorizing them into subsets becomes more and more evident.

For example, with microservices architectures, the number of deployed microservices can easily exceed 20 or more. Those components will probably be replicated

(multiple copies of the same component will be deployed) and multiple versions or releases (stable, beta, canary, and so on) will run concurrently. This can lead to hundreds of pods in the system. Without a mechanism for organizing them, you end up with a big, incomprehensible mess, such as the one shown in figure 3.6. The figure shows pods of multiple microservices, with several running multiple replicas, and others running different releases of the same microservice.



**Figure 3.6** Uncategorized pods in a microservices architecture

It's evident you need a way of organizing them into smaller groups based on arbitrary criteria, so every developer and system administrator dealing with your system can easily see which pod is which. And you'll want to operate on every pod belonging to a certain group with a single action instead of having to perform the action for each pod individually.

Organizing pods and all other Kubernetes objects is done through *labels*.

### 3.3.1 Introducing labels

Labels are a simple, yet incredibly powerful, Kubernetes feature for organizing not only pods, but all other Kubernetes resources. A label is an arbitrary key-value pair you attach to a resource, which is then utilized when selecting resources using *label selectors* (resources are filtered based on whether they include the label specified in the selector). A resource can have more than one label, as long as the keys of those labels are unique within that resource. You usually attach labels to resources when you create them, but you can also add additional labels or even modify the values of existing labels later without having to recreate the resource.

Let's turn back to the microservices example from figure 3.6. By adding labels to those pods, you get a much-better-organized system that everyone can easily make sense of. Each pod is labeled with two labels:

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

**DEFINITION** A canary release is when you deploy a new version of an application next to the stable version, and only let a small fraction of users hit the new version to see how it behaves before rolling it out to all users. This prevents bad releases from being exposed to too many users.

By adding these two labels, you've essentially organized your pods into two dimensions (horizontally by app and vertically by release), as shown in figure 3.7.

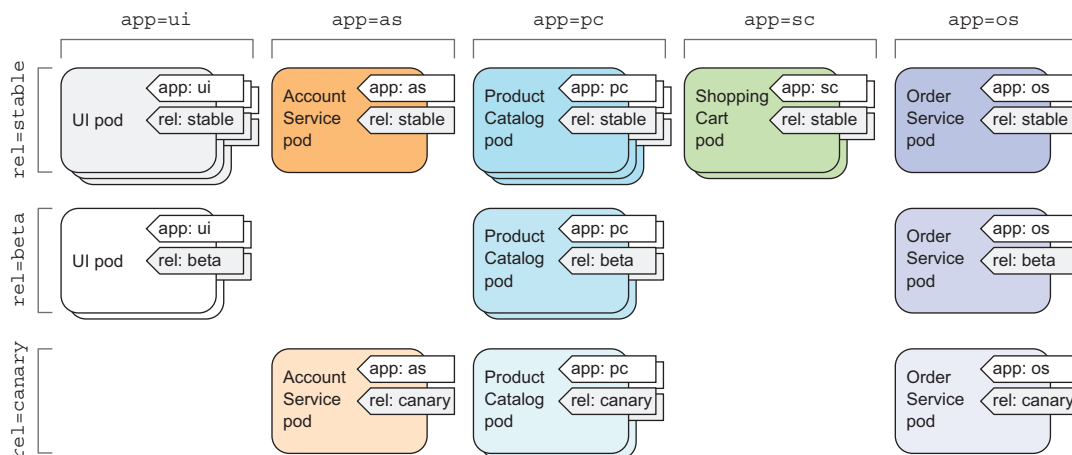


Figure 3.7 Organizing pods in a microservices architecture with pod labels

Every developer or ops person with access to your cluster can now easily see the system's structure and where each pod fits in by looking at the pod's labels.

### 3.3.2 Specifying labels when creating a pod

Now, you'll see labels in action by creating a new pod with two labels. Create a new file called `kubia-manual-with-labels.yaml` with the contents of the following listing.

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

```
apiVersion: v1
kind: Pod
metadata:
  name: kubia-manual-v2
```

```

labels:
  creation_method: manual
  env: prod
spec:
  containers:
  - image: luksa/kubia
    name: kubia
    ports:
    - containerPort: 8080
      protocol: TCP

```

Two labels are attached to the pod.

You've included the labels `creation_method=manual` and `env=prod` in the `data.labels` section. You'll create this pod now:

```

$ kubectl create -f kubia-manual-with-labels.yaml
pod "kubia-manual-v2" created

```

The `kubectl get pods` command doesn't list any labels by default, but you can see them by using the `--show-labels` switch:

```

$ kubectl get po --show-labels
NAME             READY   STATUS    RESTARTS   AGE   LABELS
kubia-manual     1/1     Running   0           16m   <none>
kubia-manual-v2  1/1     Running   0           2m    creat_method=manual,env=prod
kubia-zxzij      1/1     Running   0           1d    run=kubia

```

Instead of listing all labels, if you're only interested in certain labels, you can specify them with the `-L` switch and have each displayed in its own column. List pods again and show the columns for the two labels you've attached to your `kubia-manual-v2` pod:

```

$ kubectl get po -L creation_method,env
NAME             READY   STATUS    RESTARTS   AGE   CREATION_METHOD   ENV
kubia-manual     1/1     Running   0           16m   <none>             <none>
kubia-manual-v2  1/1     Running   0           2m    manual             prod
kubia-zxzij      1/1     Running   0           1d    <none>             <none>

```

### 3.3.3 *Modifying labels of existing pods*

Labels can also be added to and modified on existing pods. Because the `kubia-manual` pod was also created manually, let's add the `creation_method=manual` label to it:

```

$ kubectl label po kubia-manual creation_method=manual
pod "kubia-manual" labeled

```

Now, let's also change the `env=prod` label to `env=debug` on the `kubia-manual-v2` pod, to see how existing labels can be changed.

**NOTE** You need to use the `--overwrite` option when changing existing labels.

```

$ kubectl label po kubia-manual-v2 env=debug --overwrite
pod "kubia-manual-v2" labeled

```

List the pods again to see the updated labels:

```
$ kubectl get po -L creation_method,env
```

NAME	READY	STATUS	RESTARTS	AGE	CREATION_METHOD	ENV
kubia-manual	1/1	Running	0	16m	<b>manual</b>	<none>
kubia-manual-v2	1/1	Running	0	2m	manual	<b>debug</b>
kubia-zxzij	1/1	Running	0	1d	<none>	<none>

As you can see, attaching labels to resources is trivial, and so is changing them on existing resources. It may not be evident right now, but this is an incredibly powerful feature, as you'll see in the next chapter. But first, let's see what you can do with these labels, in addition to displaying them when listing pods.

### 3.4 Listing subsets of pods through label selectors

Attaching labels to resources so you can see the labels next to each resource when listing them isn't that interesting. But labels go hand in hand with *label selectors*. Label selectors allow you to select a subset of pods tagged with certain labels and perform an operation on those pods. A label selector is a criterion, which filters resources based on whether they include a certain label with a certain value.

A label selector can select resources based on whether the resource

- Contains (or doesn't contain) a label with a certain key
- Contains a label with a certain key and value
- Contains a label with a certain key, but with a value not equal to the one you specify

#### 3.4.1 Listing pods using a label selector

Let's use label selectors on the pods you've created so far. To see all pods you created manually (you labeled them with `creation_method=manual`), do the following:

```
$ kubectl get po -l creation_method=manual
```

NAME	READY	STATUS	RESTARTS	AGE
kubia-manual	1/1	Running	0	51m
kubia-manual-v2	1/1	Running	0	37m

To list all pods that include the `env` label, whatever its value is:

```
$ kubectl get po -l env
```

NAME	READY	STATUS	RESTARTS	AGE
kubia-manual-v2	1/1	Running	0	37m

And those that don't have the `env` label:

```
$ kubectl get po -l '!env'
```

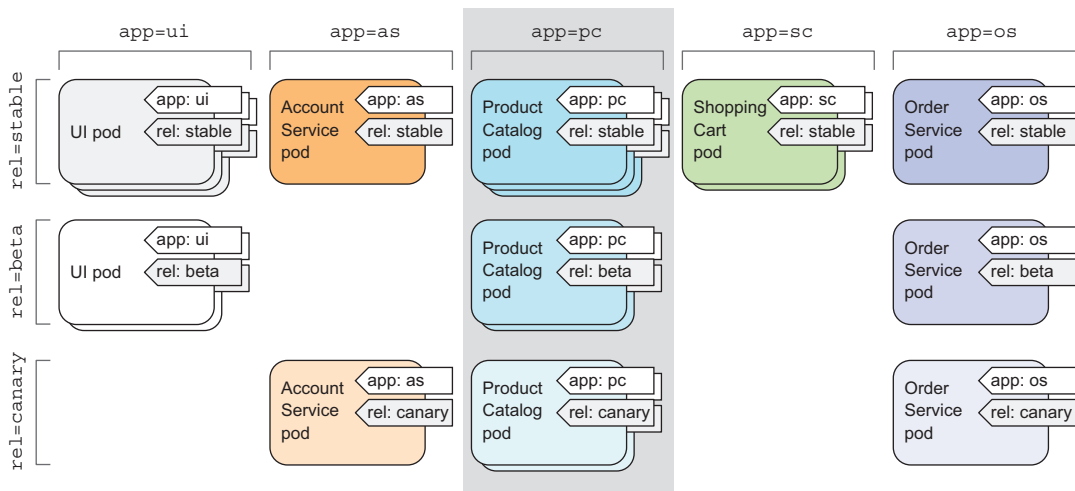
NAME	READY	STATUS	RESTARTS	AGE
kubia-manual	1/1	Running	0	51m
kubia-zxzij	1/1	Running	0	10d

**NOTE** Make sure to use single quotes around `!env`, so the bash shell doesn't evaluate the exclamation mark.

Similarly, you could also match pods with the following label selectors:

- `creation_method!=manual` to select pods with the `creation_method` label with any value other than `manual`
- `env in (prod,devel)` to select pods with the `env` label set to either `prod` or `devel`
- `env notin (prod,devel)` to select pods with the `env` label set to any value other than `prod` or `devel`

Turning back to the pods in the microservices-oriented architecture example, you could select all pods that are part of the product catalog microservice by using the `app=pc` label selector (shown in the following figure).



**Figure 3.8** Selecting the product catalog microservice pods using the “`app=pc`” label selector

### 3.4.2 Using multiple conditions in a label selector

A selector can also include multiple comma-separated criteria. Resources need to match all of them to match the selector. If, for example, you want to select only pods running the beta release of the product catalog microservice, you'd use the following selector: `app=pc,rel=beta` (visualized in figure 3.9).

Label selectors aren't useful only for listing pods, but also for performing actions on a subset of all pods. For example, later in the chapter, you'll see how to use label selectors to delete multiple pods at once. But label selectors aren't used only by `kubectl`. They're also used internally, as you'll see next.



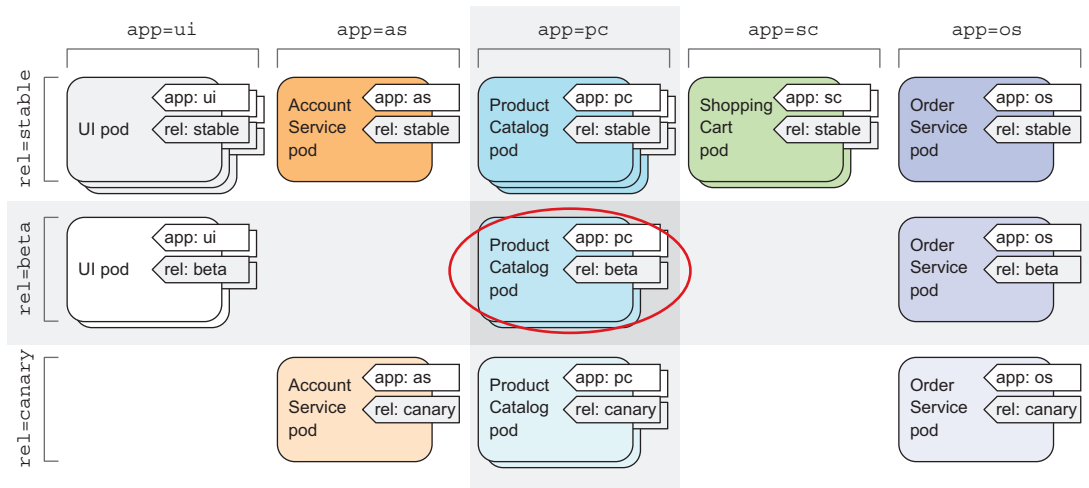


Figure 3.9 Selecting pods with multiple label selectors

### 3.5 Using labels and selectors to constrain pod scheduling

All the pods you’ve created so far have been scheduled pretty much randomly across your worker nodes. As I’ve mentioned in the previous chapter, this is the proper way of working in a Kubernetes cluster. Because Kubernetes exposes all the nodes in the cluster as a single, large deployment platform, it shouldn’t matter to you what node a pod is scheduled to. Because each pod gets the exact amount of computational resources it requests (CPU, memory, and so on) and its accessibility from other pods isn’t at all affected by the node the pod is scheduled to, usually there shouldn’t be any need for you to tell Kubernetes exactly where to schedule your pods.

Certain cases exist, however, where you’ll want to have at least a little say in where a pod should be scheduled. A good example is when your hardware infrastructure isn’t homogenous. If part of your worker nodes have spinning hard drives, whereas others have SSDs, you may want to schedule certain pods to one group of nodes and the rest to the other. Another example is when you need to schedule pods performing intensive GPU-based computation only to nodes that provide the required GPU acceleration.

You never want to say specifically what node a pod should be scheduled to, because that would couple the application to the infrastructure, whereas the whole idea of Kubernetes is hiding the actual infrastructure from the apps that run on it. But if you want to have a say in where a pod should be scheduled, instead of specifying an exact node, you should describe the node requirements and then let Kubernetes select a node that matches those requirements. This can be done through node labels and node label selectors.

### 3.5.1 Using labels for categorizing worker nodes

As you learned earlier, pods aren't the only Kubernetes resource type that you can attach a label to. Labels can be attached to any Kubernetes object, including nodes. Usually, when the ops team adds a new node to the cluster, they'll categorize the node by attaching labels specifying the type of hardware the node provides or anything else that may come in handy when scheduling pods.

Let's imagine one of the nodes in your cluster contains a GPU meant to be used for general-purpose GPU computing. You want to add a label to the node showing this feature. You're going to add the label `gpu=true` to one of your nodes (pick one out of the list returned by `kubectl get nodes`):

```
$ kubectl label node gke-kubia-85f6-node-0rrx gpu=true
node "gke-kubia-85f6-node-0rrx" labeled
```

Now you can use a label selector when listing the nodes, like you did before with pods. List only nodes that include the label `gpu=true`:

```
$ kubectl get nodes -l gpu=true
NAME                                STATUS AGE
gke-kubia-85f6-node-0rrx    Ready  1d
```

As expected, only one node has this label. You can also try listing all the nodes and tell `kubectl` to display an additional column showing the values of each node's `gpu` label (`kubectl get nodes -L gpu`).

### 3.5.2 Scheduling pods to specific nodes

Now imagine you want to deploy a new pod that needs a GPU to perform its work. To ask the scheduler to only choose among the nodes that provide a GPU, you'll add a node selector to the pod's YAML. Create a file called `kubia-gpu.yaml` with the following listing's contents and then use `kubectl create -f kubia-gpu.yaml` to create the pod.

**Listing 3.4** Using a label selector to schedule a pod to a specific node: `kubia-gpu.yaml`

```
apiVersion: v1
kind: Pod
metadata:
  name: kubia-gpu
spec:
  nodeSelector:
    gpu: "true"
  containers:
  - image: luksa/kubia
    name: kubia
```

**nodeSelector** tells Kubernetes to deploy this pod only to nodes containing the `gpu=true` label.

You've added a `nodeSelector` field under the `spec` section. When you create the pod, the scheduler will only choose among the nodes that contain the `gpu=true` label (which is only a single node in your case).

### 3.5.3 Scheduling to one specific node

Similarly, you could also schedule a pod to an exact node, because each node also has a unique label with the key `kubernetes.io/hostname` and value set to the actual hostname of the node. But setting the `nodeSelector` to a specific node by the `hostname` label may lead to the pod being unschedulable if the node is offline. You shouldn't think in terms of individual nodes. Always think about logical groups of nodes that satisfy certain criteria specified through label selectors.

This was a quick demonstration of how labels and label selectors work and how they can be used to influence the operation of Kubernetes. The importance and usefulness of label selectors will become even more evident when we talk about Replication-Controllers and Services in the next two chapters.

**NOTE** Additional ways of influencing which node a pod is scheduled to are covered in chapter 16.

## 3.6 Annotating pods

In addition to labels, pods and other objects can also contain *annotations*. Annotations are also key-value pairs, so in essence, they're similar to labels, but they aren't meant to hold identifying information. They can't be used to group objects the way labels can. While objects can be selected through label selectors, there's no such thing as an annotation selector.

On the other hand, annotations can hold much larger pieces of information and are primarily meant to be used by tools. Certain annotations are automatically added to objects by Kubernetes, but others are added by users manually.

Annotations are also commonly used when introducing new features to Kubernetes. Usually, alpha and beta versions of new features don't introduce any new fields to API objects. Annotations are used instead of fields, and then once the required API changes have become clear and been agreed upon by the Kubernetes developers, new fields are introduced and the related annotations deprecated.

A great use of annotations is adding descriptions for each pod or other API object, so that everyone using the cluster can quickly look up information about each individual object. For example, an annotation used to specify the name of the person who created the object can make collaboration between everyone working on the cluster much easier.

### 3.6.1 Looking up an object's annotations

Let's see an example of an annotation that Kubernetes added automatically to the pod you created in the previous chapter. To see the annotations, you'll need to

request the full YAML of the pod or use the `kubectl describe` command. You'll use the first option in the following listing.

#### Listing 3.5 A pod's annotations

```
$ kubectl get po kubia-zxzij -o yaml
apiVersion: v1
kind: pod
metadata:
  annotations:
    kubernetes.io/created-by: |
      {"kind": "SerializedReference", "apiVersion": "v1",
      "reference": {"kind": "ReplicationController", "namespace": "default", ...
```

Without going into too many details, as you can see, the `kubernetes.io/created-by` annotation holds JSON data about the object that created the pod. That's not something you'd want to put into a label. Labels should be short, whereas annotations can contain relatively large blobs of data (up to 256 KB in total).

**NOTE** The `kubernetes.io/created-by` annotations was deprecated in version 1.8 and will be removed in 1.9, so you will no longer see it in the YAML.

### 3.6.2 Adding and modifying annotations

Annotations can obviously be added to pods at creation time, the same way labels can. They can also be added to or modified on existing pods later. The simplest way to add an annotation to an existing object is through the `kubectl annotate` command.

You'll try adding an annotation to your `kubia-manual` pod now:

```
$ kubectl annotate pod kubia-manual mycompany.com/someannotation="foo bar"
pod "kubia-manual" annotated
```

You added the annotation `mycompany.com/someannotation` with the value `foo bar`. It's a good idea to use this format for annotation keys to prevent key collisions. When different tools or libraries add annotations to objects, they may accidentally override each other's annotations if they don't use unique prefixes like you did here.

You can use `kubectl describe` to see the annotation you added:

```
$ kubectl describe pod kubia-manual
...
Annotations:    mycompany.com/someannotation=foo bar
...
```

## 3.7 Using namespaces to group resources

Let's turn back to labels for a moment. We've seen how they organize pods and other objects into groups. Because each object can have multiple labels, those groups of objects can overlap. Plus, when working with the cluster (through `kubectl` for example), if you don't explicitly specify a label selector, you'll always see all objects.

But what about times when you want to split objects into separate, non-overlapping groups? You may want to only operate inside one group at a time. For this and other reasons, Kubernetes also groups objects into namespaces. These aren't the Linux namespaces we talked about in chapter 2, which are used to isolate processes from each other. Kubernetes namespaces provide a scope for objects names. Instead of having all your resources in one single namespace, you can split them into multiple namespaces, which also allows you to use the same resource names multiple times (across different namespaces).

### 3.7.1 Understanding the need for namespaces

Using multiple namespaces allows you to split complex systems with numerous components into smaller distinct groups. They can also be used for separating resources in a multi-tenant environment, splitting up resources into production, development, and QA environments, or in any other way you may need. Resource names only need to be unique within a namespace. Two different namespaces can contain resources of the same name. But, while most types of resources are namespaced, a few aren't. One of them is the Node resource, which is global and not tied to a single namespace. You'll learn about other cluster-level resources in later chapters.

Let's see how to use namespaces now.

### 3.7.2 Discovering other namespaces and their pods

First, let's list all namespaces in your cluster:

```
$ kubectl get ns
```

NAME	LABELS	STATUS	AGE
default	<none>	Active	1h
kube-public	<none>	Active	1h
kube-system	<none>	Active	1h

Up to this point, you've operated only in the default namespace. When listing resources with the `kubectl get` command, you've never specified the namespace explicitly, so `kubectl` always defaulted to the default namespace, showing you only the objects in that namespace. But as you can see from the list, the `kube-public` and the `kube-system` namespaces also exist. Let's look at the pods that belong to the `kube-system` namespace, by telling `kubectl` to list pods in that namespace only:

```
$ kubectl get po --namespace kube-system
```

NAME	READY	STATUS	RESTARTS	AGE
fluentd-cloud-kubia-e8fe-node-txje	1/1	Running	0	1h
heapster-v11-fz1ge	1/1	Running	0	1h
kube-dns-v9-p8a4t	0/4	Pending	0	1h
kube-ui-v4-kdlai	1/1	Running	0	1h
17-1b-controller-v0.5.2-bue96	2/2	Running	92	1h

**TIP** You can also use `-n` instead of `--namespace`.

You'll learn about these pods later in the book (don't worry if the pods shown here don't match the ones on your system exactly). It's clear from the name of the namespace that these are resources related to the Kubernetes system itself. By having them in this separate namespace, it keeps everything nicely organized. If they were all in the default namespace, mixed in with the resources you create yourself, you'd have a hard time seeing what belongs where, and you might inadvertently delete system resources.

Namespaces enable you to separate resources that don't belong together into non-overlapping groups. If several users or groups of users are using the same Kubernetes cluster, and they each manage their own distinct set of resources, they should each use their own namespace. This way, they don't need to take any special care not to inadvertently modify or delete the other users' resources and don't need to concern themselves with name conflicts, because namespaces provide a scope for resource names, as has already been mentioned.

Besides isolating resources, namespaces are also used for allowing only certain users access to particular resources and even for limiting the amount of computational resources available to individual users. You'll learn about this in chapters 12 through 14.

### 3.7.3 *Creating a namespace*

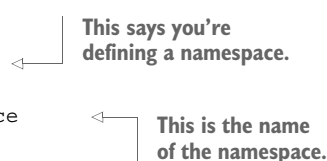
A namespace is a Kubernetes resource like any other, so you can create it by posting a YAML file to the Kubernetes API server. Let's see how to do this now.

#### CREATING A NAMESPACE FROM A YAML FILE

First, create a `custom-namespace.yaml` file with the following listing's contents (you'll find the file in the book's code archive).

**Listing 3.6** A YAML definition of a namespace: `custom-namespace.yaml`

```
apiVersion: v1
kind: Namespace
metadata:
  name: custom-namespace
```



This says you're defining a namespace.

This is the name of the namespace.

Now, use `kubectl` to post the file to the Kubernetes API server:

```
$ kubectl create -f custom-namespace.yaml
namespace "custom-namespace" created
```

#### CREATING A NAMESPACE WITH `KUBECTL CREATE NAMESPACE`

Although writing a file like the previous one isn't a big deal, it's still a hassle. Luckily, you can also create namespaces with the dedicated `kubectl create namespace` command, which is quicker than writing a YAML file. By having you create a YAML manifest for the namespace, I wanted to reinforce the idea that everything in Kubernetes

has a corresponding API object that you can create, read, update, and delete by posting a YAML manifest to the API server.

You could have created the namespace like this:

```
$ kubectl create namespace custom-namespace
namespace "custom-namespace" created
```

**NOTE** Although most objects' names must conform to the naming conventions specified in RFC 1035 (Domain names), which means they may contain only letters, digits, dashes, and dots, namespaces (and a few others) aren't allowed to contain dots.

### 3.7.4 Managing objects in other namespaces

To create resources in the namespace you've created, either add a `namespace: custom-namespace` entry to the metadata section, or specify the namespace when creating the resource with the `kubectl create` command:

```
$ kubectl create -f kuba-manual.yaml -n custom-namespace
pod "kuba-manual" created
```

You now have two pods with the same name (`kuba-manual`). One is in the default namespace, and the other is in your `custom-namespace`.

When listing, describing, modifying, or deleting objects in other namespaces, you need to pass the `--namespace` (or `-n`) flag to `kubectl`. If you don't specify the namespace, `kubectl` performs the action in the default namespace configured in the current `kubectl` context. The current context's namespace and the current context itself can be changed through `kubectl config` commands. To learn more about managing `kubectl` contexts, refer to appendix A.

**TIP** To quickly switch to a different namespace, you can set up the following alias: `alias kcd='kubectl config set-context $(kubectl config current-context) --namespace '`. You can then switch between namespaces using `kcd some-namespace`.

### 3.7.5 Understanding the isolation provided by namespaces

To wrap up this section about namespaces, let me explain what namespaces don't provide—at least not out of the box. Although namespaces allow you to isolate objects into distinct groups, which allows you to operate only on those belonging to the specified namespace, they don't provide any kind of isolation of running objects.

For example, you may think that when different users deploy pods across different namespaces, those pods are isolated from each other and can't communicate, but that's not necessarily the case. Whether namespaces provide network isolation depends on which networking solution is deployed with Kubernetes. When the solution doesn't provide inter-namespace network isolation, if a pod in namespace `foo` knows the IP

address of a pod in namespace bar, there is nothing preventing it from sending traffic, such as HTTP requests, to the other pod.

### 3.8 *Stopping and removing pods*

You've created a number of pods, which should all still be running. You have four pods running in the default namespace and one pod in custom-namespace. You're going to stop all of them now, because you don't need them anymore.

#### 3.8.1 *Deleting a pod by name*

First, delete the kuba-gpu pod by name:

```
$ kubectl delete po kuba-gpu
pod "kuba-gpu" deleted
```

By deleting a pod, you're instructing Kubernetes to terminate all the containers that are part of that pod. Kubernetes sends a SIGTERM signal to the process and waits a certain number of seconds (30 by default) for it to shut down gracefully. If it doesn't shut down in time, the process is then killed through SIGKILL. To make sure your processes are always shut down gracefully, they need to handle the SIGTERM signal properly.

**TIP** You can also delete more than one pod by specifying multiple, space-separated names (for example, `kubectl delete po pod1 pod2`).

#### 3.8.2 *Deleting pods using label selectors*

Instead of specifying each pod to delete by name, you'll now use what you've learned about label selectors to stop both the kuba-manual and the kuba-manual-v2 pod. Both pods include the `creation_method=manual` label, so you can delete them by using a label selector:

```
$ kubectl delete po -l creation_method=manual
pod "kuba-manual" deleted
pod "kuba-manual-v2" deleted
```

In the earlier microservices example, where you had tens (or possibly hundreds) of pods, you could, for instance, delete all canary pods at once by specifying the `rel=canary` label selector (visualized in figure 3.10):

```
$ kubectl delete po -l rel=canary
```

#### 3.8.3 *Deleting pods by deleting the whole namespace*

Okay, back to your real pods. What about the pod in the custom-namespace? You no longer need either the pods in that namespace, or the namespace itself. You can



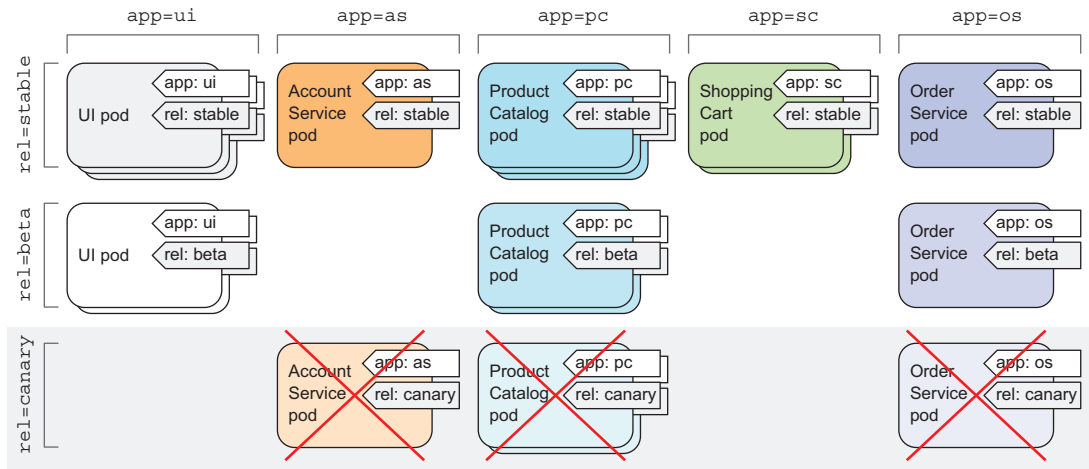


Figure 3.10 Selecting and deleting all canary pods through the `rel=canary` label selector

delete the whole namespace (the pods will be deleted along with the namespace automatically), using the following command:

```
$ kubectl delete ns custom-namespace
namespace "custom-namespace" deleted
```

### 3.8.4 Deleting all pods in a namespace, while keeping the namespace

You've now cleaned up almost everything. But what about the pod you created with the `kubectl run` command in chapter 2? That one is still running:

```
$ kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
kubia-zxzij   1/1     Running   0           1d
```

This time, instead of deleting the specific pod, tell Kubernetes to delete all pods in the current namespace by using the `--all` option:

```
$ kubectl delete po --all
pod "kubia-zxzij" deleted
```

Now, double check that no pods were left running:

```
$ kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
kubia-09as0   1/1     Running   0           1d
kubia-zxzij   1/1     Terminating   0           1d
```

Wait, what!?! The `kubia-zxzij` pod is terminating, but a new pod called `kubia-09as0`, which wasn't there before, has appeared. No matter how many times you delete all pods, a new pod called *kubia-something* will emerge.

You may remember you created your first pod with the `kubectl run` command. In chapter 2, I mentioned that this doesn't create a pod directly, but instead creates a `ReplicationController`, which then creates the pod. As soon as you delete a pod created by the `ReplicationController`, it immediately creates a new one. To delete the pod, you also need to delete the `ReplicationController`.

### 3.8.5 *Deleting (almost) all resources in a namespace*

You can delete the `ReplicationController` and the pods, as well as all the `Services` you've created, by deleting all resources in the current namespace with a single command:

```
$ kubectl delete all --all
pod "kubia-09as0" deleted
replicationcontroller "kubia" deleted
service "kubernetes" deleted
service "kubia-http" deleted
```

The first `all` in the command specifies that you're deleting resources of all types, and the `--all` option specifies that you're deleting all resource instances instead of specifying them by name (you already used this option when you ran the previous `delete` command).

**NOTE** Deleting everything with the `all` keyword doesn't delete absolutely everything. Certain resources (like `Secrets`, which we'll introduce in chapter 7) are preserved and need to be deleted explicitly.

As it deletes resources, `kubectl` will print the name of every resource it deletes. In the list, you should see the `kubia` `ReplicationController` and the `kubia-http` `Service` you created in chapter 2.

**NOTE** The `kubectl delete all --all` command also deletes the `kubernetes` `Service`, but it should be recreated automatically in a few moments.

## 3.9 *Summary*

After reading this chapter, you should now have a decent knowledge of the central building block in Kubernetes. Every other concept you'll learn about in the next few chapters is directly related to pods.

In this chapter, you've learned

- How to decide whether certain containers should be grouped together in a pod or not.

- Pods can run multiple processes and are similar to physical hosts in the non-container world.
- YAML or JSON descriptors can be written and used to create pods and then examined to see the specification of a pod and its current state.
- Labels and label selectors should be used to organize pods and easily perform operations on multiple pods at once.
- You can use node labels and selectors to schedule pods only to nodes that have certain features.
- Annotations allow attaching larger blobs of data to pods either by people or tools and libraries.
- Namespaces can be used to allow different teams to use the same cluster as though they were using separate Kubernetes clusters.
- How to use the `kubectl explain` command to quickly look up the information on any Kubernetes resource.

In the next chapter, you'll learn about ReplicationControllers and other resources that manage pods.

# *Replication and other controllers: deploying managed pods*

---

## **This chapter covers**

- Keeping pods healthy
- Running multiple instances of the same pod
- Automatically rescheduling pods after a node fails
- Scaling pods horizontally
- Running system-level pods on each cluster node
- Running batch jobs
- Scheduling jobs to run periodically or once in the future

As you’ve learned so far, pods represent the basic deployable unit in Kubernetes. You know how to create, supervise, and manage them manually. But in real-world use cases, you want your deployments to stay up and running automatically and remain healthy without any manual intervention. To do this, you almost never create pods directly. Instead, you create other types of resources, such as Replication-Controllers or Deployments, which then create and manage the actual pods.

When you create unmanaged pods (such as the ones you created in the previous chapter), a cluster node is selected to run the pod and then its containers are run on that node. In this chapter, you’ll learn that Kubernetes then monitors

those containers and automatically restarts them if they fail. But if the whole node fails, the pods on the node are lost and will not be replaced with new ones, unless those pods are managed by the previously mentioned ReplicationControllers or similar. In this chapter, you'll learn how Kubernetes checks if a container is still alive and restarts it if it isn't. You'll also learn how to run managed pods—both those that run indefinitely and those that perform a single task and then stop.

## 4.1 Keeping pods healthy

One of the main benefits of using Kubernetes is the ability to give it a list of containers and let it keep those containers running somewhere in the cluster. You do this by creating a Pod resource and letting Kubernetes pick a worker node for it and run the pod's containers on that node. But what if one of those containers dies? What if all containers of a pod die?

As soon as a pod is scheduled to a node, the Kubelet on that node will run its containers and, from then on, keep them running as long as the pod exists. If the container's main process crashes, the Kubelet will restart the container. If your application has a bug that causes it to crash every once in a while, Kubernetes will restart it automatically, so even without doing anything special in the app itself, running the app in Kubernetes automatically gives it the ability to heal itself.

But sometimes apps stop working without their process crashing. For example, a Java app with a memory leak will start throwing `OutOfMemoryErrors`, but the JVM process will keep running. It would be great to have a way for an app to signal to Kubernetes that it's no longer functioning properly and have Kubernetes restart it.

We've said that a container that crashes is restarted automatically, so maybe you're thinking you could catch these types of errors in the app and exit the process when they occur. You can certainly do that, but it still doesn't solve all your problems.

For example, what about those situations when your app stops responding because it falls into an infinite loop or a deadlock? To make sure applications are restarted in such cases, you must check an application's health from the outside and not depend on the app doing it internally.

### 4.1.1 Introducing liveness probes

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

**NOTE** Kubernetes also supports *readiness probes*, which we'll learn about in the next chapter. Be sure not to confuse the two. They're used for two different things.

Kubernetes can probe a container using one of the three mechanisms:

- An *HTTP GET* probe performs an HTTP GET request on the container's IP address, a port and path you specify. If the probe receives a response, and the

response code doesn't represent an error (in other words, if the HTTP response code is 2xx or 3xx), the probe is considered successful. If the server returns an error response code or if it doesn't respond at all, the probe is considered a failure and the container will be restarted as a result.

- A *TCP Socket* probe tries to open a TCP connection to the specified port of the container. If the connection is established successfully, the probe is successful. Otherwise, the container is restarted.
- An *Exec* probe executes an arbitrary command inside the container and checks the command's exit status code. If the status code is 0, the probe is successful. All other codes are considered failures.

#### 4.1.2 Creating an HTTP-based liveness probe

Let's see how to add a liveness probe to your Node.js app. Because it's a web app, it makes sense to add a liveness probe that will check whether its web server is serving requests. But because this particular Node.js app is too simple to ever fail, you'll need to make the app fail artificially.

To properly demo liveness probes, you'll modify the app slightly and make it return a 500 Internal Server Error HTTP status code for each request after the fifth one—your app will handle the first five client requests properly and then return an error on every subsequent request. Thanks to the liveness probe, it should be restarted when that happens, allowing it to properly handle client requests again.

You can find the code of the new app in the book's code archive (in the folder Chapter04/kubia-unhealthy). I've pushed the container image to Docker Hub, so you don't need to build it yourself.

You'll create a new pod that includes an HTTP GET liveness probe. The following listing shows the YAML for the pod.

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

```
apiVersion: v1
kind: pod
metadata:
  name: kubia-liveness
spec:
  containers:
    - image: luksa/kubia-unhealthy
      name: kubia
      livenessProbe:
        httpGet:
          path: /
          port: 8080
```

This is the image containing the (somewhat) broken app.



A liveness probe that will perform an HTTP GET



The path to request in the HTTP request

The network port the probe should connect to



The pod descriptor defines an `httpGet` liveness probe, which tells Kubernetes to periodically perform HTTP GET requests on path `/` on port 8080 to determine if the container is still healthy. These requests start as soon as the container is run.

After five such requests (or actual client requests), your app starts returning HTTP status code 500, which Kubernetes will treat as a probe failure, and will thus restart the container.

### 4.1.3 Seeing a liveness probe in action

To see what the liveness probe does, try creating the pod now. After about a minute and a half, the container will be restarted. You can see that by running `kubectl get`:

```
$ kubectl get po kubia-liveness
NAME          READY    STATUS    RESTARTS   AGE
kubia-liveness 1/1      Running   1          2m
```

The `RESTARTS` column shows that the pod's container has been restarted once (if you wait another minute and a half, it gets restarted again, and then the cycle continues indefinitely).

#### Obtaining the application log of a crashed container

In the previous chapter, you learned how to print the application's log with `kubectl logs`. If your container is restarted, the `kubectl logs` command will show the log of the current container.

When you want to figure out why the previous container terminated, you'll want to see those logs instead of the current container's logs. This can be done by using the `--previous` option:

```
$ kubectl logs mypod --previous
```

You can see why the container had to be restarted by looking at what `kubectl describe` prints out, as shown in the following listing.

#### Listing 4.2 A pod's description after its container is restarted

```
$ kubectl describe po kubia-liveness
Name:          kubia-liveness
...
Containers:
  kubia:
    Container ID:  docker://480986f8
    Image:         luksa/kubia-unhealthy
    Image ID:      docker://sha256:2b208508
    Port:
    State:         Running
      Started:     Sun, 14 May 2017 11:41:40 +0200
```

**The container is currently running.**

```

Last State:      Terminated
Reason:         Error
Exit Code:      137
Started:        Mon, 01 Jan 0001 00:00:00 +0000
Finished:       Sun, 14 May 2017 11:41:38 +0200
Ready:         True
Restart Count:  1
Liveness:       http-get http://:8080/ delay=0s timeout=1s
                period=10s #success=1 #failure=3
...
Events:
... Killing container with id docker://95246981:pod "kubia-liveness ..."
    container "kubia" is unhealthy, it will be killed and re-created.

```

The previous container terminated with an error and exited with code 137.

The container has been restarted once.

You can see that the container is currently running, but it previously terminated because of an error. The exit code was 137, which has a special meaning—it denotes that the process was terminated by an external signal. The number 137 is a sum of two numbers:  $128+x$ , where  $x$  is the signal number sent to the process that caused it to terminate. In the example,  $x$  equals 9, which is the number of the `SIGKILL` signal, meaning the process was killed forcibly.

The events listed at the bottom show why the container was killed—Kubernetes detected the container was unhealthy, so it killed and re-created it.

**NOTE** When a container is killed, a completely new container is created—it's not the same container being restarted again.

#### 4.1.4 *Configuring additional properties of the liveness probe*

You may have noticed that `kubectl describe` also displays additional information about the liveness probe:

```

Liveness: http-get http://:8080/ delay=0s timeout=1s period=10s #success=1
          #failure=3

```

Beside the liveness probe options you specified explicitly, you can also see additional properties, such as `delay`, `timeout`, `period`, and so on. The `delay=0s` part shows that the probing begins immediately after the container is started. The `timeout` is set to only 1 second, so the container must return a response in 1 second or the probe is counted as failed. The container is probed every 10 seconds (`period=10s`) and the container is restarted after the probe fails three consecutive times (`#failure=3`).

These additional parameters can be customized when defining the probe. For example, to set the initial delay, add the `initialDelaySeconds` property to the liveness probe as shown in the following listing.

#### **Listing 4.3** A liveness probe with an initial delay: `kubia-liveness-probe-initial-delay.yaml`

```

livenessProbe:
  httpGet:
    path: /

```



```
port: 8080
initialDelaySeconds: 15
```

← | **Kubernetes will wait 15 seconds  
before executing the first probe.**

If you don't set the initial delay, the prober will start probing the container as soon as it starts, which usually leads to the probe failing, because the app isn't ready to start receiving requests. If the number of failures exceeds the failure threshold, the container is restarted before it's even able to start responding to requests properly.

**TIP** Always remember to set an initial delay to account for your app's startup time.

I've seen this on many occasions and users were confused why their container was being restarted. But if they'd used `kubectl describe`, they'd have seen that the container terminated with exit code 137 or 143, telling them that the pod was terminated externally. Additionally, the listing of the pod's events would show that the container was killed because of a failed liveness probe. If you see this happening at pod startup, it's because you failed to set `initialDelaySeconds` appropriately.

**NOTE** Exit code 137 signals that the process was killed by an external signal (exit code is  $128 + 9$  (SIGKILL)). Likewise, exit code 143 corresponds to  $128 + 15$  (SIGTERM).

#### 4.1.5 Creating effective liveness probes

For pods running in production, you should always define a liveness probe. Without one, Kubernetes has no way of knowing whether your app is still alive or not. As long as the process is still running, Kubernetes will consider the container to be healthy.

##### WHAT A LIVENESS PROBE SHOULD CHECK

Your simplistic liveness probe simply checks if the server is responding. While this may seem overly simple, even a liveness probe like this does wonders, because it causes the container to be restarted if the web server running within the container stops responding to HTTP requests. Compared to having no liveness probe, this is a major improvement, and may be sufficient in most cases.

But for a better liveness check, you'd configure the probe to perform requests on a specific URL path (`/health`, for example) and have the app perform an internal status check of all the vital components running inside the app to ensure none of them has died or is unresponsive.

**TIP** Make sure the `/health` HTTP endpoint doesn't require authentication; otherwise the probe will always fail, causing your container to be restarted indefinitely.

Be sure to check only the internals of the app and nothing influenced by an external factor. For example, a frontend web server's liveness probe shouldn't return a failure when the server can't connect to the backend database. If the underlying cause is in the database itself, restarting the web server container will not fix the problem.

Because the liveness probe will fail again, you'll end up with the container restarting repeatedly until the database becomes accessible again.

#### **KEEPING PROBES LIGHT**

Liveness probes shouldn't use too many computational resources and shouldn't take too long to complete. By default, the probes are executed relatively often and are only allowed one second to complete. Having a probe that does heavy lifting can slow down your container considerably. Later in the book, you'll also learn about how to limit CPU time available to a container. The probe's CPU time is counted in the container's CPU time quota, so having a heavyweight liveness probe will reduce the CPU time available to the main application processes.

**TIP** If you're running a Java app in your container, be sure to use an HTTP GET liveness probe instead of an Exec probe, where you spin up a whole new JVM to get the liveness information. The same goes for any JVM-based or similar applications, whose start-up procedure requires considerable computational resources.

#### **DON'T BOTHER IMPLEMENTING RETRY LOOPS IN YOUR PROBES**

You've already seen that the failure threshold for the probe is configurable and usually the probe must fail multiple times before the container is killed. But even if you set the failure threshold to 1, Kubernetes will retry the probe several times before considering it a single failed attempt. Therefore, implementing your own retry loop into the probe is wasted effort.

#### **LIVENESS PROBE WRAP-UP**

You now understand that Kubernetes keeps your containers running by restarting them if they crash or if their liveness probes fail. This job is performed by the Kubelet on the node hosting the pod—the Kubernetes Control Plane components running on the master(s) have no part in this process.

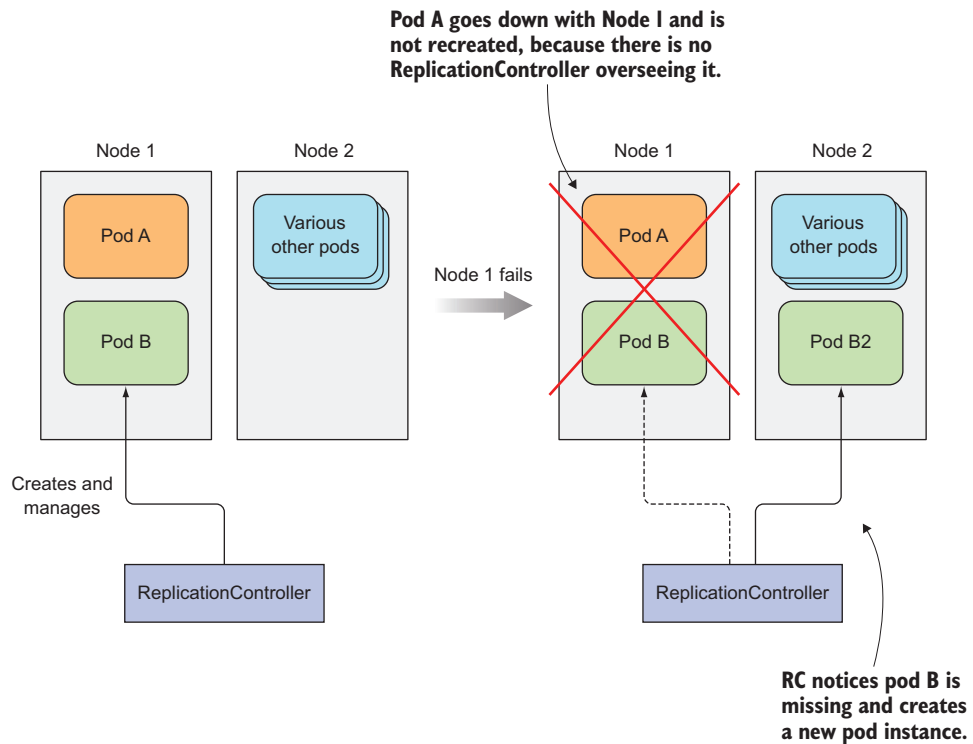
But if the node itself crashes, it's the Control Plane that must create replacements for all the pods that went down with the node. It doesn't do that for pods that you create directly. Those pods aren't managed by anything except by the Kubelet, but because the Kubelet runs on the node itself, it can't do anything if the node fails.

To make sure your app is restarted on another node, you need to have the pod managed by a ReplicationController or similar mechanism, which we'll discuss in the rest of this chapter.

## **4.2 Introducing ReplicationControllers**

A ReplicationController is a Kubernetes resource that ensures its pods are always kept running. If the pod disappears for any reason, such as in the event of a node disappearing from the cluster or because the pod was evicted from the node, the ReplicationController notices the missing pod and creates a replacement pod.

Figure 4.1 shows what happens when a node goes down and takes two pods with it. Pod A was created directly and is therefore an unmanaged pod, while pod B is managed



**Figure 4.1** When a node fails, only pods backed by a ReplicationController are recreated.

by a ReplicationController. After the node fails, the ReplicationController creates a new pod (pod B2) to replace the missing pod B, whereas pod A is lost completely—nothing will ever recreate it.

The ReplicationController in the figure manages only a single pod, but ReplicationControllers, in general, are meant to create and manage multiple copies (replicas) of a pod. That’s where ReplicationControllers got their name from.

#### 4.2.1 The operation of a ReplicationController

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

You might be wondering how there can be more than the desired number of replicas. This can happen for a few reasons:

- Someone creates a pod of the same type manually.
- Someone changes an existing pod’s “type.”
- Someone decreases the desired number of pods, and so on.

I’ve used the term pod “type” a few times. But no such thing exists. Replication-Controllers don’t operate on pod types, but on sets of pods that match a certain label selector (you learned about them in the previous chapter).

#### INTRODUCING THE CONTROLLER’S RECONCILIATION LOOP

A ReplicationController’s job is to make sure that an exact number of pods always matches its label selector. If it doesn’t, the ReplicationController takes the appropriate action to reconcile the actual with the desired number. The operation of a Replication-Controller is shown in figure 4.2.

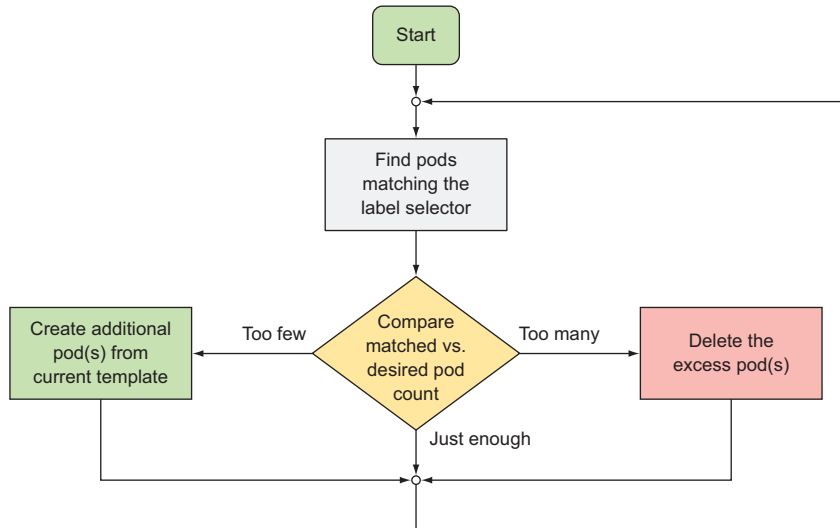


Figure 4.2 A ReplicationController’s reconciliation loop

#### UNDERSTANDING THE THREE PARTS OF A REPLICATIONCONTROLLER

A ReplicationController has three essential parts (also shown in figure 4.3):

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

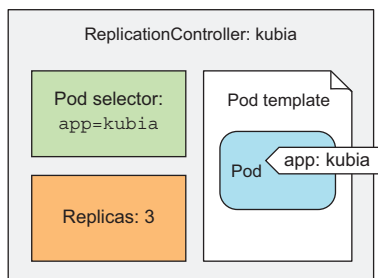


Figure 4.3 The three key parts of a ReplicationController (pod selector, replica count, and pod template)

A ReplicationController’s replica count, the label selector, and even the pod template can all be modified at any time, but only changes to the replica count affect existing pods.

#### UNDERSTANDING THE EFFECT OF CHANGING THE CONTROLLER’S LABEL SELECTOR OR POD TEMPLATE

Changes to the label selector and the pod template have no effect on existing pods. Changing the label selector makes the existing pods fall out of the scope of the ReplicationController, so the controller stops caring about them. ReplicationControllers also don’t care about the actual “contents” of its pods (the container images, environment variables, and other things) after they create the pod. The template therefore only affects new pods created by this ReplicationController. You can think of it as a cookie cutter for cutting out new pods.

#### UNDERSTANDING THE BENEFITS OF USING A REPLICATIONCONTROLLER

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

- It makes sure a pod (or multiple pod replicas) is always running by starting a new pod when an existing one goes missing.
- When a cluster node fails, it creates replacement replicas for all the pods that were running on the failed node (those that were under the ReplicationController’s control).
- It enables easy horizontal scaling of pods—both manual and automatic (see horizontal pod auto-scaling in chapter 15).

**NOTE** A pod instance is never relocated to another node. Instead, the ReplicationController creates a completely new pod instance that has no relation to the instance it’s replacing.

### 4.2.2 Creating a ReplicationController

Let’s look at how to create a ReplicationController and then see how it keeps your pods running. Like pods and other Kubernetes resources, you create a ReplicationController by posting a JSON or YAML descriptor to the Kubernetes API server.

You’re going to create a YAML file called `kubia-rc.yaml` for your ReplicationController, as shown in the following listing.

**Listing 4.4** A YAML definition of a ReplicationController: `kubia-rc.yaml`

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: kubia
spec:
  replicas: 3
  selector:
    app: kubia
```

This manifest defines a ReplicationController (RC)

The name of this ReplicationController

The desired number of pod instances

The pod selector determining what pods the RC is operating on

```

template:
  metadata:
    labels:
      app: kubern
  spec:
    containers:
      - name: kubern
        image: luksa/kubern
        ports:
          - containerPort: 8080

```

**The pod template  
for creating new  
pods**

When you post the file to the API server, Kubernetes creates a new ReplicationController named `kubern`, which makes sure three pod instances always match the label selector `app=kubern`. When there aren't enough pods, new pods will be created from the provided pod template. The contents of the template are almost identical to the pod definition you created in the previous chapter.

The pod labels in the template must obviously match the label selector of the ReplicationController; otherwise the controller would create new pods indefinitely, because spinning up a new pod wouldn't bring the actual replica count any closer to the desired number of replicas. To prevent such scenarios, the API server verifies the ReplicationController definition and will not accept it if it's misconfigured.

Not specifying the selector at all is also an option. In that case, it will be configured automatically from the labels in the pod template.

**TIP** Don't specify a pod selector when defining a ReplicationController. Let Kubernetes extract it from the pod template. This will keep your YAML shorter and simpler.

To create the ReplicationController, use the `kubectl create` command, which you already know:

```

$ kubectl create -f kubia-rc.yaml
replicationcontroller "kubern" created

```

As soon as the ReplicationController is created, it goes to work. Let's see what it does.

### 4.2.3 *Seeing the ReplicationController in action*

Because no pods exist with the `app=kubern` label, the ReplicationController should spin up three new pods from the pod template. List the pods to see if the ReplicationController has done what it's supposed to:

```

$ kubectl get pods
NAME          READY   STATUS             RESTARTS   AGE
kubern-53thy  0/1     ContainerCreating  0           2s
kubern-k0xz6  0/1     ContainerCreating  0           2s
kubern-q3vkg  0/1     ContainerCreating  0           2s

```

Indeed, it has! You wanted three pods, and it created three pods. It's now managing those three pods. Next you'll mess with them a little to see how the ReplicationController responds.

#### SEEING THE REPLICATIONCONTROLLER RESPOND TO A DELETED POD

First, you'll delete one of the pods manually to see how the ReplicationController spins up a new one immediately, bringing the number of matching pods back to three:

```
$ kubectl delete pod kuba-53thy
pod "kuba-53thy" deleted
```

Listing the pods again shows four of them, because the one you deleted is terminating, and a new pod has already been created:

```
$ kubectl get pods
NAME          READY   STATUS             RESTARTS   AGE
kuba-53thy    1/1     Terminating       0          3m
kuba-oini2    0/1     ContainerCreating   0          2s
kuba-k0xz6    1/1     Running             0          3m
kuba-q3vkg    1/1     Running             0          3m
```

The ReplicationController has done its job again. It's a nice little helper, isn't it?

#### GETTING INFORMATION ABOUT A REPLICATIONCONTROLLER

Now, let's see what information the `kubectl get` command shows for ReplicationControllers:

```
$ kubectl get rc
NAME      DESIRED   CURRENT   READY   AGE
kuba      3         3         2       3m
```

**NOTE** We're using `rc` as a shorthand for `replicationcontroller`.

You see three columns showing the desired number of pods, the actual number of pods, and how many of them are ready (you'll learn what that means in the next chapter, when we talk about readiness probes).

You can see additional information about your ReplicationController with the `kubectl describe` command, as shown in the following listing.

#### Listing 4.5 Displaying details of a ReplicationController with `kubectl describe`

```
$ kubectl describe rc kuba
Name:          kuba
Namespace:     default
Selector:      app=kuba
Labels:        app=kuba
Annotations:   <none>
Replicas:      3 current / 3 desired
Pods Status:   4 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:      app=kuba
  Containers:  ...
```

The actual vs. the desired number of pod instances

Number of pod instances per pod status

Volumes:	<none>			<div>← The events related to this ReplicationController</div>	
Events:					
From	Type	Reason	Message		
----	-----	-----	-----		
replication-controller	Normal	SuccessfulCreate	Created pod: kubia-53thy		
replication-controller	Normal	SuccessfulCreate	Created pod: kubia-k0xz6		
replication-controller	Normal	SuccessfulCreate	Created pod: kubia-q3vkg		
replication-controller	Normal	SuccessfulCreate	Created pod: kubia-oini2		

The current number of replicas matches the desired number, because the controller has already created a new pod. It shows four running pods because a pod that's terminating is still considered running, although it isn't counted in the current replica count.

The list of events at the bottom shows the actions taken by the ReplicationController—it has created four pods so far.

#### UNDERSTANDING EXACTLY WHAT CAUSED THE CONTROLLER TO CREATE A NEW POD

The controller is responding to the deletion of a pod by creating a new replacement pod (see figure 4.4). Well, technically, it isn't responding to the deletion itself, but the resulting state—the inadequate number of pods.

While a ReplicationController is immediately notified about a pod being deleted (the API server allows clients to watch for changes to resources and resource lists), that's not what causes it to create a replacement pod. The notification triggers the controller to check the actual number of pods and take appropriate action.

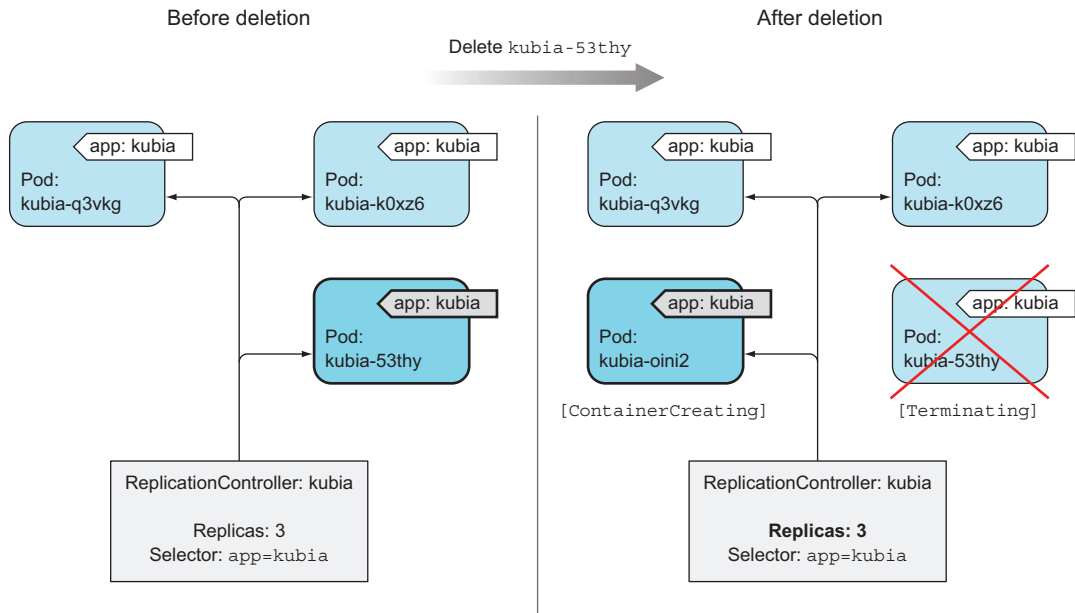


Figure 4.4 If a pod disappears, the ReplicationController sees too few pods and creates a new replacement pod.



**RESPONDING TO A NODE FAILURE**

Seeing the ReplicationController respond to the manual deletion of a pod isn't too interesting, so let's look at a better example. If you're using Google Kubernetes Engine to run these examples, you have a three-node Kubernetes cluster. You're going to disconnect one of the nodes from the network to simulate a node failure.

**NOTE** If you're using Minikube, you can't do this exercise, because you only have one node that acts both as a master and a worker node.

If a node fails in the non-Kubernetes world, the ops team would need to migrate the applications running on that node to other machines manually. Kubernetes, on the other hand, does that automatically. Soon after the ReplicationController detects that its pods are down, it will spin up new pods to replace them.

Let's see this in action. You need to ssh into one of the nodes with the `gcloud compute ssh` command and then shut down its network interface with `sudo ifconfig eth0 down`, as shown in the following listing.

**NOTE** Choose a node that runs at least one of your pods by listing pods with the `-o wide` option.

**Listing 4.6 Simulating a node failure by shutting down its network interface**

```
$ gcloud compute ssh gke-kubia-default-pool-b46381f1-zwko
Enter passphrase for key '/home/luksa/.ssh/google_compute_engine':

Welcome to Kubernetes v1.6.4!
...

luksa@gke-kubia-default-pool-b46381f1-zwko ~ $ sudo ifconfig eth0 down
```

When you shut down the network interface, the ssh session will stop responding, so you need to open up another terminal or hard-exit from the ssh session. In the new terminal you can list the nodes to see if Kubernetes has detected that the node is down. This takes a minute or so. Then, the node's status is shown as `NotReady`:

```
$ kubectl get node
```

NAME	STATUS	AGE
gke-kubia-default-pool-b46381f1-opc5	Ready	5h
gke-kubia-default-pool-b46381f1-s8gj	Ready	5h
gke-kubia-default-pool-b46381f1-zwko	NotReady	5h

Node isn't ready, because it's disconnected from the network

If you list the pods now, you'll still see the same three pods as before, because Kubernetes waits a while before rescheduling pods (in case the node is unreachable because of a temporary network glitch or because the Kubelet is restarting). If the node stays unreachable for several minutes, the status of the pods that were scheduled to that node changes to `Unknown`. At that point, the ReplicationController will immediately spin up a new pod. You can see this by listing the pods again:

\$ kubectl get pods					
NAME	READY	STATUS	RESTARTS	AGE	
kubia-oln12	1/1	Running	0	10m	This pod's status is unknown, because its node is unreachable.
kubia-k0xxz6	1/1	Running	0	10m	
kubia-g3vkg	1/1	Unknown	0	10m	
kubia-dmdck	1/1	Running	0	5s	This pod was created five seconds ago.

Looking at the age of the pods, you see that the kubia-dmdck pod is new. You again have three pod instances running, which means the ReplicationController has again done its job of bringing the actual state of the system to the desired state.

The same thing happens if a node fails (either breaks down or becomes unreachable). No immediate human intervention is necessary. The system heals itself automatically.

To bring the node back, you need to reset it with the following command:

```
$ gcloud compute instances reset gke-kubia-default-pool-b46381f1-zwko
```

When the node boots up again, its status should return to Ready, and the pod whose status was Unknown will be deleted.

4.2.4 Moving pods in and out of the scope of a ReplicationController

Pods created by a ReplicationController aren't tied to the ReplicationController in any way. At any moment, a ReplicationController manages pods that match its label selector. By changing a pod's labels, it can be removed from or added to the scope of a ReplicationController. It can even be moved from one ReplicationController to another.

**TIP** Although a pod isn't tied to a ReplicationController, the pod does reference it in the metadata.ownerReferences field, which you can use to easily find which ReplicationController a pod belongs to.

If you change a pod's labels so they no longer match a ReplicationController's label selector, the pod becomes like any other manually created pod. It's no longer managed by anything. If the node running the pod fails, the pod is obviously not rescheduled. But keep in mind that when you changed the pod's labels, the replication controller noticed one pod was missing and spun up a new pod to replace it.

Let's try this with your pods. Because your ReplicationController manages pods that have the app=kubia label, you need to either remove this label or change its value to move the pod out of the ReplicationController's scope. Adding another label will have no effect, because the ReplicationController doesn't care if the pod has any additional labels. It only cares whether the pod has all the labels referenced in the label selector.

**ADDING LABELS TO PODS MANAGED BY A REPLICATIONCONTROLLER**

Let's confirm that a ReplicationController doesn't care if you add additional labels to its managed pods:

```
$ kubectl label pod kubia-dmdck type=special
pod "kubia-dmdck" labeled

$ kubectl get pods --show-labels
NAME          READY   STATUS    RESTARTS   AGE   LABELS
kubia-oini2    1/1     Running   0           11m   app=kubia
kubia-k0xz6    1/1     Running   0           11m   app=kubia
kubia-dmdck    1/1     Running   0           1m    app=kubia,type=special
```

You've added the `type=special` label to one of the pods. Listing all pods again shows the same three pods as before, because no change occurred as far as the ReplicationController is concerned.

**CHANGING THE LABELS OF A MANAGED POD**

Now, you'll change the `app=kubia` label to something else. This will make the pod no longer match the ReplicationController's label selector, leaving it to only match two pods. The ReplicationController should therefore start a new pod to bring the number back to three:

```
$ kubectl label pod kubia-dmdck app=foo --overwrite
pod "kubia-dmdck" labeled
```

The `--overwrite` argument is necessary; otherwise `kubectl` will only print out a warning and won't change the label, to prevent you from inadvertently changing an existing label's value when your intent is to add a new one.

Listing all the pods again should now show four pods:

```
$ kubectl get pods -L app
NAME          READY   STATUS             RESTARTS   AGE   APP
kubia-2qneh    0/1     ContainerCreating   0          2s    kubia
kubia-oini2    1/1     Running             0          20m   kubia
kubia-k0xz6    1/1     Running             0          20m   kubia
kubia-dmdck    1/1     Running             0          10m   foo
```

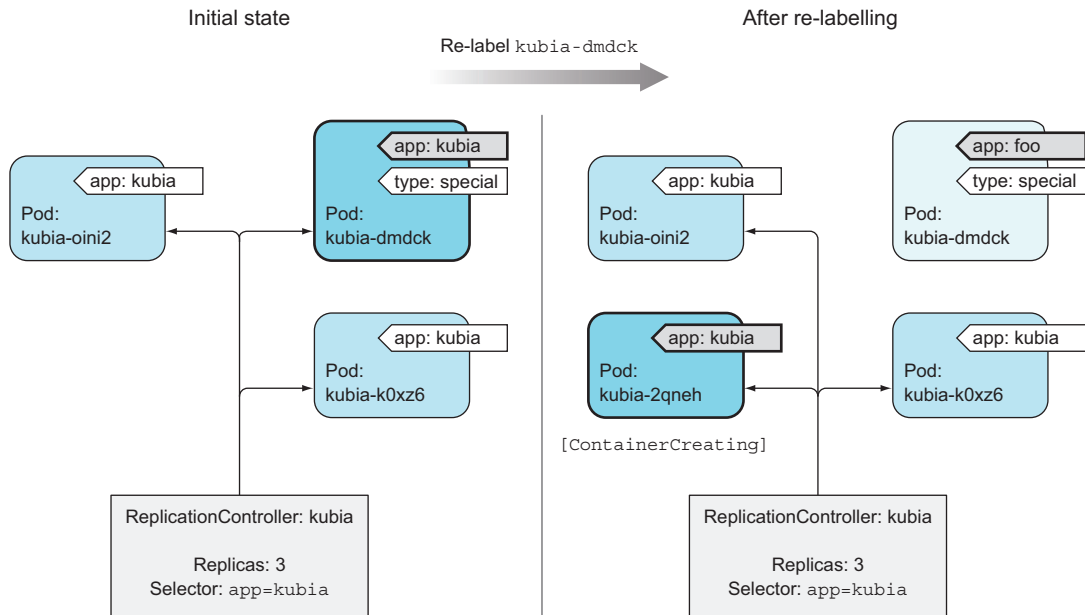
Newly created pod that replaces the pod you removed from the scope of the ReplicationController

Pod no longer managed by the ReplicationController

**NOTE** You're using the `-L app` option to display the app label in a column.

There, you now have four pods altogether: one that isn't managed by your ReplicationController and three that are. Among them is the newly created pod.

Figure 4.5 illustrates what happened when you changed the pod's labels so they no longer matched the ReplicationController's pod selector. You can see your three pods and your ReplicationController. After you change the pod's label from `app=kubia` to `app=foo`, the ReplicationController no longer cares about the pod. Because the controller's replica count is set to 3 and only two pods match the label selector, the



**Figure 4.5** Removing a pod from the scope of a ReplicationController by changing its labels

ReplicationController spins up pod `kubia-2qneh` to bring the number back up to three. Pod `kubia-dmdck` is now completely independent and will keep running until you delete it manually (you can do that now, because you don't need it anymore).

#### REMOVING PODS FROM CONTROLLERS IN PRACTICE

Removing a pod from the scope of the ReplicationController comes in handy when you want to perform actions on a specific pod. For example, you might have a bug that causes your pod to start behaving badly after a specific amount of time or a specific event. If you know a pod is malfunctioning, you can take it out of the ReplicationController's scope, let the controller replace it with a new one, and then debug or play with the pod in any way you want. Once you're done, you delete the pod.

#### CHANGING THE REPLICATIONCONTROLLER'S LABEL SELECTOR

As an exercise to see if you fully understand ReplicationControllers, what do you think would happen if instead of changing the labels of a pod, you modified the ReplicationController's label selector?

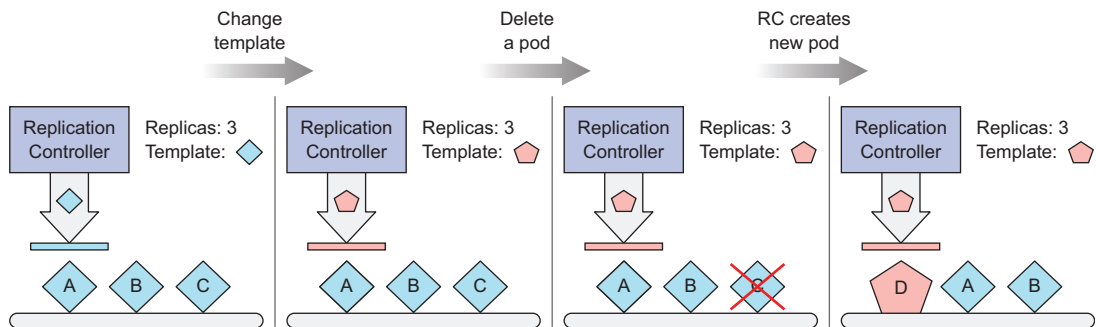
If your answer is that it would make all the pods fall out of the scope of the ReplicationController, which would result in it creating three new pods, you're absolutely right. And it shows that you understand how ReplicationControllers work.

Kubernetes does allow you to change a ReplicationController's label selector, but that's not the case for the other resources that are covered in the second half of this

chapter and which are also used for managing pods. You'll never change a controller's label selector, but you'll regularly change its pod template. Let's take a look at that.

## 4.2.5 Changing the pod template

A ReplicationController's pod template can be modified at any time. Changing the pod template is like replacing a cookie cutter with another one. It will only affect the cookies you cut out afterward and will have no effect on the ones you've already cut (see figure 4.6). To modify the old pods, you'd need to delete them and let the ReplicationController replace them with new ones based on the new template.



**Figure 4.6** Changing a ReplicationController's pod template only affects pods created afterward and has no effect on existing pods.

As an exercise, you can try editing the ReplicationController and adding a label to the pod template. You can edit the ReplicationController with the following command:

```
$ kubectl edit rc kubia
```

This will open the ReplicationController's YAML definition in your default text editor. Find the pod template section and add an additional label to the metadata. After you save your changes and exit the editor, `kubectl` will update the ReplicationController and print the following message:

```
replicationcontroller "kubia" edited
```

You can now list pods and their labels again and confirm that they haven't changed. But if you delete the pods and wait for their replacements to be created, you'll see the new label.

Editing a ReplicationController like this to change the container image in the pod template, deleting the existing pods, and letting them be replaced with new ones from the new template could be used for upgrading pods, but you'll learn a better way of doing that in chapter 9.

**Configuring kubectl edit to use a different text editor**

You can tell kubectl to use a text editor of your choice by setting the `KUBE_EDITOR` environment variable. For example, if you'd like to use `nano` for editing Kubernetes resources, execute the following command (or put it into your `~/.bashrc` or an equivalent file):

```
export KUBE_EDITOR="/usr/bin/nano"
```

If the `KUBE_EDITOR` environment variable isn't set, kubectl edit falls back to using the default editor, usually configured through the `EDITOR` environment variable.

**4.2.6 Horizontally scaling pods**

You've seen how ReplicationControllers make sure a specific number of pod instances is always running. Because it's incredibly simple to change the desired number of replicas, this also means scaling pods horizontally is trivial.

Scaling the number of pods up or down is as easy as changing the value of the `replicas` field in the ReplicationController resource. After the change, the ReplicationController will either see too many pods exist (when scaling down) and delete part of them, or see too few of them (when scaling up) and create additional pods.

**SCALING UP A REPLICATIONCONTROLLER**

Your ReplicationController has been keeping three instances of your pod running. You're going to scale that number up to 10 now. As you may remember, you've already scaled a ReplicationController in chapter 2. You could use the same command as before:

```
$ kubectl scale rc kubia --replicas=10
```

But you'll do it differently this time.

**SCALING A REPLICATIONCONTROLLER BY EDITING ITS DEFINITION**

Instead of using the `kubectl scale` command, you're going to scale it in a declarative way by editing the ReplicationController's definition:

```
$ kubectl edit rc kubia
```

When the text editor opens, find the `spec.replicas` field and change its value to 10, as shown in the following listing.

**Listing 4.7 Editing the RC in a text editor by running kubectl edit**

```
# Please edit the object below. Lines beginning with a '#' will be ignored,
# and an empty file will abort the edit. If an error occurs while saving
# this file will be reopened with the relevant failures.
apiVersion: v1
kind: ReplicationController
```

```

metadata:
  ...
spec:
  replicas: 3
  selector:
    app: kubia
  ...

```

← | Change the number 3 to number 10 in this line.

When you save the file and close the editor, the ReplicationController is updated and it immediately scales the number of pods to 10:

```
$ kubectl get rc
```

NAME	DESIRED	CURRENT	READY	AGE
kubia	10	10	4	21m

There you go. If the `kubectl scale` command makes it look as though you're telling Kubernetes exactly what to do, it's now much clearer that you're making a declarative change to the desired state of the ReplicationController and not telling Kubernetes to do something.

#### SCALING DOWN WITH THE KUBECTL SCALE COMMAND

Now scale back down to 3. You can use the `kubectl scale` command:

```
$ kubectl scale rc kubia --replicas=3
```

All this command does is modify the `spec.replicas` field of the ReplicationController's definition—like when you changed it through `kubectl edit`.

#### UNDERSTANDING THE DECLARATIVE APPROACH TO SCALING

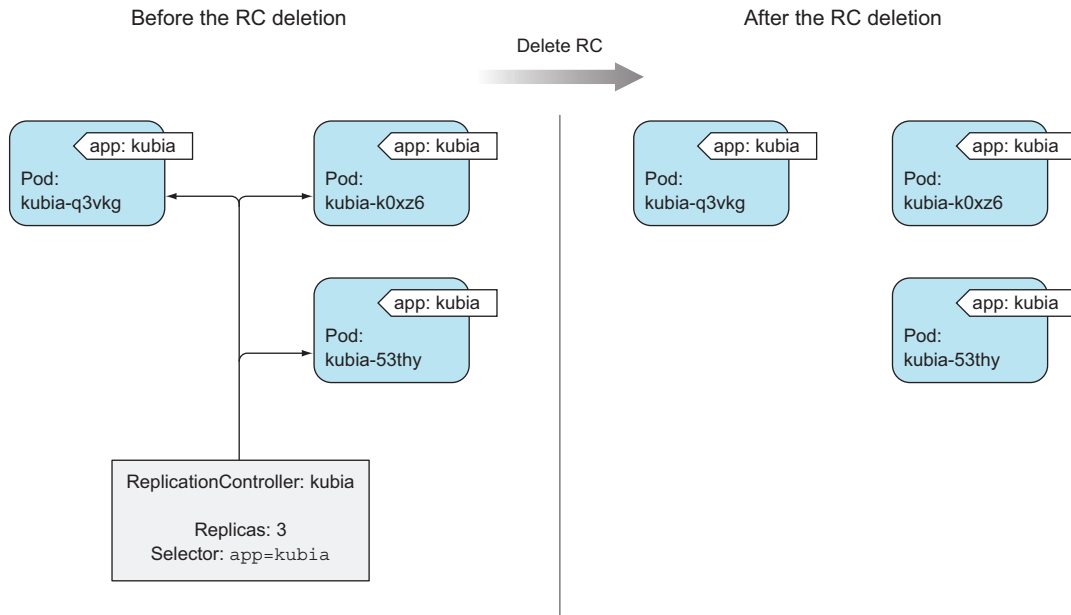
Horizontally scaling pods in Kubernetes is a matter of stating your desire: “I want to have  $x$  number of instances running.” You're not telling Kubernetes what or how to do it. You're just specifying the desired state.

This declarative approach makes interacting with a Kubernetes cluster easy. Imagine if you had to manually determine the current number of running instances and then explicitly tell Kubernetes how many additional instances to run. That's more work and is much more error-prone. Changing a simple number is much easier, and in chapter 15, you'll learn that even that can be done by Kubernetes itself if you enable horizontal pod auto-scaling.

### 4.2.7 Deleting a ReplicationController

When you delete a ReplicationController through `kubectl delete`, the pods are also deleted. But because pods created by a ReplicationController aren't an integral part of the ReplicationController, and are only managed by it, you can delete only the ReplicationController and leave the pods running, as shown in figure 4.7.

This may be useful when you initially have a set of pods managed by a ReplicationController, and then decide to replace the ReplicationController with a ReplicaSet, for example (you'll learn about them next.). You can do this without affecting the



**Figure 4.7** Deleting a replication controller with `--cascade=false` leaves pods unmanaged.

pods and keep them running without interruption while you replace the ReplicationController that manages them.

When deleting a ReplicationController with `kubectl delete`, you can keep its pods running by passing the `--cascade=false` option to the command. Try that now:

```
$ kubectl delete rc kubia --cascade=false
replicationcontroller "kubia" deleted
```

You've deleted the ReplicationController so the pods are on their own. They are no longer managed. But you can always create a new ReplicationController with the proper label selector and make them managed again.

### 4.3 *Using ReplicaSets instead of ReplicationControllers*

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

You could have started this chapter by creating a ReplicaSet instead of a ReplicationController, but I felt it would be a good idea to start with what was initially available in Kubernetes. Plus, you'll still see ReplicationControllers used in the wild, so it's good for you to know about them. That said, you should always create ReplicaSets instead of ReplicationControllers from now on. They're almost identical, so you shouldn't have any trouble using them instead.



You usually won't create them directly, but instead have them created automatically when you create the higher-level Deployment resource, which you'll learn about in chapter 9. In any case, you should understand ReplicaSets, so let's see how they differ from ReplicationControllers.

### 4.3.1 Comparing a ReplicaSet to a ReplicationController

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

Also, for example, a single ReplicationController can't match pods with the label `env=production` and those with the label `env=devel` at the same time. It can only match either pods with the `env=production` label or pods with the `env=devel` label. But a single ReplicaSet can match both sets of pods and treat them as a single group.

Similarly, a ReplicationController can't match pods based merely on the presence of a label key, regardless of its value, whereas a ReplicaSet can. For example, a ReplicaSet can match all pods that include a label with the key `env`, whatever its actual value is (you can think of it as `env=*`).

### 4.3.2 Defining a ReplicaSet

You're going to create a ReplicaSet now to see how the orphaned pods that were created by your ReplicationController and then abandoned earlier can now be adopted by a ReplicaSet. First, you'll rewrite your ReplicationController into a ReplicaSet by creating a new file called `kubia-replicaset.yaml` with the contents in the following listing.

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

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

ReplicaSets aren't part of the v1 API, but belong to the apps API group and version v1beta2.

You're using the simpler `matchLabels` selector here, which is much like a ReplicationController's selector.

The template is the same as in the ReplicationController.

The first thing to note is that ReplicaSets aren't part of the v1 API, so you need to ensure you specify the proper `apiVersion` when creating the resource. You're creating a resource of type `ReplicaSet` which has much the same contents as the `ReplicationController` you created earlier.

The only difference is in the selector. Instead of listing labels the pods need to have directly under the `selector` property, you're specifying them under `selector.matchLabels`. This is the simpler (and less expressive) way of defining label selectors in a `ReplicaSet`. Later, you'll look at the more expressive option, as well.

### About the API version attribute

This is your first opportunity to see that the `apiVersion` property specifies two things:

- The API group (which is `apps` in this case)
- The actual API version (`v1beta2`)

You'll see throughout the book that certain Kubernetes resources are in what's called the core API group, which doesn't need to be specified in the `apiVersion` field (you just specify the version—for example, you've been using `apiVersion: v1` when defining `Pod` resources). Other resources, which were introduced in later Kubernetes versions, are categorized into several API groups. Look at the inside of the book's covers to see all resources and their respective API groups.

Because you still have three pods matching the `app=kubia` selector running from earlier, creating this `ReplicaSet` will not cause any new pods to be created. The `ReplicaSet` will take those existing three pods under its wing.

### 4.3.3 Creating and examining a ReplicaSet

Create the `ReplicaSet` from the YAML file with the `kubectl create` command. After that, you can examine the `ReplicaSet` with `kubectl get` and `kubectl describe`:

```
$ kubectl get rs
NAME          DESIRED   CURRENT   READY   AGE
kubia         3         3         3       3s
```

**TIP** Use `rs` shorthand, which stands for `replicaset`.

```
$ kubectl describe rs
Name:          kubia
Namespace:     default
Selector:      app=kubia
Labels:        app=kubia
Annotations:   <none>
Replicas:      3 current / 3 desired
Pods Status:   3 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:      app=kubia
```

```
Containers:  ...
Volumes:    <none>
Events:     <none>
```

As you can see, the ReplicaSet isn't any different from a ReplicationController. It's showing it has three replicas matching the selector. If you list all the pods, you'll see they're still the same three pods you had before. The ReplicaSet didn't create any new ones.

#### 4.3.4 Using the ReplicaSet's more expressive label selectors

The main improvements of ReplicaSets over ReplicationControllers are their more expressive label selectors. You intentionally used the simpler `matchLabels` selector in the first ReplicaSet example to see that ReplicaSets are no different from ReplicationControllers. Now, you'll rewrite the selector to use the more powerful `matchExpressions` property, as shown in the following listing.

**Listing 4.9** A `matchExpressions` selector: `kubia-replicaset-matchexpressions.yaml`

```
selector:
  matchExpressions:
    - key: app
      operator: In
      values:
        - kubia
```

This selector requires the pod to contain a label with the "app" key.

The label's value must be "kubia".

**NOTE** Only the selector is shown. You'll find the whole ReplicaSet definition in the book's code archive.

You can add additional expressions to the selector. As in the example, each expression must contain a key, an operator, and possibly (depending on the operator) a list of values. You'll see four valid operators:

- `In-Label`'s value must match one of the specified values.
- `NotIn-Label`'s value must not match any of the specified values.
- `Exists-Pod` must include a label with the specified key (the value isn't important). When using this operator, you shouldn't specify the values field.
- `DoesNotExist-Pod` must not include a label with the specified key. The values property must not be specified.

If you specify multiple expressions, all those expressions must evaluate to true for the selector to match a pod. If you specify both `matchLabels` and `matchExpressions`, all the labels must match and all the expressions must evaluate to true for the pod to match the selector.

### 4.3.5 Wrapping up ReplicaSets

This was a quick introduction to ReplicaSets as an alternative to ReplicationControllers. Remember, always use them instead of ReplicationControllers, but you may still find ReplicationControllers in other people's deployments.

Now, delete the ReplicaSet to clean up your cluster a little. You can delete the ReplicaSet the same way you'd delete a ReplicationController:

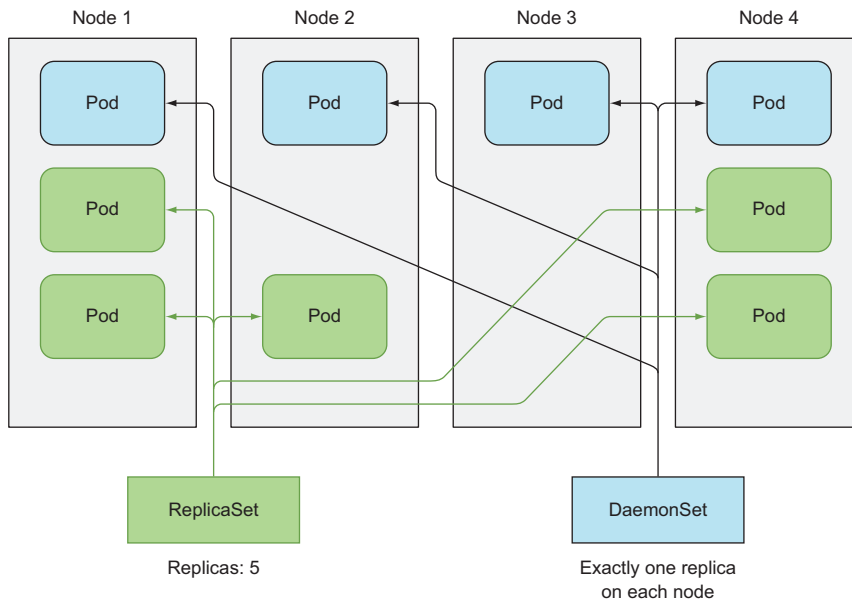
```
$ kubectl delete rs kubia
replicaset "kubia" deleted
```

Deleting the ReplicaSet should delete all the pods. List the pods to confirm that's the case.

## 4.4 Running exactly one pod on each node with DaemonSets

Both ReplicationControllers and ReplicaSets are used for running a specific number of pods deployed anywhere in the Kubernetes cluster. But certain cases exist when you want a pod to run on each and every node in the cluster (and each node needs to run exactly one instance of the pod, as shown in figure 4.8).

Those cases include infrastructure-related pods that perform system-level operations. For example, you'll want to run a log collector and a resource monitor on every node. Another good example is Kubernetes' own kube-proxy process, which needs to run on all nodes to make services work.



**Figure 4.8** DaemonSets run only a single pod replica on each node, whereas ReplicaSets scatter them around the whole cluster randomly.

Outside of Kubernetes, such processes would usually be started through system init scripts or the systemd daemon during node boot up. On Kubernetes nodes, you can still use systemd to run your system processes, but then you can't take advantage of all the features Kubernetes provides.

#### 4.4.1 Using a DaemonSet to run a pod on every node

To run a pod on all cluster nodes, you create a DaemonSet object, which is much like a ReplicationController or a ReplicaSet, except that pods created by a DaemonSet already have a target node specified and skip the Kubernetes Scheduler. They aren't scattered around the cluster randomly.

A DaemonSet makes sure it creates as many pods as there are nodes and deploys each one on its own node, as shown in figure 4.8.

Whereas a ReplicaSet (or ReplicationController) makes sure that a desired number of pod replicas exist in the cluster, a DaemonSet doesn't have any notion of a desired replica count. It doesn't need it because its job is to ensure that a pod matching its pod selector is running on each node.

If a node goes down, the DaemonSet doesn't cause the pod to be created elsewhere. But when a new node is added to the cluster, the DaemonSet immediately deploys a new pod instance to it. It also does the same if someone inadvertently deletes one of the pods, leaving the node without the DaemonSet's pod. Like a ReplicaSet, a DaemonSet creates the pod from the pod template configured in it.

#### 4.4.2 Using a DaemonSet to run pods only on certain nodes

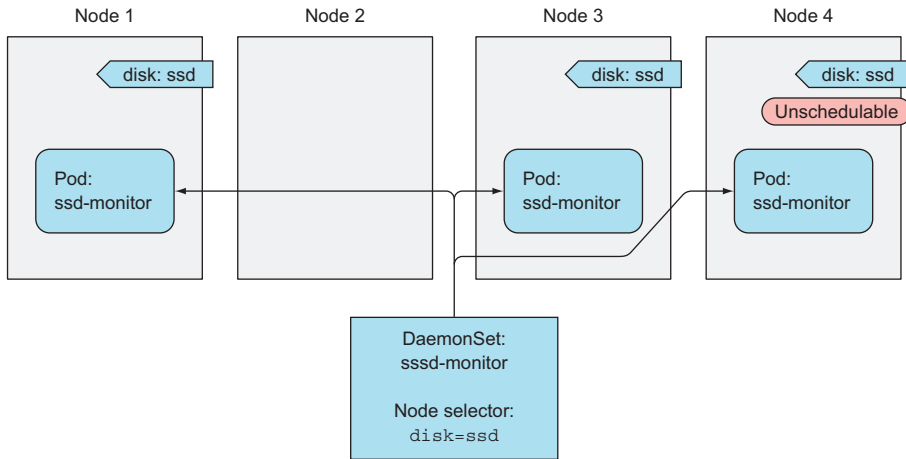
A DaemonSet deploys pods to all nodes in the cluster, unless you specify that the pods should only run on a subset of all the nodes. This is done by specifying the `nodeSelector` property in the pod template, which is part of the DaemonSet definition (similar to the pod template in a ReplicaSet or ReplicationController).

You've already used node selectors to deploy a pod onto specific nodes in chapter 3. A node selector in a DaemonSet is similar—it defines the nodes the DaemonSet must deploy its pods to.

**NOTE** Later in the book, you'll learn that nodes can be made unschedulable, preventing pods from being deployed to them. A DaemonSet will deploy pods even to such nodes, because the `unschedulable` attribute is only used by the Scheduler, whereas pods managed by a DaemonSet bypass the Scheduler completely. This is usually desirable, because DaemonSets are meant to run system services, which usually need to run even on unschedulable nodes.

#### EXPLAINING DAEMONSETS WITH AN EXAMPLE

Let's imagine having a daemon called `ssd-monitor` that needs to run on all nodes that contain a solid-state drive (SSD). You'll create a DaemonSet that runs this daemon on all nodes that are marked as having an SSD. The cluster administrators have added the `disk=ssd` label to all such nodes, so you'll create the DaemonSet with a node selector that only selects nodes with that label, as shown in figure 4.9.



**Figure 4.9** Using a DaemonSet with a node selector to deploy system pods only on certain nodes

### CREATING A DAEMONSET YAML DEFINITION

You'll create a DaemonSet that runs a mock `ssd-monitor` process, which prints "SSD OK" to the standard output every five seconds. I've already prepared the mock container image and pushed it to Docker Hub, so you can use it instead of building your own. Create the YAML for the DaemonSet, as shown in the following listing.

#### Listing 4.10 A YAML for a DaemonSet: `ssd-monitor-daemonset.yaml`

```
apiVersion: apps/v1beta2
kind: DaemonSet
metadata:
  name: ssd-monitor
spec:
  selector:
    matchLabels:
      app: ssd-monitor
  template:
    metadata:
      labels:
        app: ssd-monitor
    spec:
      nodeSelector:
        disk: ssd
      containers:
        - name: main
          image: luksa/ssd-monitor
```

DaemonSets are in the apps API group, version v1beta2.

The pod template includes a node selector, which selects nodes with the `disk=ssd` label.

You're defining a DaemonSet that will run a pod with a single container based on the `luksa/ssd-monitor` container image. An instance of this pod will be created for each node that has the `disk=ssd` label.

**CREATING THE DAEMONSET**

You'll create the DaemonSet like you always create resources from a YAML file:

```
$ kubectl create -f ssd-monitor-daemonset.yaml
daemonset "ssd-monitor" created
```

Let's see the created DaemonSet:

```
$ kubectl get ds
NAME           DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE-SELECTOR
ssd-monitor    0         0         0       0            0          disk=ssd
```

Those zeroes look strange. Didn't the DaemonSet deploy any pods? List the pods:

```
$ kubectl get po
No resources found.
```

Where are the pods? Do you know what's going on? Yes, you forgot to label your nodes with the `disk=ssd` label. No problem—you can do that now. The DaemonSet should detect that the nodes' labels have changed and deploy the pod to all nodes with a matching label. Let's see if that's true.

**ADDING THE REQUIRED LABEL TO YOUR NODE(S)**

Regardless if you're using Minikube, GKE, or another multi-node cluster, you'll need to list the nodes first, because you'll need to know the node's name when labeling it:

```
$ kubectl get node
NAME           STATUS    AGE           VERSION
minikube       Ready     4d            v1.6.0
```

Now, add the `disk=ssd` label to one of your nodes like this:

```
$ kubectl label node minikube disk=ssd
node "minikube" labeled
```

**NOTE** Replace `minikube` with the name of one of your nodes if you're not using Minikube.

The DaemonSet should have created one pod now. Let's see:

```
$ kubectl get po
NAME                READY   STATUS    RESTARTS   AGE
ssd-monitor-hgxwq   1/1     Running   0          35s
```

Okay; so far so good. If you have multiple nodes and you add the same label to further nodes, you'll see the DaemonSet spin up pods for each of them.

**REMOVING THE REQUIRED LABEL FROM THE NODE**

Now, imagine you've made a mistake and have mislabeled one of the nodes. It has a spinning disk drive, not an SSD. What happens if you change the node's label?

```
$ kubectl label node minikube disk=hdd --overwrite
node "minikube" labeled
```

Let's see if the change has any effect on the pod that was running on that node:

```
$ kubectl get po
NAME                READY    STATUS      RESTARTS   AGE
ssd-monitor-hgxwq   1/1     Terminating 0           4m
```

The pod is being terminated. But you knew that was going to happen, right? This wraps up your exploration of DaemonSets, so you may want to delete your `ssd-monitor` DaemonSet. If you still have any other daemon pods running, you'll see that deleting the DaemonSet deletes those pods as well.

## 4.5 *Running pods that perform a single completable task*

Up to now, we've only talked about pods that need to run continuously. You'll have cases where you only want to run a task that terminates after completing its work. ReplicationControllers, ReplicaSets, and DaemonSets run continuous tasks that are never considered completed. Processes in such pods are restarted when they exit. But in a completable task, after its process terminates, it should not be restarted again.

### 4.5.1 *Introducing the Job resource*

Kubernetes includes support for this through the Job resource, which is similar to the other resources we've discussed in this chapter, but it allows you to run a pod whose container isn't restarted when the process running inside finishes successfully. Once it does, the pod is considered complete.

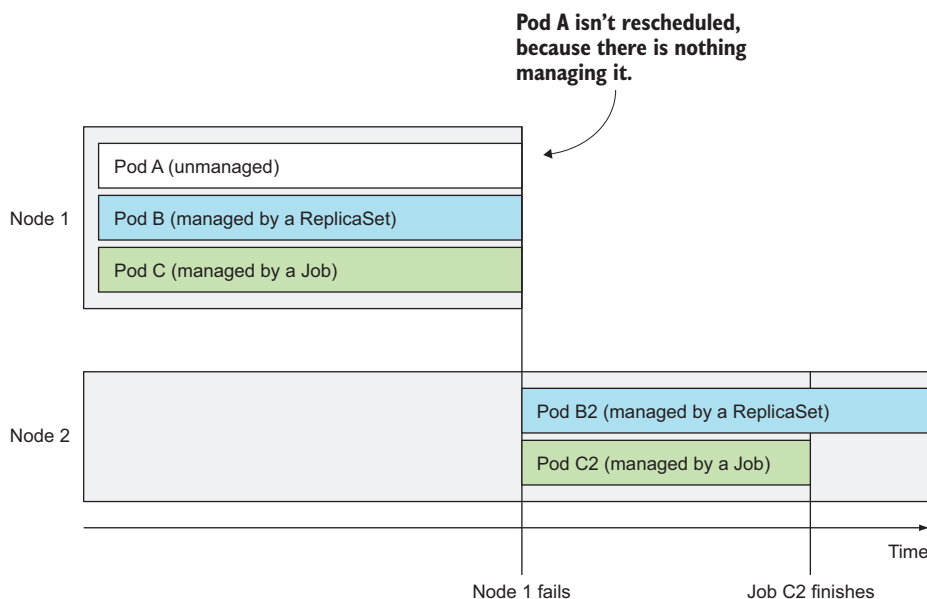
In the event of a node failure, the pods on that node that are managed by a Job will be rescheduled to other nodes the way ReplicaSet pods are. In the event of a failure of the process itself (when the process returns an error exit code), the Job can be configured to either restart the container or not.

Figure 4.10 shows how a pod created by a Job is rescheduled to a new node if the node it was initially scheduled to fails. The figure also shows both a managed pod, which isn't rescheduled, and a pod backed by a ReplicaSet, which is.

For example, Jobs are useful for ad hoc tasks, where it's crucial that the task finishes properly. You could run the task in an unmanaged pod and wait for it to finish, but in the event of a node failing or the pod being evicted from the node while it is performing its task, you'd need to manually recreate it. Doing this manually doesn't make sense—especially if the job takes hours to complete.

An example of such a job would be if you had data stored somewhere and you needed to transform and export it somewhere. You're going to emulate this by running a container image built on top of the `busybox` image, which invokes the `sleep` command for two minutes. I've already built the image and pushed it to Docker Hub, but you can peek into its Dockerfile in the book's code archive.





**Figure 4.10** Pods managed by Jobs are rescheduled until they finish successfully.

### 4.5.2 Defining a Job resource

Create the Job manifest as in the following listing.

**Listing 4.11** A YAML definition of a Job: exporter.yaml

```
apiVersion: batch/v1
kind: Job
metadata:
  name: batch-job
spec:
  template:
    metadata:
      labels:
        app: batch-job
    spec:
      restartPolicy: OnFailure
      containers:
        - name: main
          image: luksa/batch-job
```

**Jobs are in the batch API group, version v1.**

**You're not specifying a pod selector (it will be created based on the labels in the pod template).**

**Jobs can't use the default restart policy, which is Always.**

Jobs are part of the batch API group and v1 API version. The YAML defines a resource of type Job that will run the luksa/batch-job image, which invokes a process that runs for exactly 120 seconds and then exits.

In a pod's specification, you can specify what Kubernetes should do when the processes running in the container finish. This is done through the `restartPolicy`

pod spec property, which defaults to Always. Job pods can't use the default policy, because they're not meant to run indefinitely. Therefore, you need to explicitly set the restart policy to either OnFailure or Never. This setting is what prevents the container from being restarted when it finishes (not the fact that the pod is being managed by a Job resource).

### 4.5.3 Seeing a Job run a pod

After you create this Job with the `kubectl create` command, you should see it start up a pod immediately:

```
$ kubectl get jobs
NAME          DESIRED  SUCCESSFUL  AGE
batch-job     1        0           2s

$ kubectl get po
NAME          READY    STATUS    RESTARTS  AGE
batch-job-28qf4  1/1      Running   0         4s
```

After the two minutes have passed, the pod will no longer show up in the pod list and the Job will be marked as completed. By default, completed pods aren't shown when you list pods, unless you use the `--show-all` (or `-a`) switch:

```
$ kubectl get po -a
NAME          READY    STATUS    RESTARTS  AGE
batch-job-28qf4  0/1      Completed  0         2m
```

The reason the pod isn't deleted when it completes is to allow you to examine its logs; for example:

```
$ kubectl logs batch-job-28qf4
Fri Apr 29 09:58:22 UTC 2016 Batch job starting
Fri Apr 29 10:00:22 UTC 2016 Finished successfully
```

The pod will be deleted when you delete it or the Job that created it. Before you do that, let's look at the Job resource again:

```
$ kubectl get job
NAME          DESIRED  SUCCESSFUL  AGE
batch-job     1        1           9m
```

The Job is shown as having completed successfully. But why is that piece of information shown as a number instead of as yes or true? And what does the DESIRED column indicate?

### 4.5.4 Running multiple pod instances in a Job

Jobs may be configured to create more than one pod instance and run them in parallel or sequentially. This is done by setting the `completions` and the `parallelism` properties in the Job spec.

**RUNNING JOB PODS SEQUENTIALLY**

If you need a Job to run more than once, you set completions to how many times you want the Job's pod to run. The following listing shows an example.

**Listing 4.12 A Job requiring multiple completions: multi-completion-batch-job.yaml**

```
apiVersion: batch/v1
kind: Job
metadata:
  name: multi-completion-batch-job
spec:
  completions: 5
  template:
    <template is the same as in listing 4.11>
```

Setting completions to 5 makes this Job run five pods sequentially.

This Job will run five pods one after the other. It initially creates one pod, and when the pod's container finishes, it creates the second pod, and so on, until five pods complete successfully. If one of the pods fails, the Job creates a new pod, so the Job may create more than five pods overall.

**RUNNING JOB PODS IN PARALLEL**

Instead of running single Job pods one after the other, you can also make the Job run multiple pods in parallel. You specify how many pods are allowed to run in parallel with the parallelism Job spec property, as shown in the following listing.

**Listing 4.13 Running Job pods in parallel: multi-completion-parallel-batch-job.yaml**

```
apiVersion: batch/v1
kind: Job
metadata:
  name: multi-completion-batch-job
spec:
  completions: 5
  parallelism: 2
  template:
    <same as in listing 4.11>
```

This job must ensure five pods complete successfully.

Up to two pods can run in parallel.

By setting parallelism to 2, the Job creates two pods and runs them in parallel:

```
$ kubectl get po
```

NAME	READY	STATUS	RESTARTS	AGE
multi-completion-batch-job-lmmnk	1/1	Running	0	21s
multi-completion-batch-job-qx4nq	1/1	Running	0	21s

As soon as one of them finishes, the Job will run the next pod, until five pods finish successfully.

### SCALING A JOB

You can even change a Job's parallelism property while the Job is running. This is similar to scaling a ReplicaSet or ReplicationController, and can be done with the `kubectl scale` command:

```
$ kubectl scale job multi-completion-batch-job --replicas 3
job "multi-completion-batch-job" scaled
```

Because you've increased parallelism from 2 to 3, another pod is immediately spun up, so three pods are now running.

#### 4.5.5 *Limiting the time allowed for a Job pod to complete*

We need to discuss one final thing about Jobs. How long should the Job wait for a pod to finish? What if the pod gets stuck and can't finish at all (or it can't finish fast enough)?

A pod's time can be limited by setting the `activeDeadlineSeconds` property in the pod spec. If the pod runs longer than that, the system will try to terminate it and will mark the Job as failed.

**NOTE** You can configure how many times a Job can be retried before it is marked as failed by specifying the `spec.backoffLimit` field in the Job manifest. If you don't explicitly specify it, it defaults to 6.

## 4.6 *Scheduling Jobs to run periodically or once in the future*

Job resources run their pods immediately when you create the Job resource. But many batch jobs need to be run at a specific time in the future or repeatedly in the specified interval. In Linux- and UNIX-like operating systems, these jobs are better known as cron jobs. Kubernetes supports them, too.

A cron job in Kubernetes is configured by creating a `CronJob` resource. The schedule for running the job is specified in the well-known cron format, so if you're familiar with regular cron jobs, you'll understand Kubernetes' `CronJobs` in a matter of seconds.

At the configured time, Kubernetes will create a Job resource according to the Job template configured in the `CronJob` object. When the Job resource is created, one or more pod replicas will be created and started according to the Job's pod template, as you learned in the previous section. There's nothing more to it.

Let's look at how to create `CronJobs`.

### 4.6.1 *Creating a CronJob*

Imagine you need to run the batch job from your previous example every 15 minutes. To do that, create a `CronJob` resource with the following specification.

**Listing 4.14** YAML for a CronJob resource: cronjob.yaml

```

apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: batch-job-every-fifteen-minutes
spec:
  schedule: "0,15,30,45 * * * *"
  jobTemplate:
    spec:
      template:
        metadata:
          labels:
            app: periodic-batch-job
        spec:
          restartPolicy: OnFailure
          containers:
            - name: main
              image: luksa/batch-job

```

API group is batch,  
version is v1beta1

This job should run at the  
0, 15, 30 and 45 minutes of  
every hour, every day.

The template for the  
Job resources that  
will be created by  
this CronJob

As you can see, it's not too complicated. You've specified a schedule and a template from which the Job objects will be created.

**CONFIGURING THE SCHEDULE**

If you're unfamiliar with the cron schedule format, you'll find great tutorials and explanations online, but as a quick introduction, from left to right, the schedule contains the following five entries:

- Minute
- Hour
- Day of month
- Month
- Day of week.

In the example, you want to run the job every 15 minutes, so the schedule needs to be "0,15,30,45 \* \* \* \*", which means at the 0, 15, 30 and 45 minutes mark of every hour (first asterisk), of every day of the month (second asterisk), of every month (third asterisk) and on every day of the week (fourth asterisk).

If, instead, you wanted it to run every 30 minutes, but only on the first day of the month, you'd set the schedule to "0,30 \* 1 \* \*", and if you want it to run at 3AM every Sunday, you'd set it to "0 3 \* \* 0" (the last zero stands for Sunday).

**CONFIGURING THE JOB TEMPLATE**

A CronJob creates Job resources from the jobTemplate property configured in the CronJob spec, so refer to section 4.5 for more information on how to configure it.

**4.6.2 Understanding how scheduled jobs are run**

Job resources will be created from the CronJob resource at approximately the scheduled time. The Job then creates the pods.

It may happen that the Job or pod is created and run relatively late. You may have a hard requirement for the job to not be started too far over the scheduled time. In that case, you can specify a deadline by specifying the `startingDeadlineSeconds` field in the CronJob specification as shown in the following listing.

#### Listing 4.15 Specifying a `startingDeadlineSeconds` for a CronJob

```
apiVersion: batch/v1beta1
kind: CronJob
spec:
  schedule: "0,15,30,45 * * * *"
  startingDeadlineSeconds: 15
  ...
```

At the latest, the pod must start running at 15 seconds past the scheduled time.

In the example in listing 4.15, one of the times the job is supposed to run is 10:30:00. If it doesn't start by 10:30:15 for whatever reason, the job will not run and will be shown as Failed.

In normal circumstances, a CronJob always creates only a single Job for each execution configured in the schedule, but it may happen that two Jobs are created at the same time, or none at all. To combat the first problem, your jobs should be idempotent (running them multiple times instead of once shouldn't lead to unwanted results). For the second problem, make sure that the next job run performs any work that should have been done by the previous (missed) run.

## 4.7 Summary

You've now learned how to keep pods running and have them rescheduled in the event of node failures. You should now know that

- You can specify a liveness probe to have Kubernetes restart your container as soon as it's no longer healthy (where the app defines what's considered healthy).
- Pods shouldn't be created directly, because they will not be re-created if they're deleted by mistake, if the node they're running on fails, or if they're evicted from the node.
- ReplicationControllers always keep the desired number of pod replicas running.
- Scaling pods horizontally is as easy as changing the desired replica count on a ReplicationController.
- Pods aren't owned by the ReplicationControllers and can be moved between them if necessary.
- A ReplicationController creates new pods from a pod template. Changing the template has no effect on existing pods.

- ReplicationControllers should be replaced with ReplicaSets and Deployments, which provide the same functionality, but with additional powerful features.
- ReplicationControllers and ReplicaSets schedule pods to random cluster nodes, whereas DaemonSets make sure every node runs a single instance of a pod defined in the DaemonSet.
- Pods that perform a batch task should be created through a Kubernetes Job resource, not directly or through a ReplicationController or similar object.
- Jobs that need to run sometime in the future can be created through CronJob resources.

# 5

## *Services: enabling clients to discover and talk to pods*

---

### **This chapter covers**

- Creating Service resources to expose a group of pods at a single address
- Discovering services in the cluster
- Exposing services to external clients
- Connecting to external services from inside the cluster
- Controlling whether a pod is ready to be part of the service or not
- Troubleshooting services

You've learned about pods and how to deploy them through ReplicaSets and similar resources to ensure they keep running. Although certain pods can do their work independently of an external stimulus, many applications these days are meant to respond to external requests. For example, in the case of microservices, pods will usually respond to HTTP requests coming either from other pods inside the cluster or from clients outside the cluster.

Pods need a way of finding other pods if they want to consume the services they provide. Unlike in the non-Kubernetes world, where a sysadmin would configure



each client app by specifying the exact IP address or hostname of the server providing the service in the client's configuration files, doing the same in Kubernetes wouldn't work, because

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

To solve these problems, Kubernetes also provides another resource type—Services—that we'll discuss in this chapter.

## 5.1 Introducing services

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

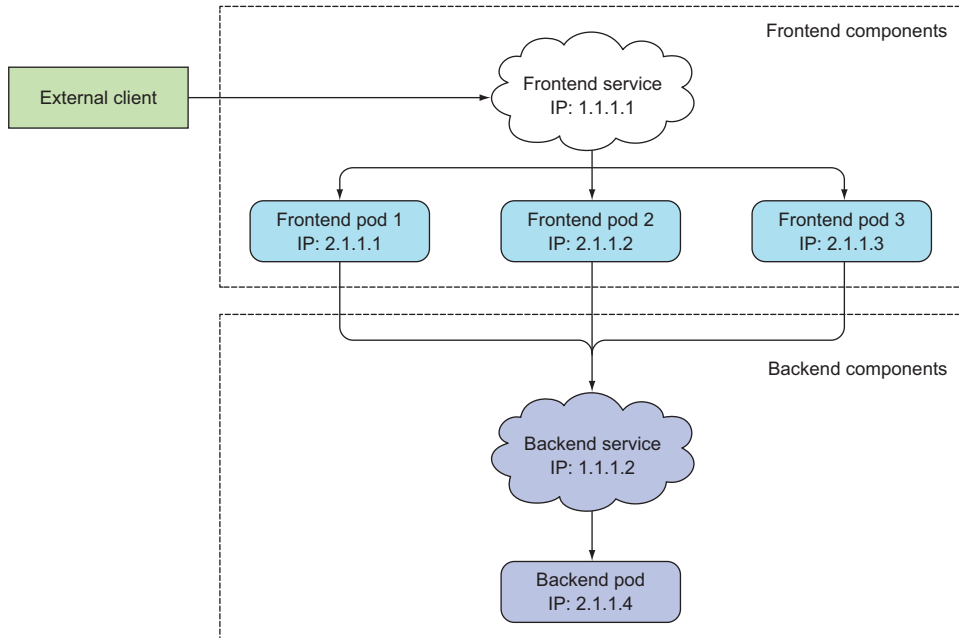
### EXPLAINING SERVICES WITH AN EXAMPLE

Let's revisit the example where you have a frontend web server and a backend database server. There may be multiple pods that all act as the frontend, but there may only be a single backend database pod. You need to solve two problems to make the system function:

- External clients need to connect to the frontend pods without caring if there's only a single web server or hundreds.
- The frontend pods need to connect to the backend database. Because the database runs inside a pod, it may be moved around the cluster over time, causing its IP address to change. You don't want to reconfigure the frontend pods every time the backend database is moved.

By creating a service for the frontend pods and configuring it to be accessible from outside the cluster, you expose a single, constant IP address through which external clients can connect to the pods. Similarly, by also creating a service for the backend pod, you create a stable address for the backend pod. The service address doesn't

change even if the pod's IP address changes. Additionally, by creating the service, you also enable the frontend pods to easily find the backend service by its name through either environment variables or DNS. All the components of your system (the two services, the two sets of pods backing those services, and the interdependencies between them) are shown in figure 5.1.



**Figure 5.1** Both internal and external clients usually connect to pods through services.

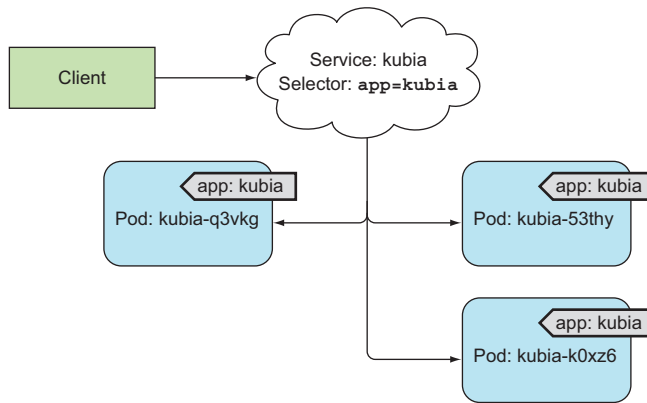
You now understand the basic idea behind services. Now, let's dig deeper by first seeing how they can be created.

### 5.1.1 *Creating services*

As you've seen, a service can be backed by more than one pod. Connections to the service are load-balanced across all the backing pods. But how exactly do you define which pods are part of the service and which aren't?

You probably remember label selectors and how they're used in ReplicationControllers and other pod controllers to specify which pods belong to the same set. The same mechanism is used by services in the same way, as you can see in figure 5.2.

In the previous chapter, you created a ReplicationController which then ran three instances of the pod containing the Node.js app. Create the ReplicationController again and verify three pod instances are up and running. After that, you'll create a Service for those three pods.



**Figure 5.2** Label selectors determine which pods belong to the Service.

### CREATING A SERVICE THROUGH KUBECTL EXPOSE

The easiest way to create a service is through `kubectl expose`, which you’ve already used in chapter 2 to expose the ReplicationController you created earlier. The `expose` command created a Service resource with the same pod selector as the one used by the ReplicationController, thereby exposing all its pods through a single IP address and port.

Now, instead of using the `expose` command, you’ll create a service manually by posting a YAML to the Kubernetes API server.

### CREATING A SERVICE THROUGH A YAML DESCRIPTOR

Create a file called `kubia-svc.yaml` with the following listing’s contents.

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

```

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

The port this service  
will be available on

The container port the  
service will forward to

All pods with the `app=kubia`  
label will be part of this service.

You’re defining a service called `kubia`, which will accept connections on port 80 and route each connection to port 8080 of one of the pods matching the `app=kubia` label selector.

Go ahead and create the service by posting the file using `kubectl create`.

**EXAMINING YOUR NEW SERVICE**

After posting the YAML, you can list all Service resources in your namespace and see that an internal cluster IP has been assigned to your service:

```
$ kubectl get svc
```

NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	10.111.240.1	<none>	443/TCP	30d
kubia	10.111.249.153	<none>	80/TCP	6m

Here's your service.

The list shows that the IP address assigned to the service is 10.111.249.153. Because this is the cluster IP, it's only accessible from inside the cluster. The primary purpose of services is exposing groups of pods to other pods in the cluster, but you'll usually also want to expose services externally. You'll see how to do that later. For now, let's use your service from inside the cluster and see what it does.

**TESTING YOUR SERVICE FROM WITHIN THE CLUSTER**

You can send requests to your service from within the cluster in a few ways:

- The obvious way is to create a pod that will send the request to the service's cluster IP and log the response. You can then examine the pod's log to see what the service's response was.
- You can `ssh` into one of the Kubernetes nodes and use the `curl` command.
- You can execute the `curl` command inside one of your existing pods through the `kubectl exec` command.

Let's go for the last option, so you also learn how to run commands in existing pods.

**REMOTELY EXECUTING COMMANDS IN RUNNING CONTAINERS**

The `kubectl exec` command allows you to remotely run arbitrary commands inside an existing container of a pod. This comes in handy when you want to examine the contents, state, and/or environment of a container. List the pods with the `kubectl get pods` command and choose one as your target for the `exec` command (in the following example, I've chosen the `kubia-7nog1` pod as the target). You'll also need to obtain the cluster IP of your service (using `kubectl get svc`, for example). When running the following commands yourself, be sure to replace the pod name and the service IP with your own:

```
$ kubectl exec kubia-7nog1 -- curl -s http://10.111.249.153
You've hit kubia-gzwli
```

If you've used `ssh` to execute commands on a remote system before, you'll recognize that `kubectl exec` isn't much different.

### Why the double dash?

The double dash (--) in the command signals the end of command options for `kubectl`. Everything after the double dash is the command that should be executed inside the pod. Using the double dash isn't necessary if the command has no arguments that start with a dash. But in your case, if you don't use the double dash there, the `-s` option would be interpreted as an option for `kubectl exec` and would result in the following strange and highly misleading error:

```
$ kubectl exec kubia-7nog1 curl -s http://10.111.249.153
The connection to the server 10.111.249.153 was refused - did you
specify the right host or port?
```

This has nothing to do with your service refusing the connection. It's because `kubectl` is not able to connect to an API server at 10.111.249.153 (the `-s` option is used to tell `kubectl` to connect to a different API server than the default).

Let's go over what transpired when you ran the command. Figure 5.3 shows the sequence of events. You instructed Kubernetes to execute the `curl` command inside the container of one of your pods. Curl sent an HTTP request to the service IP, which is backed by three pods. The Kubernetes service proxy intercepted the connection, selected a random pod among the three pods, and forwarded the request to it. Node.js running inside that pod then handled the request and returned an HTTP response containing the pod's name. Curl then printed the response to the standard output, which was intercepted and printed to its standard output on your local machine by `kubectl`.

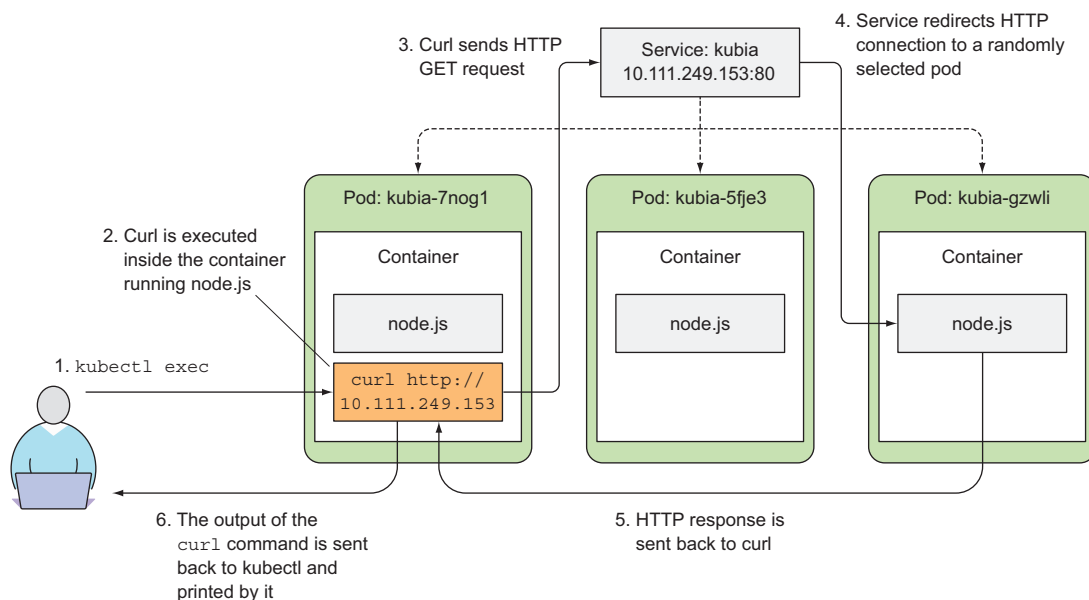


Figure 5.3 Using `kubectl exec` to test out a connection to the service by running `curl` in one of the pods

In the previous example, you executed the `curl` command as a separate process, but inside the pod's main container. This isn't much different from the actual main process in the container talking to the service.

#### CONFIGURING SESSION AFFINITY ON THE SERVICE

If you execute the same command a few more times, you should hit a different pod with every invocation, because the service proxy normally forwards each connection to a randomly selected backing pod, even if the connections are coming from the same client.

If, on the other hand, you want all requests made by a certain client to be redirected to the same pod every time, you can set the service's `sessionAffinity` property to `ClientIP` (instead of `None`, which is the default), as shown in the following listing.

#### Listing 5.2 A example of a service with `ClientIP` session affinity configured

```
apiVersion: v1
kind: Service
spec:
  sessionAffinity: ClientIP
  ...
```

This makes the service proxy redirect all requests originating from the same client IP to the same pod. As an exercise, you can create an additional service with session affinity set to `ClientIP` and try sending requests to it.

Kubernetes supports only two types of service session affinity: `None` and `ClientIP`. You may be surprised it doesn't have a cookie-based session affinity option, but you need to understand that Kubernetes services don't operate at the HTTP level. Services deal with TCP and UDP packets and don't care about the payload they carry. Because cookies are a construct of the HTTP protocol, services don't know about them, which explains why session affinity cannot be based on cookies.

#### EXPOSING MULTIPLE PORTS IN THE SAME SERVICE

Your service exposes only a single port, but services can also support multiple ports. For example, if your pods listened on two ports—let's say 8080 for HTTP and 8443 for HTTPS—you could use a single service to forward both port 80 and 443 to the pod's ports 8080 and 8443. You don't need to create two different services in such cases. Using a single, multi-port service exposes all the service's ports through a single cluster IP.

**NOTE** When creating a service with multiple ports, you must specify a name for each port.

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

#### Listing 5.3 Specifying multiple ports in a service definition

```
apiVersion: v1
kind: Service
metadata:
  name: kubia
```

```
spec:
  ports:
    - name: http
      port: 80
      targetPort: 8080
    - name: https
      port: 443
      targetPort: 8443
  selector:
    app: kuba
```

Port 80 is mapped to the pods' port 8080.

Port 443 is mapped to pods' port 8443.

The label selector always applies to the whole service.

**NOTE** The label selector applies to the service as a whole—it can't be configured for each port individually. If you want different ports to map to different subsets of pods, you need to create two services.

Because your kuba pods don't listen on multiple ports, creating a multi-port service and a multi-port pod is left as an exercise to you.

### USING NAMED PORTS

In all these examples, you've referred to the target port by its number, but you can also give a name to each pod's port and refer to it by name in the service spec. This makes the service spec slightly clearer, especially if the port numbers aren't well-known.

For example, suppose your pod defines names for its ports as shown in the following listing.

#### Listing 5.4 Specifying port names in a pod definition

```
kind: Pod
spec:
  containers:
    - name: kuba
      ports:
        - name: http
          containerPort: 8080
        - name: https
          containerPort: 8443
```

Container's port 8080 is called http

Port 8443 is called https.

You can then refer to those ports by name in the service spec, as shown in the following listing.

#### Listing 5.5 Referring to named ports in a service

```
apiVersion: v1
kind: Service
spec:
  ports:
    - name: http
      port: 80
      targetPort: http
    - name: https
      port: 443
      targetPort: https
```

Port 80 is mapped to the container's port called http.

Port 443 is mapped to the container's port, whose name is https.

But why should you even bother with naming ports? The biggest benefit of doing so is that it enables you to change port numbers later without having to change the service spec. Your pod currently uses port 8080 for http, but what if you later decide you'd like to move that to port 80?

If you're using named ports, all you need to do is change the port number in the pod spec (while keeping the port's name unchanged). As you spin up pods with the new ports, client connections will be forwarded to the appropriate port numbers, depending on the pod receiving the connection (port 8080 on old pods and port 80 on the new ones).

### 5.1.2 *Discovering services*

By creating a service, you now have a single and stable IP address and port that you can hit to access your pods. This address will remain unchanged throughout the whole lifetime of the service. Pods behind this service may come and go, their IPs may change, their number can go up or down, but they'll always be accessible through the service's single and constant IP address.

But how do the client pods know the IP and port of a service? Do you need to create the service first, then manually look up its IP address and pass the IP to the configuration options of the client pod? Not really. Kubernetes also provides ways for client pods to discover a service's IP and port.

#### DISCOVERING SERVICES THROUGH ENVIRONMENT VARIABLES

When a pod is started, Kubernetes initializes a set of environment variables pointing to each service that exists at that moment. If you create the service before creating the client pods, processes in those pods can get the IP address and port of the service by inspecting their environment variables.

Let's see what those environment variables look like by examining the environment of one of your running pods. You've already learned that you can use the `kubectl exec` command to run a command in the pod, but because you created the service only after your pods had been created, the environment variables for the service couldn't have been set yet. You'll need to address that first.

Before you can see environment variables for your service, you first need to delete all the pods and let the ReplicationController create new ones. You may remember you can delete all pods without specifying their names like this:

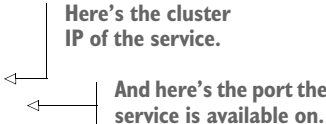
```
$ kubectl delete po --all
pod "kubia-7nog1" deleted
pod "kubia-bf50t" deleted
pod "kubia-gzwli" deleted
```

Now you can list the new pods (I'm sure you know how to do that) and pick one as your target for the `kubectl exec` command. Once you've selected your target pod, you can list environment variables by running the `env` command inside the container, as shown in the following listing.



**Listing 5.6 Service-related environment variables in a container**

```
$ kubectl exec kuba-3inly env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=kuba-3inly
KUBERNETES_SERVICE_HOST=10.111.240.1
KUBERNETES_SERVICE_PORT=443
...
KUBIA_SERVICE_HOST=10.111.249.153
KUBIA_SERVICE_PORT=80
...
```



Here's the cluster IP of the service.

And here's the port the service is available on.

Two services are defined in your cluster: the `kubernetes` and the `kubia` service (you saw this earlier with the `kubectl get svc` command); consequently, two sets of service-related environment variables are in the list. Among the variables that pertain to the `kubia` service you created at the beginning of the chapter, you'll see the `KUBIA_SERVICE_HOST` and the `KUBIA_SERVICE_PORT` environment variables, which hold the IP address and port of the `kubia` service, respectively.

Turning back to the frontend-backend example we started this chapter with, when you have a frontend pod that requires the use of a backend database server pod, you can expose the backend pod through a service called `backend-database` and then have the frontend pod look up its IP address and port through the environment variables `BACKEND_DATABASE_SERVICE_HOST` and `BACKEND_DATABASE_SERVICE_PORT`.

**NOTE** Dashes in the service name are converted to underscores and all letters are uppercased when the service name is used as the prefix in the environment variable's name.

Environment variables are one way of looking up the IP and port of a service, but isn't this usually the domain of DNS? Why doesn't Kubernetes include a DNS server and allow you to look up service IPs through DNS instead? As it turns out, it does!

#### DISCOVERING SERVICES THROUGH DNS

Remember in chapter 3 when you listed pods in the `kube-system` namespace? One of the pods was called `kube-dns`. The `kube-system` namespace also includes a corresponding service with the same name.

As the name suggests, the pod runs a DNS server, which all other pods running in the cluster are automatically configured to use (Kubernetes does that by modifying each container's `/etc/resolv.conf` file). Any DNS query performed by a process running in a pod will be handled by Kubernetes' own DNS server, which knows all the services running in your system.

**NOTE** Whether a pod uses the internal DNS server or not is configurable through the `dnsPolicy` property in each pod's spec.

Each service gets a DNS entry in the internal DNS server, and client pods that know the name of the service can access it through its fully qualified domain name (FQDN) instead of resorting to environment variables.

**CONNECTING TO THE SERVICE THROUGH ITS FQDN**

To revisit the frontend-backend example, a frontend pod can connect to the backend-database service by opening a connection to the following FQDN:

```
backend-database.default.svc.cluster.local
```

backend-database corresponds to the service name, default stands for the namespace the service is defined in, and svc.cluster.local is a configurable cluster domain suffix used in all cluster local service names.

**NOTE** The client must still know the service’s port number. If the service is using a standard port (for example, 80 for HTTP or 5432 for Postgres), that shouldn’t be a problem. If not, the client can get the port number from the environment variable.

Connecting to a service can be even simpler than that. You can omit the `svc.cluster.local` suffix and even the namespace, when the frontend pod is in the same namespace as the database pod. You can thus refer to the service simply as `backend-database`. That’s incredibly simple, right?

Let’s try this. You’ll try to access the `kubia` service through its FQDN instead of its IP. Again, you’ll need to do that inside an existing pod. You already know how to use `kubectl exec` to run a single command in a pod’s container, but this time, instead of running the `curl` command directly, you’ll run the `bash` shell instead, so you can then run multiple commands in the container. This is similar to what you did in chapter 2 when you entered the container you ran with Docker by using the `docker exec -it bash` command.

**RUNNING A SHELL IN A POD’S CONTAINER**

You can use the `kubectl exec` command to run `bash` (or any other shell) inside a pod’s container. This way you’re free to explore the container as long as you want, without having to perform a `kubectl exec` for every command you want to run.

**NOTE** The shell’s binary executable must be available in the container image for this to work.

To use the shell properly, you need to pass the `-it` option to `kubectl exec`:

```
$ kubectl exec -it kubia-3inly bash
root@kubia-3inly:/#
```

You’re now inside the container. You can use the `curl` command to access the `kubia` service in any of the following ways:

```
root@kubia-3inly:/# curl http://kubia.default.svc.cluster.local
You’ve hit kubia-5asi2

root@kubia-3inly:/# curl http://kubia.default
You’ve hit kubia-3inly
```

```
root@kubia-3inly:/# curl http://kubia
You've hit kubia-8awf3
```

You can hit your service by using the service's name as the hostname in the requested URL. You can omit the namespace and the `svc.cluster.local` suffix because of how the DNS resolver inside each pod's container is configured. Look at the `/etc/resolv.conf` file in the container and you'll understand:

```
root@kubia-3inly:/# cat /etc/resolv.conf
search default.svc.cluster.local svc.cluster.local cluster.local ...
```

### UNDERSTANDING WHY YOU CAN'T PING A SERVICE IP

One last thing before we move on. You know how to create services now, so you'll soon create your own. But what if, for whatever reason, you can't access your service?

You'll probably try to figure out what's wrong by entering an existing pod and trying to access the service like you did in the last example. Then, if you still can't access the service with a simple `curl` command, maybe you'll try to ping the service IP to see if it's up. Let's try that now:

```
root@kubia-3inly:/# ping kubia
PING kubia.default.svc.cluster.local (10.111.249.153): 56 data bytes
^C--- kubia.default.svc.cluster.local ping statistics ---
54 packets transmitted, 0 packets received, 100% packet loss
```

Hmm. `curl`-ing the service works, but pinging it doesn't. That's because the service's cluster IP is a virtual IP, and only has meaning when combined with the service port. We'll explain what that means and how services work in chapter 11. I wanted to mention that here because it's the first thing users do when they try to debug a broken service and it catches most of them off guard.

## 5.2 Connecting to services living outside the cluster

Up to now, we've talked about services backed by one or more pods running inside the cluster. But cases exist when you'd like to expose external services through the Kubernetes services feature. Instead of having the service redirect connections to pods in the cluster, you want it to redirect to external IP(s) and port(s).

This allows you to take advantage of both service load balancing and service discovery. Client pods running in the cluster can connect to the external service like they connect to internal services.

### 5.2.1 Introducing service endpoints

Before going into how to do this, let me first shed more light on services. Services don't link to pods directly. Instead, a resource sits in between—the Endpoints resource. You may have already noticed endpoints if you used the `kubectl describe` command on your service, as shown in the following listing.

**Listing 5.7** Full details of a service displayed with `kubectl describe`

```
$ kubectl describe svc kuba
Name:          kuba
Namespace:     default
Labels:        <none>
Selector:      app=kuba
Type:          ClusterIP
IP:            10.111.249.153
Port:          <unset> 80/TCP
Endpoints:     10.108.1.4:8080,10.108.2.5:8080,10.108.2.6:8080
Session Affinity: None
No events.
```

The service's pod selector is used to create the list of endpoints.

The list of pod IPs and ports that represent the endpoints of this service

An Endpoints resource (yes, plural) is a list of IP addresses and ports exposing a service. The Endpoints resource is like any other Kubernetes resource, so you can display its basic info with `kubectl get`:

```
$ kubectl get endpoints kuba
NAME      ENDPOINTS                                     AGE
kuba      10.108.1.4:8080,10.108.2.5:8080,10.108.2.6:8080 1h
```

Although the pod selector is defined in the service spec, it's not used directly when redirecting incoming connections. Instead, the selector is used to build a list of IPs and ports, which is then stored in the Endpoints resource. When a client connects to a service, the service proxy selects one of those IP and port pairs and redirects the incoming connection to the server listening at that location.

## 5.2.2 Manually configuring service endpoints

You may have probably realized this already, but having the service's endpoints decoupled from the service allows them to be configured and updated manually.

If you create a service without a pod selector, Kubernetes won't even create the Endpoints resource (after all, without a selector, it can't know which pods to include in the service). It's up to you to create the Endpoints resource to specify the list of endpoints for the service.

To create a service with manually managed endpoints, you need to create both a Service and an Endpoints resource.

### CREATING A SERVICE WITHOUT A SELECTOR

You'll first create the YAML for the service itself, as shown in the following listing.

**Listing 5.8** A service without a pod selector: `external-service.yaml`

```
apiVersion: v1
kind: Service
metadata:
  name: external-service
spec:
  ports:
    - port: 80
```

The name of the service must match the name of the Endpoints object (see next listing).

This service has no selector defined.

You're defining a service called `external-service` that will accept incoming connections on port 80. You didn't define a pod selector for the service.

#### CREATING AN ENDPOINTS RESOURCE FOR A SERVICE WITHOUT A SELECTOR

Endpoints are a separate resource and not an attribute of a service. Because you created the service without a selector, the corresponding Endpoints resource hasn't been created automatically, so it's up to you to create it. The following listing shows its YAML manifest.

**Listing 5.9** A manually created Endpoints resource: `external-service-endpoints.yaml`

```
apiVersion: v1
kind: Endpoints
metadata:
  name: external-service
subsets:
  - addresses:
    - ip: 11.11.11.11
    - ip: 22.22.22.22
    ports:
    - port: 80
```

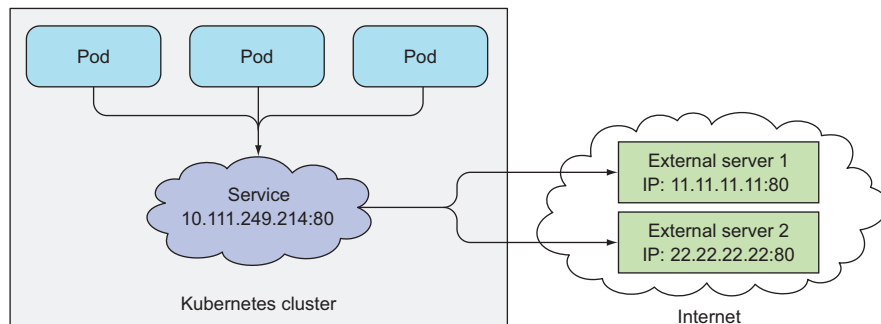
← The name of the Endpoints object must match the name of the service (see previous listing).

← The IPs of the endpoints that the service will forward connections to

← The target port of the endpoints

The Endpoints object needs to have the same name as the service and contain the list of target IP addresses and ports for the service. After both the Service and the Endpoints resource are posted to the server, the service is ready to be used like any regular service with a pod selector. Containers created after the service is created will include the environment variables for the service, and all connections to its IP:port pair will be load balanced between the service's endpoints.

Figure 5.4 shows three pods connecting to the service with external endpoints.



**Figure 5.4** Pods consuming a service with two external endpoints.

If you later decide to migrate the external service to pods running inside Kubernetes, you can add a selector to the service, thereby making its Endpoints managed automatically. The same is also true in reverse—by removing the selector from a Service,

Kubernetes stops updating its Endpoints. This means a service IP address can remain constant while the actual implementation of the service is changed.

### 5.2.3 *Creating an alias for an external service*

Instead of exposing an external service by manually configuring the service's Endpoints, a simpler method allows you to refer to an external service by its fully qualified domain name (FQDN).

#### CREATING AN ExternalName SERVICE

To create a service that serves as an alias for an external service, you create a Service resource with the type field set to `ExternalName`. For example, let's imagine there's a public API available at [api.somecompany.com](https://api.somecompany.com). You can define a service that points to it as shown in the following listing.

**Listing 5.10** An `ExternalName`-type service: `external-service-externalname.yaml`

```
apiVersion: v1
kind: Service
metadata:
  name: external-service
spec:
  type: ExternalName
  externalName: someapi.somecompany.com
  ports:
    - port: 80
```



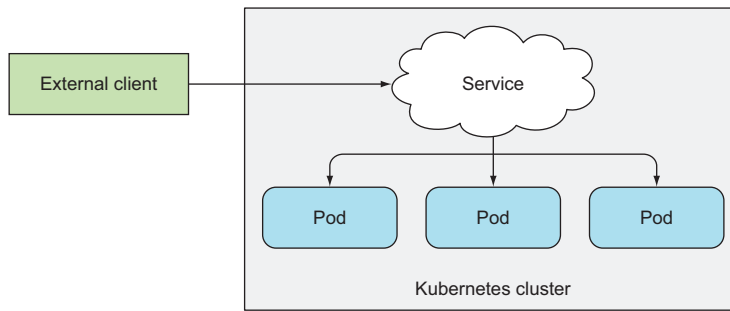
After the service is created, pods can connect to the external service through the `external-service.default.svc.cluster.local` domain name (or even `external-service`) instead of using the service's actual FQDN. This hides the actual service name and its location from pods consuming the service, allowing you to modify the service definition and point it to a different service any time later, by only changing the `externalName` attribute or by changing the type back to `ClusterIP` and creating an Endpoints object for the service—either manually or by specifying a label selector on the service and having it created automatically.

`ExternalName` services are implemented solely at the DNS level—a simple `CNAME` DNS record is created for the service. Therefore, clients connecting to the service will connect to the external service directly, bypassing the service proxy completely. For this reason, these types of services don't even get a cluster IP.

**NOTE** A `CNAME` record points to a fully qualified domain name instead of a numeric IP address.

### 5.3 *Exposing services to external clients*

Up to now, we've only talked about how services can be consumed by pods from inside the cluster. But you'll also want to expose certain services, such as frontend web servers, to the outside, so external clients can access them, as depicted in figure 5.5.



**Figure 5.5** Exposing a service to external clients

You have a few ways to make a service accessible externally:

- *Setting the service type to NodePort*—For a NodePort service, each cluster node opens a port on the node itself (hence the name) and redirects traffic received on that port to the underlying service. The service isn't accessible only at the internal cluster IP and port, but also through a dedicated port on all nodes.
- *Setting the service type to LoadBalancer, an extension of the NodePort type*—This makes the service accessible through a dedicated load balancer, provisioned from the cloud infrastructure Kubernetes is running on. The load balancer redirects traffic to the node port across all the nodes. Clients connect to the service through the load balancer's IP.
- *Creating an Ingress resource, a radically different mechanism for exposing multiple services through a single IP address*—It operates at the HTTP level (network layer 7) and can thus offer more features than layer 4 services can. We'll explain Ingress resources in section 5.4.

### 5.3.1 Using a NodePort service

The first method of exposing a set of pods to external clients is by creating a service and setting its type to NodePort. By creating a NodePort service, you make Kubernetes reserve a port on all its nodes (the same port number is used across all of them) and forward incoming connections to the pods that are part of the service.

This is similar to a regular service (their actual type is ClusterIP), but a NodePort service can be accessed not only through the service's internal cluster IP, but also through any node's IP and the reserved node port.

This will make more sense when you try interacting with a NodePort service.

#### CREATING A NODEPORT SERVICE

You'll now create a NodePort service to see how you can use it. The following listing shows the YAML for the service.

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

```

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

```

Set the service type to NodePort.

This is the port of the service's internal cluster IP.

This is the target port of the backing pods.

The service will be accessible through port 30123 of each of your cluster nodes.

You set the type to `NodePort` and specify the node port this service should be bound to across all cluster nodes. Specifying the port isn't mandatory; Kubernetes will choose a random port if you omit it.

**NOTE** When you create the service in GKE, `kubectl` prints out a warning about having to configure firewall rules. We'll see how to do that soon.

#### EXAMINING YOUR NODEPORT SERVICE

Let's see the basic information of your service to learn more about it:

```

$ kubectl get svc kubia-nodeport
NAME          CLUSTER-IP      EXTERNAL-IP      PORT(S)          AGE
kubia-nodeport 10.111.254.223  <nodes>          80:30123/TCP     2m

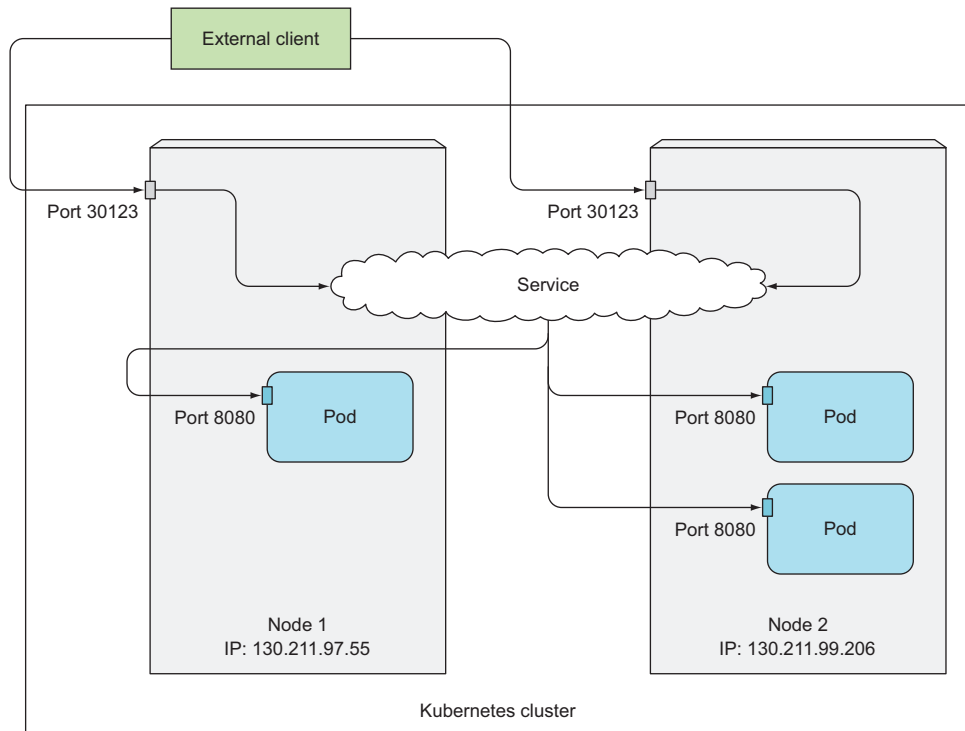
```

Look at the `EXTERNAL-IP` column. It shows `<nodes>`, indicating the service is accessible through the IP address of any cluster node. The `PORT(S)` column shows both the internal port of the cluster IP (80) and the node port (30123). The service is accessible at the following addresses:

- 10.11.254.223:80
- <1st node's IP>:30123
- <2nd node's IP>:30123, and so on.

Figure 5.6 shows your service exposed on port 30123 of both of your cluster nodes (this applies if you're running this on GKE; Minikube only has a single node, but the principle is the same). An incoming connection to one of those ports will be redirected to a randomly selected pod, which may or may not be the one running on the node the connection is being made to.





**Figure 5.6** An external client connecting to a NodePort service either through Node 1 or 2

A connection received on port 30123 of the first node might be forwarded either to the pod running on the first node or to one of the pods running on the second node.

#### CHANGING FIREWALL RULES TO LET EXTERNAL CLIENTS ACCESS OUR NODEPORT SERVICE

As I've mentioned previously, before you can access your service through the node port, you need to configure the Google Cloud Platform's firewalls to allow external connections to your nodes on that port. You'll do this now:

```
$ gcloud compute firewall-rules create kubia-svc-rule --allow=tcp:30123
Created [https://www.googleapis.com/compute/v1/projects/kubia-1295/global/firewalls/kubia-svc-rule].
NAME          NETWORK  SRC_RANGES  RULES      SRC_TAGS  TARGET_TAGS
kubia-svc-rule  default  0.0.0.0/0   tcp:30123
```

You can access your service through port 30123 of one of the node's IPs. But you need to figure out the IP of a node first. Refer to the sidebar on how to do that.

### Using JSONPath to get the IPs of all your nodes

You can find the IP in the JSON or YAML descriptors of the nodes. But instead of sifting through the relatively large JSON, you can tell `kubectl` to print out only the node IP instead of the whole service definition:

```
$ kubectl get nodes -o jsonpath='{.items[*].status.  
➡ addresses[?(@.type=="ExternalIP")].address}'  
130.211.97.55 130.211.99.206
```

You're telling `kubectl` to only output the information you want by specifying a JSONPath. You're probably familiar with XPath and how it's used with XML. JSONPath is basically XPath for JSON. The JSONPath in the previous example instructs `kubectl` to do the following:

- Go through all the elements in the `items` attribute.
- For each element, enter the `status` attribute.
- Filter elements of the `addresses` attribute, taking only those that have the `type` attribute set to `ExternalIP`.
- Finally, print the `address` attribute of the filtered elements.

To learn more about how to use JSONPath with `kubectl`, refer to the documentation at <http://kubernetes.io/docs/user-guide/jsonpath>.

Once you know the IPs of your nodes, you can try accessing your service through them:

```
$ curl http://130.211.97.55:30123  
You've hit kubia-ym8or  
$ curl http://130.211.99.206:30123  
You've hit kubia-xueql
```

**TIP** When using Minikube, you can easily access your NodePort services through your browser by running `minikube service <service-name> [-n <namespace>]`.

As you can see, your pods are now accessible to the whole internet through port 30123 on any of your nodes. It doesn't matter what node a client sends the request to. But if you only point your clients to the first node, when that node fails, your clients can't access the service anymore. That's why it makes sense to put a load balancer in front of the nodes to make sure you're spreading requests across all healthy nodes and never sending them to a node that's offline at that moment.

If your Kubernetes cluster supports it (which is mostly true when Kubernetes is deployed on cloud infrastructure), the load balancer can be provisioned automatically by creating a `LoadBalancer` instead of a `NodePort` service. We'll look at this next.

### 5.3.2 Exposing a service through an external load balancer

Kubernetes clusters running on cloud providers usually support the automatic provision of a load balancer from the cloud infrastructure. All you need to do is set the

service's type to `LoadBalancer` instead of `NodePort`. The load balancer will have its own unique, publicly accessible IP address and will redirect all connections to your service. You can thus access your service through the load balancer's IP address.

If Kubernetes is running in an environment that doesn't support `LoadBalancer` services, the load balancer will not be provisioned, but the service will still behave like a `NodePort` service. That's because a `LoadBalancer` service is an extension of a `NodePort` service. You'll run this example on Google Kubernetes Engine, which supports `LoadBalancer` services. Minikube doesn't, at least not as of this writing.

### CREATING A `LOADBALANCER` SERVICE

To create a service with a load balancer in front, create the service from the following YAML manifest, as shown in the following listing.

**Listing 5.12** A `LoadBalancer`-type service: `kubia-svc-loadbalancer.yaml`

```
apiVersion: v1
kind: Service
metadata:
  name: kubia-loadbalancer
spec:
  type: LoadBalancer
  ports:
    - port: 80
      targetPort: 8080
  selector:
    app: kubia
```

← This type of service obtains a load balancer from the infrastructure hosting the Kubernetes cluster.

The service type is set to `LoadBalancer` instead of `NodePort`. You're not specifying a specific node port, although you could (you're letting Kubernetes choose one instead).

### CONNECTING TO THE SERVICE THROUGH THE LOAD BALANCER

After you create the service, it takes time for the cloud infrastructure to create the load balancer and write its IP address into the Service object. Once it does that, the IP address will be listed as the external IP address of your service:

```
$ kubectl get svc kubia-loadbalancer
```

NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubia-loadbalancer	10.111.241.153	130.211.53.173	80:32143/TCP	1m

In this case, the load balancer is available at IP `130.211.53.173`, so you can now access the service at that IP address:

```
$ curl http://130.211.53.173
You've hit kubia-xueq1
```

Success! As you may have noticed, this time you didn't need to mess with firewalls the way you had to before with the `NodePort` service.

### Session affinity and web browsers

Because your service is now exposed externally, you may try accessing it with your web browser. You'll see something that may strike you as odd—the browser will hit the exact same pod every time. Did the service's session affinity change in the meantime? With `kubectl explain`, you can double-check that the service's session affinity is still set to `None`, so why don't different browser requests hit different pods, as is the case when using `curl`?

Let me explain what's happening. The browser is using keep-alive connections and sends all its requests through a single connection, whereas `curl` opens a new connection every time. Services work at the connection level, so when a connection to a service is first opened, a random pod is selected and then all network packets belonging to that connection are all sent to that single pod. Even if session affinity is set to `None`, users will always hit the same pod (until the connection is closed).

See figure 5.7 to see how HTTP requests are delivered to the pod. External clients (`curl` in your case) connect to port 80 of the load balancer and get routed to the

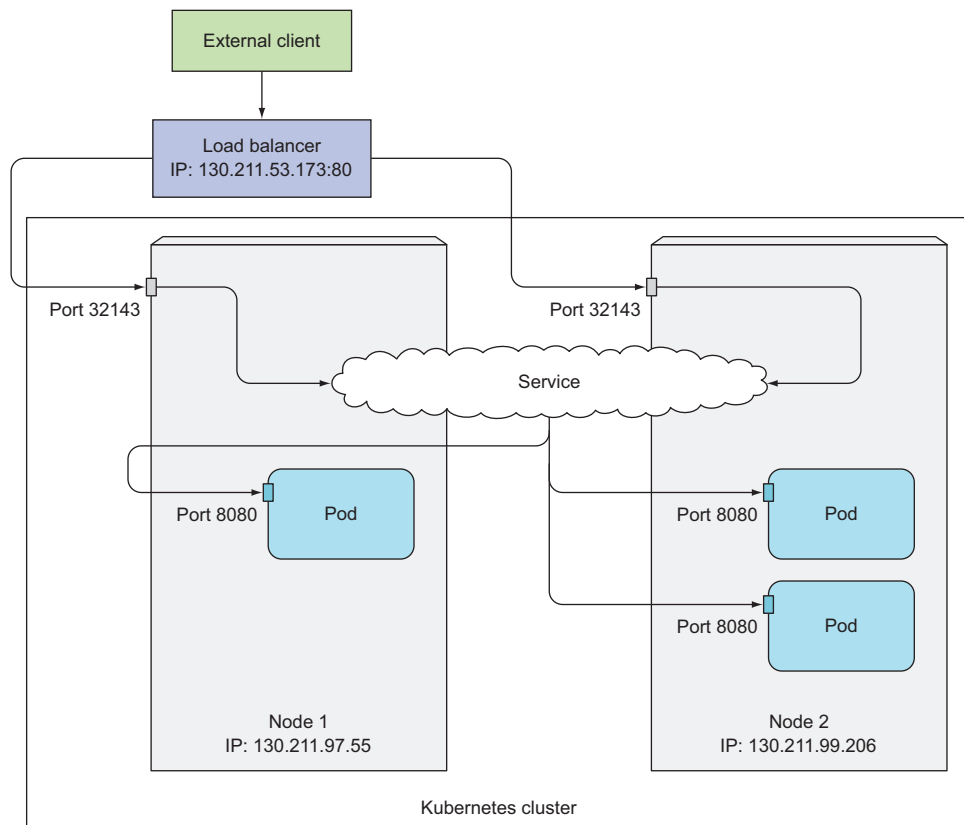


Figure 5.7 An external client connecting to a LoadBalancer service

implicitly assigned node port on one of the nodes. From there, the connection is forwarded to one of the pod instances.

As already mentioned, a `LoadBalancer`-type service is a `NodePort` service with an additional infrastructure-provided load balancer. If you use `kubectl describe` to display additional info about the service, you'll see that a node port has been selected for the service. If you were to open the firewall for this port, the way you did in the previous section about `NodePort` services, you could access the service through the node IPs as well.

**TIP** If you're using Minikube, even though the load balancer will never be provisioned, you can still access the service through the node port (at the Minikube VM's IP address).

### 5.3.3 Understanding the peculiarities of external connections

You must be aware of several things related to externally originating connections to services.

#### UNDERSTANDING AND PREVENTING UNNECESSARY NETWORK HOPS

When an external client connects to a service through the node port (this also includes cases when it goes through the load balancer first), the randomly chosen pod may or may not be running on the same node that received the connection. An additional network hop is required to reach the pod, but this may not always be desirable.

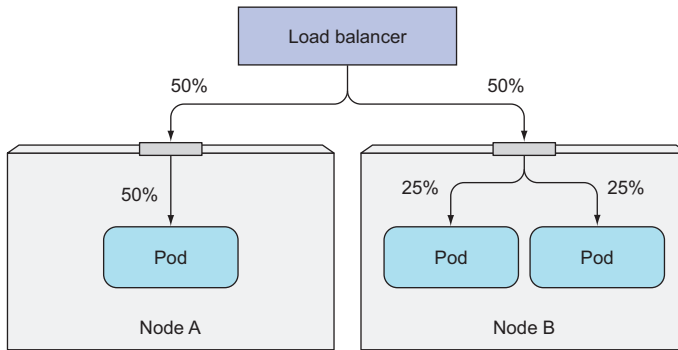
You can prevent this additional hop by configuring the service to redirect external traffic only to pods running on the node that received the connection. This is done by setting the `externalTrafficPolicy` field in the service's spec section:

```
spec:
  externalTrafficPolicy: Local
  ...
```

If a service definition includes this setting and an external connection is opened through the service's node port, the service proxy will choose a locally running pod. If no local pods exist, the connection will hang (it won't be forwarded to a random global pod, the way connections are when not using the annotation). You therefore need to ensure the load balancer forwards connections only to nodes that have at least one such pod.

Using this annotation also has other drawbacks. Normally, connections are spread evenly across all the pods, but when using this annotation, that's no longer the case.

Imagine having two nodes and three pods. Let's say node A runs one pod and node B runs the other two. If the load balancer spreads connections evenly across the two nodes, the pod on node A will receive 50% of all connections, but the two pods on node B will only receive 25% each, as shown in figure 5.8.



**Figure 5.8** A Service using the `Local` external traffic policy may lead to uneven load distribution across pods.

### BEING AWARE OF THE NON-PRESERVATION OF THE CLIENT'S IP

Usually, when clients inside the cluster connect to a service, the pods backing the service can obtain the client's IP address. But when the connection is received through a node port, the packets' source IP is changed, because Source Network Address Translation (SNAT) is performed on the packets.

The backing pod can't see the actual client's IP, which may be a problem for some applications that need to know the client's IP. In the case of a web server, for example, this means the access log won't show the browser's IP.

The `Local` external traffic policy described in the previous section affects the preservation of the client's IP, because there's no additional hop between the node receiving the connection and the node hosting the target pod (SNAT isn't performed).

## 5.4 Exposing services externally through an Ingress resource

You've now seen two ways of exposing a service to clients outside the cluster, but another method exists—creating an Ingress resource.

**DEFINITION** *Ingress* (noun)—The act of going in or entering; the right to enter; a means or place of entering; entryway.

Let me first explain why you need another way to access Kubernetes services from the outside.

### UNDERSTANDING WHY INGRESSES ARE NEEDED

One important reason is that each `LoadBalancer` service requires its own load balancer with its own public IP address, whereas an Ingress only requires one, even when providing access to dozens of services. When a client sends an HTTP request to the Ingress, the host and path in the request determine which service the request is forwarded to, as shown in figure 5.9.

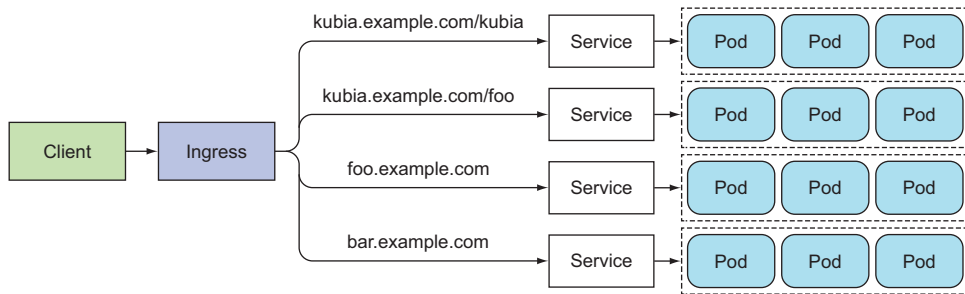


Figure 5.9 Multiple services can be exposed through a single Ingress.

Ingresses operate at the application layer of the network stack (HTTP) and can provide features such as cookie-based session affinity and the like, which services can't.

#### UNDERSTANDING THAT AN INGRESS CONTROLLER IS REQUIRED

Before we go into the features an Ingress object provides, let me emphasize that to make Ingress resources work, an Ingress controller needs to be running in the cluster. Different Kubernetes environments use different implementations of the controller, but several don't provide a default controller at all.

For example, Google Kubernetes Engine uses Google Cloud Platform's own HTTP load-balancing features to provide the Ingress functionality. Initially, Minikube didn't provide a controller out of the box, but it now includes an add-on that can be enabled to let you try out the Ingress functionality. Follow the instructions in the following sidebar to ensure it's enabled.

#### Enabling the Ingress add-on in Minikube

If you're using Minikube to run the examples in this book, you'll need to ensure the Ingress add-on is enabled. You can check whether it is by listing all the add-ons:

```

$ minikube addons list
- default-storageclass: enabled
- kube-dns: enabled
- heapster: disabled
- ingress: disabled
- registry-creds: disabled
- addon-manager: enabled
- dashboard: enabled
  
```

← The Ingress add-on isn't enabled.

You'll learn about what these add-ons are throughout the book, but it should be pretty clear what the dashboard and the kube-dns add-ons do. Enable the Ingress add-on so you can see Ingresses in action:

```

$ minikube addons enable ingress
ingress was successfully enabled
  
```

*(continued)*

This should have spun up an Ingress controller as another pod. Most likely, the controller pod will be in the `kube-system` namespace, but not necessarily, so list all the running pods across all namespaces by using the `--all-namespaces` option:

```
$ kubectl get po --all-namespaces
```

NAMESPACE	NAME	READY	STATUS	RESTARTS	AGE
default	kubia-rsv5m	1/1	Running	0	13h
default	kubia-fe4ad	1/1	Running	0	13h
default	kubia-ke823	1/1	Running	0	13h
kube-system	default-http-backend-5wb0h	1/1	Running	0	18m
kube-system	kube-addon-manager-minikube	1/1	Running	3	6d
kube-system	kube-dns-v20-101vq	3/3	Running	9	6d
kube-system	kubernetes-dashboard-jxd9l	1/1	Running	3	6d
kube-system	nginx-ingress-controller-gdts0	1/1	Running	0	18m

At the bottom of the output, you see the Ingress controller pod. The name suggests that Nginx (an open-source HTTP server and reverse proxy) is used to provide the Ingress functionality.

**TIP** The `--all-namespaces` option mentioned in the sidebar is handy when you don't know what namespace your pod (or other type of resource) is in, or if you want to list resources across all namespaces.

### 5.4.1 Creating an Ingress resource

You've confirmed there's an Ingress controller running in your cluster, so you can now create an Ingress resource. The following listing shows what the YAML manifest for the Ingress looks like.

**Listing 5.13** An Ingress resource definition: `kubia-ingress.yaml`

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

This Ingress maps the **kubia.example.com** domain name to your service.

All requests will be sent to port 80 of the kubia-nodeport service.

This defines an Ingress with a single rule, which makes sure all HTTP requests received by the Ingress controller, in which the host `kubia.example.com` is requested, will be sent to the `kubia-nodeport` service on port 80.



**NOTE** Ingress controllers on cloud providers (in GKE, for example) require the Ingress to point to a NodePort service. But that's not a requirement of Kubernetes itself.

### 5.4.2 Accessing the service through the Ingress

To access your service through <http://kubia.example.com>, you'll need to make sure the domain name resolves to the IP of the Ingress controller.

#### OBTAINING THE IP ADDRESS OF THE INGRESS

To look up the IP, you need to list Ingresses:

```
$ kubectl get ingresses
```

NAME	HOSTS	ADDRESS	PORTS	AGE
kubia	kubia.example.com	192.168.99.100	80	29m

**NOTE** When running on cloud providers, the address may take time to appear, because the Ingress controller provisions a load balancer behind the scenes.

The IP is shown in the ADDRESS column.

#### ENSURING THE HOST CONFIGURED IN THE INGRESS POINTS TO THE INGRESS' IP ADDRESS

Once you know the IP, you can then either configure your DNS servers to resolve [kubia.example.com](http://kubia.example.com) to that IP or you can add the following line to `/etc/hosts` (or `C:\windows\system32\drivers\etc\hosts` on Windows):

```
192.168.99.100    kubia.example.com
```

#### ACCESSING PODS THROUGH THE INGRESS

Everything is now set up, so you can access the service at <http://kubia.example.com> (using a browser or curl):

```
$ curl http://kubia.example.com
You've hit kubia-ke823
```

You've successfully accessed the service through an Ingress. Let's take a better look at how that unfolded.

#### UNDERSTANDING HOW INGRESSES WORK

Figure 5.10 shows how the client connected to one of the pods through the Ingress controller. The client first performed a DNS lookup of [kubia.example.com](http://kubia.example.com), and the DNS server (or the local operating system) returned the IP of the Ingress controller. The client then sent an HTTP request to the Ingress controller and specified [kubia.example.com](http://kubia.example.com) in the Host header. From that header, the controller determined which service the client is trying to access, looked up the pod IPs through the Endpoints object associated with the service, and forwarded the client's request to one of the pods.

As you can see, the Ingress controller didn't forward the request to the service. It only used it to select a pod. Most, if not all, controllers work like this.

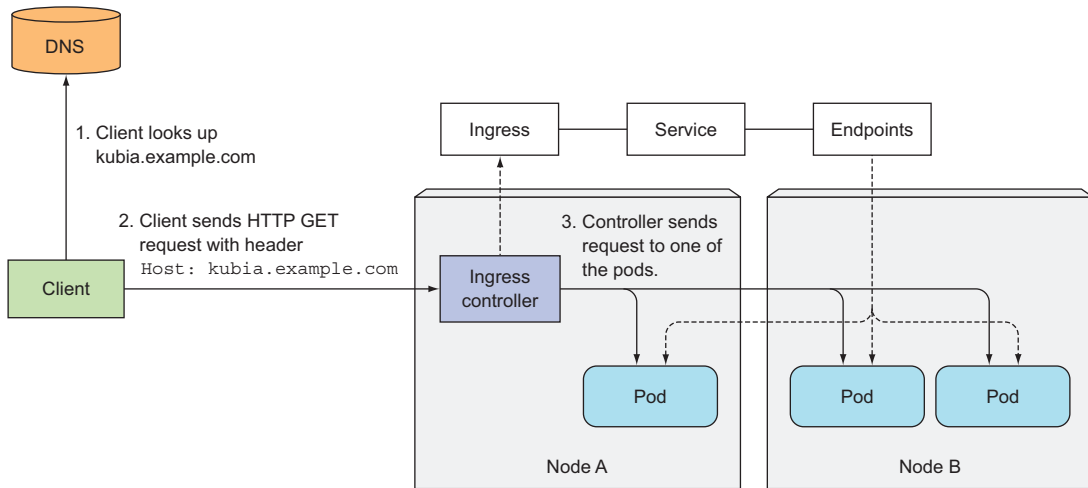


Figure 5.10 Accessing pods through an Ingress

### 5.4.3 Exposing multiple services through the same Ingress

If you look at the Ingress spec closely, you'll see that both `rules` and `paths` are arrays, so they can contain multiple items. An Ingress can map multiple hosts and paths to multiple services, as you'll see next. Let's focus on paths first.

#### MAPPING DIFFERENT SERVICES TO DIFFERENT PATHS OF THE SAME HOST

You can map multiple paths on the same host to different services, as shown in the following listing.

#### Listing 5.14 Ingress exposing multiple services on same host, but different paths

```
...
- host: kubia.example.com
  http:
    paths:
      - path: /kubia
        backend:
          serviceName: kubia
          servicePort: 80
      - path: /foo
        backend:
          serviceName: bar
          servicePort: 80
```

Requests to `kubia.example.com/kubia` will be routed to the `kubia` service.

Requests to `kubia.example.com/bar` will be routed to the `bar` service.

In this case, requests will be sent to two different services, depending on the path in the requested URL. Clients can therefore reach two different services through a single IP address (that of the Ingress controller).

**MAPPING DIFFERENT SERVICES TO DIFFERENT HOSTS**

Similarly, you can use an Ingress to map to different services based on the host in the HTTP request instead of (only) the path, as shown in the next listing.

**Listing 5.15** Ingress exposing multiple services on different hosts

```
spec:
  rules:
    - host: foo.example.com
      http:
        paths:
          - path: /
            backend:
              serviceName: foo
              servicePort: 80
    - host: bar.example.com
      http:
        paths:
          - path: /
            backend:
              serviceName: bar
              servicePort: 80
```

Requests for **foo.example.com** will be routed to service foo.

Requests for **bar.example.com** will be routed to service bar.

Requests received by the controller will be forwarded to either service `foo` or `bar`, depending on the `Host` header in the request (the way virtual hosts are handled in web servers). DNS needs to point both the `foo.example.com` and the `bar.example.com` domain names to the Ingress controller's IP address.

#### 5.4.4 Configuring Ingress to handle TLS traffic

You've seen how an Ingress forwards HTTP traffic. But what about HTTPS? Let's take a quick look at how to configure Ingress to support TLS.

**CREATING A TLS CERTIFICATE FOR THE INGRESS**

When a client opens a TLS connection to an Ingress controller, the controller terminates the TLS connection. The communication between the client and the controller is encrypted, whereas the communication between the controller and the backend pod isn't. The application running in the pod doesn't need to support TLS. For example, if the pod runs a web server, it can accept only HTTP traffic and let the Ingress controller take care of everything related to TLS. To enable the controller to do that, you need to attach a certificate and a private key to the Ingress. The two need to be stored in a Kubernetes resource called a `Secret`, which is then referenced in the Ingress manifest. We'll explain `Secrets` in detail in chapter 7. For now, you'll create the `Secret` without paying too much attention to it.

First, you need to create the private key and certificate:

```
$ openssl genrsa -out tls.key 2048
$ openssl req -new -x509 -key tls.key -out tls.cert -days 360 -subj
➡ /CN=kubia.example.com
```

Then you create the Secret from the two files like this:

```
$ kubectl create secret tls tls-secret --cert=tls.cert --key=tls.key
secret "tls-secret" created
```

### Signing certificates through the CertificateSigningRequest resource

Instead of signing the certificate ourselves, you can get the certificate signed by creating a CertificateSigningRequest (CSR) resource. Users or their applications can create a regular certificate request, put it into a CSR, and then either a human operator or an automated process can approve the request like this:

```
$ kubectl certificate approve <name of the CSR>
```

The signed certificate can then be retrieved from the CSR's `status.certificate` field.

Note that a certificate signer component must be running in the cluster; otherwise creating CertificateSigningRequest and approving or denying them won't have any effect.

The private key and the certificate are now stored in the Secret called `tls-secret`. Now, you can update your Ingress object so it will also accept HTTPS requests for [kubia.example.com](http://kubia.example.com). The Ingress manifest should now look like the following listing.

#### Listing 5.16 Ingress handling TLS traffic: `kubia-ingress-tls.yaml`

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: kubia
spec:
  tls:
    - hosts:
      - kubia.example.com
      secretName: tls-secret
  rules:
    - host: kubia.example.com
      http:
        paths:
          - path: /
            backend:
              serviceName: kubia-nodeport
              servicePort: 80
```

The whole TLS configuration  
is under this attribute.

TLS connections will be accepted for  
the `kubia.example.com` hostname.

The private key and the certificate  
should be obtained from the `tls-`  
`secret` you created previously.

**TIP** Instead of deleting the Ingress and re-creating it from the new file, you can invoke `kubectl apply -f kubia-ingress-tls.yaml`, which updates the Ingress resource with what's specified in the file.

You can now use HTTPS to access your service through the Ingress:

```
$ curl -k -v https://kubia.example.com/kubia
* About to connect() to kubia.example.com port 443 (#0)
...
* Server certificate:
*   subject: CN=kubia.example.com
...
> GET /kubia HTTP/1.1
> ...
You've hit kubia-xueq1
```

The command's output shows the response from the app, as well as the server certificate you configured the Ingress with.

**NOTE** Support for Ingress features varies between the different Ingress controller implementations, so check the implementation-specific documentation to see what's supported.

Ingresses are a relatively new Kubernetes feature, so you can expect to see many improvements and new features in the future. Although they currently support only L7 (HTTP/HTTPS) load balancing, support for L4 load balancing is also planned.

## 5.5 Signaling when a pod is ready to accept connections

There's one more thing we need to cover regarding both Services and Ingresses. You've already learned that pods are included as endpoints of a service if their labels match the service's pod selector. As soon as a new pod with proper labels is created, it becomes part of the service and requests start to be redirected to the pod. But what if the pod isn't ready to start serving requests immediately?

The pod may need time to load either configuration or data, or it may need to perform a warm-up procedure to prevent the first user request from taking too long and affecting the user experience. In such cases you don't want the pod to start receiving requests immediately, especially when the already-running instances can process requests properly and quickly. It makes sense to not forward requests to a pod that's in the process of starting up until it's fully ready.

### 5.5.1 Introducing readiness probes

In the previous chapter you learned about liveness probes and how they help keep your apps healthy by ensuring unhealthy containers are restarted automatically. Similar to liveness probes, Kubernetes allows you to also define a readiness probe for your pod.

The readiness probe is invoked periodically and determines whether the specific pod should receive client requests or not. When a container's readiness probe returns success, it's signaling that the container is ready to accept requests.

This notion of being ready is obviously something that's specific to each container. Kubernetes can merely check if the app running in the container responds to a simple

GET / request or it can hit a specific URL path, which causes the app to perform a whole list of checks to determine if it's ready. Such a detailed readiness probe, which takes the app's specifics into account, is the app developer's responsibility.

#### TYPES OF READINESS PROBES

Like liveness probes, three types of readiness probes exist:

- An *Exec* probe, where a process is executed. The container's status is determined by the process' exit status code.
- An *HTTP GET* probe, which sends an HTTP GET request to the container and the HTTP status code of the response determines whether the container is ready or not.
- A *TCP Socket* probe, which opens a TCP connection to a specified port of the container. If the connection is established, the container is considered ready.

#### UNDERSTANDING THE OPERATION OF READINESS PROBES

When a container is started, Kubernetes can be configured to wait for a configurable amount of time to pass before performing the first readiness check. After that, it invokes the probe periodically and acts based on the result of the readiness probe. If a pod reports that it's not ready, it's removed from the service. If the pod then becomes ready again, it's re-added.

Unlike liveness probes, if a container fails the readiness check, it won't be killed or restarted. This is an important distinction between liveness and readiness probes. Liveness probes keep pods healthy by killing off unhealthy containers and replacing them with new, healthy ones, whereas readiness probes make sure that only pods that are ready to serve requests receive them. This is mostly necessary during container start up, but it's also useful after the container has been running for a while.

As you can see in figure 5.11, if a pod's readiness probe fails, the pod is removed from the Endpoints object. Clients connecting to the service will not be redirected to the pod. The effect is the same as when the pod doesn't match the service's label selector at all.

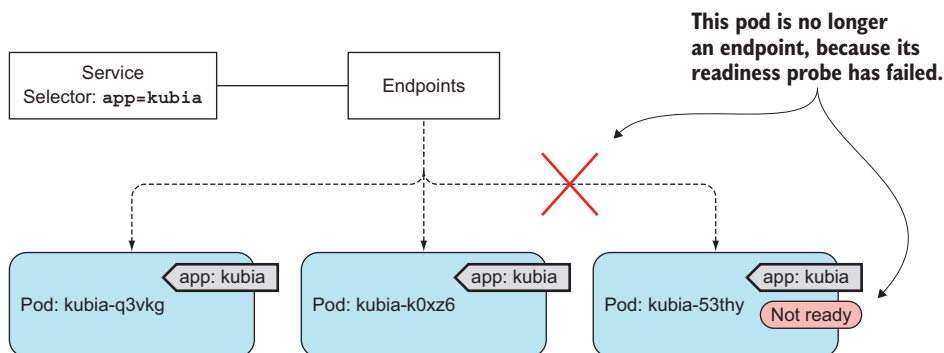


Figure 5.11 A pod whose readiness probe fails is removed as an endpoint of a service.

**UNDERSTANDING WHY READINESS PROBES ARE IMPORTANT**

Imagine that a group of pods (for example, pods running application servers) depends on a service provided by another pod (a backend database, for example). If at any point one of the frontend pods experiences connectivity problems and can't reach the database anymore, it may be wise for its readiness probe to signal to Kubernetes that the pod isn't ready to serve any requests at that time. If other pod instances aren't experiencing the same type of connectivity issues, they can serve requests normally. A readiness probe makes sure clients only talk to those healthy pods and never notice there's anything wrong with the system.

**5.5.2 Adding a readiness probe to a pod**

Next you'll add a readiness probe to your existing pods by modifying the ReplicationController's pod template.

**ADDING A READINESS PROBE TO THE POD TEMPLATE**

You'll use the `kubect1 edit rc kubil` command to add the probe to the pod template in your existing ReplicationController:

```
$ kubect1 edit rc kubil
```

When the ReplicationController's YAML opens in the text editor, find the container specification in the pod template and add the following readiness probe definition to the first container under `spec.template.spec.containers`. The YAML should look like the following listing.

**Listing 5.17 RC creating a pod with a readiness probe: kubil-rc-readinessprobe.yaml**

```
apiVersion: v1
kind: ReplicationController
...
spec:
  ...
  template:
    ...
    spec:
      containers:
      - name: kubil
        image: luksa/kubil
        readinessProbe:
          exec:
            command:
            - ls
            - /var/ready
        ...
```

**A readinessProbe may be defined for each container in the pod.**

The readiness probe will periodically perform the command `ls /var/ready` inside the container. The `ls` command returns exit code zero if the file exists, or a non-zero exit code otherwise. If the file exists, the readiness probe will succeed; otherwise, it will fail.

The reason you’re defining such a strange readiness probe is so you can toggle its result by creating or removing the file in question. The file doesn’t exist yet, so all the pods should now report not being ready, right? Well, not exactly. As you may remember from the previous chapter, changing a ReplicationController’s pod template has no effect on existing pods.

In other words, all your existing pods still have no readiness probe defined. You can see this by listing the pods with `kubectl get pods` and looking at the `READY` column. You need to delete the pods and have them re-created by the Replication-Controller. The new pods will fail the readiness check and won’t be included as endpoints of the service until you create the `/var/ready` file in each of them.

#### OBSERVING AND MODIFYING THE PODS’ READINESS STATUS

List the pods again and inspect whether they’re ready or not:

```
$ kubectl get po
NAME          READY    STATUS    RESTARTS   AGE
kubia-2r1qb   0/1      Running   0           1m
kubia-3rax1    0/1      Running   0           1m
kubia-3yw4s    0/1      Running   0           1m
```

The `READY` column shows that none of the containers are ready. Now make the readiness probe of one of them start returning success by creating the `/var/ready` file, whose existence makes your mock readiness probe succeed:

```
$ kubectl exec kubia-2r1qb -- touch /var/ready
```

You’ve used the `kubectl exec` command to execute the `touch` command inside the container of the `kubia-2r1qb` pod. The `touch` command creates the file if it doesn’t yet exist. The pod’s readiness probe command should now exit with status code 0, which means the probe is successful, and the pod should now be shown as ready. Let’s see if it is:

```
$ kubectl get po kubia-2r1qb
NAME          READY    STATUS    RESTARTS   AGE
kubia-2r1qb   0/1      Running   0           2m
```

The pod still isn’t ready. Is there something wrong or is this the expected result? Take a more detailed look at the pod with `kubectl describe`. The output should contain the following line:

```
Readiness: exec [ls /var/ready] delay=0s timeout=1s period=10s #success=1
➡ #failure=3
```

The readiness probe is checked periodically—every 10 seconds by default. The pod isn’t ready because the readiness probe hasn’t been invoked yet. But in 10 seconds at the latest, the pod should become ready and its IP should be listed as the only endpoint of the service (run `kubectl get endpoints kubia-loadbalancer` to confirm).



### HITTING THE SERVICE WITH THE SINGLE READY POD

You can now hit the service URL a few times to see that each and every request is redirected to this one pod:

```
$ curl http://130.211.53.173
You've hit kubia-2r1qb
$ curl http://130.211.53.173
You've hit kubia-2r1qb
...
$ curl http://130.211.53.173
You've hit kubia-2r1qb
```

Even though there are three pods running, only a single pod is reporting as being ready and is therefore the only pod receiving requests. If you now delete the file, the pod will be removed from the service again.

### 5.5.3 Understanding what real-world readiness probes should do

This mock readiness probe is useful only for demonstrating what readiness probes do. In the real world, the readiness probe should return success or failure depending on whether the app can (and wants to) receive client requests or not.

Manually removing pods from services should be performed by either deleting the pod or changing the pod's labels instead of manually flipping a switch in the probe.

**TIP** If you want to add or remove a pod from a service manually, add `enabled=true` as a label to your pod and to the label selector of your service. Remove the label when you want to remove the pod from the service.

#### ALWAYS DEFINE A READINESS PROBE

Before we conclude this section, there are two final notes about readiness probes that I need to emphasize. First, if you don't add a readiness probe to your pods, they'll become service endpoints almost immediately. If your application takes too long to start listening for incoming connections, client requests hitting the service will be forwarded to the pod while it's still starting up and not ready to accept incoming connections. Clients will therefore see "Connection refused" types of errors.

**TIP** You should always define a readiness probe, even if it's as simple as sending an HTTP request to the base URL.

#### DON'T INCLUDE POD SHUTDOWN LOGIC INTO YOUR READINESS PROBES

The other thing I need to mention applies to the other end of the pod's life (pod shutdown) and is also related to clients experiencing connection errors.

When a pod is being shut down, the app running in it usually stops accepting connections as soon as it receives the termination signal. Because of this, you might think you need to make your readiness probe start failing as soon as the shutdown procedure is initiated, ensuring the pod is removed from all services it's part of. But that's not necessary, because Kubernetes removes the pod from all services as soon as you delete the pod.

## 5.6 Using a headless service for discovering individual pods

You’ve seen how services can be used to provide a stable IP address allowing clients to connect to pods (or other endpoints) backing each service. Each connection to the service is forwarded to one randomly selected backing pod. But what if the client needs to connect to all of those pods? What if the backing pods themselves need to each connect to all the other backing pods? Connecting through the service clearly isn’t the way to do this. What is?

For a client to connect to all pods, it needs to figure out the IP of each individual pod. One option is to have the client call the Kubernetes API server and get the list of pods and their IP addresses through an API call, but because you should always strive to keep your apps Kubernetes-agnostic, using the API server isn’t ideal.

Luckily, Kubernetes allows clients to discover pod IPs through DNS lookups. Usually, when you perform a DNS lookup for a service, the DNS server returns a single IP—the service’s cluster IP. But if you tell Kubernetes you don’t need a cluster IP for your service (you do this by setting the `clusterIP` field to `None` in the service specification), the DNS server will return the pod IPs instead of the single service IP.

Instead of returning a single DNS A record, the DNS server will return multiple A records for the service, each pointing to the IP of an individual pod backing the service at that moment. Clients can therefore do a simple DNS A record lookup and get the IPs of all the pods that are part of the service. The client can then use that information to connect to one, many, or all of them.

### 5.6.1 Creating a headless service

Setting the `clusterIP` field in a service spec to `None` makes the service *headless*, as Kubernetes won’t assign it a cluster IP through which clients could connect to the pods backing it.

You’ll create a headless service called `kubia-headless` now. The following listing shows its definition.

**Listing 5.18** A headless service: `kubia-svc-headless.yaml`

```
apiVersion: v1
kind: Service
metadata:
  name: kubia-headless
spec:
  clusterIP: None
  ports:
    - port: 80
      targetPort: 8080
  selector:
    app: kubia
```

← This makes the service headless.

After you create the service with `kubectl create`, you can inspect it with `kubectl get` and `kubectl describe`. You’ll see it has no cluster IP and its endpoints include (part of)

the pods matching its pod selector. I say “part of” because your pods contain a readiness probe, so only pods that are ready will be listed as endpoints of the service. Before continuing, please make sure at least two pods report being ready, by creating the `/var/ready` file, as in the previous example:

```
$ kubectl exec <pod name> -- touch /var/ready
```

## 5.6.2 Discovering pods through DNS

With your pods ready, you can now try performing a DNS lookup to see if you get the actual pod IPs or not. You’ll need to perform the lookup from inside one of the pods. Unfortunately, your `kubia` container image doesn’t include the `nslookup` (or the `dig`) binary, so you can’t use it to perform the DNS lookup.

All you’re trying to do is perform a DNS lookup from inside a pod running in the cluster. Why not run a new pod based on an image that contains the binaries you need? To perform DNS-related actions, you can use the `tutum/dnsutils` container image, which is available on Docker Hub and contains both the `nslookup` and the `dig` binaries. To run the pod, you can go through the whole process of creating a YAML manifest for it and passing it to `kubectl create`, but that’s too much work, right? Luckily, there’s a faster way.

### RUNNING A POD WITHOUT WRITING A YAML MANIFEST

In chapter 1, you already created pods without writing a YAML manifest by using the `kubectl run` command. But this time you want to create only a pod—you don’t need to create a `ReplicationController` to manage the pod. You can do that like this:

```
$ kubectl run dnsutils --image=tutum/dnsutils --generator=run-pod/v1
➡ --command -- sleep infinity
pod "dnsutils" created
```

The trick is in the `--generator=run-pod/v1` option, which tells `kubectl` to create the pod directly, without any kind of `ReplicationController` or similar behind it.

### UNDERSTANDING DNS A RECORDS RETURNED FOR A HEADLESS SERVICE

Let’s use the newly created pod to perform a DNS lookup:

```
$ kubectl exec dnsutils nslookup kubia-headless
...
Name:      kubia-headless.default.svc.cluster.local
Address: 10.108.1.4
Name:      kubia-headless.default.svc.cluster.local
Address: 10.108.2.5
```

The DNS server returns two different IPs for the `kubia-headless.default.svc.cluster.local` FQDN. Those are the IPs of the two pods that are reporting being ready. You can confirm this by listing pods with `kubectl get pods -o wide`, which shows the pods’ IPs.

This is different from what DNS returns for regular (non-headless) services, such as for your `kubia` service, where the returned IP is the service's cluster IP:

```
$ kubectl exec dnsutils nslookup kubia
...
Name:      kubia.default.svc.cluster.local
Address: 10.111.249.153
```

Although headless services may seem different from regular services, they aren't that different from the clients' perspective. Even with a headless service, clients can connect to its pods by connecting to the service's DNS name, as they can with regular services. But with headless services, because DNS returns the pods' IPs, clients connect directly to the pods, instead of through the service proxy.

**NOTE** A headless services still provides load balancing across pods, but through the DNS round-robin mechanism instead of through the service proxy.

### 5.6.3 *Discovering all pods—even those that aren't ready*

You've seen that only pods that are ready become endpoints of services. But sometimes you want to use the service discovery mechanism to find all pods matching the service's label selector, even those that aren't ready.

Luckily, you don't have to resort to querying the Kubernetes API server. You can use the DNS lookup mechanism to find even those unready pods. To tell Kubernetes you want all pods added to a service, regardless of the pod's readiness status, you must add the following annotation to the service:

```
kind: Service
metadata:
  annotations:
    service.alpha.kubernetes.io/tolerate-unready-endpoints: "true"
```

**WARNING** As the annotation name suggests, as I'm writing this, this is an alpha feature. The Kubernetes Service API already supports a new service spec field called `publishNotReadyAddresses`, which will replace the `tolerate-unready-endpoints` annotation. In Kubernetes version 1.9.0, the field is not honored yet (the annotation is what determines whether unready endpoints are included in the DNS or not). Check the documentation to see whether that's changed.

## 5.7 *Troubleshooting services*

Services are a crucial Kubernetes concept and the source of frustration for many developers. I've seen many developers lose heaps of time figuring out why they can't connect to their pods through the service IP or FQDN. For this reason, a short look at how to troubleshoot services is in order.

When you're unable to access your pods through the service, you should start by going through the following list:

- First, make sure you're connecting to the service's cluster IP from within the cluster, not from the outside.
- Don't bother pinging the service IP to figure out if the service is accessible (remember, the service's cluster IP is a virtual IP and pinging it will never work).
- If you've defined a readiness probe, make sure it's succeeding; otherwise the pod won't be part of the service.
- To confirm that a pod is part of the service, examine the corresponding Endpoints object with `kubectl get endpoints`.
- If you're trying to access the service through its FQDN or a part of it (for example, `myservice.mynamespace.svc.cluster.local` or `myservice.mynamespace`) and it doesn't work, see if you can access it using its cluster IP instead of the FQDN.
- Check whether you're connecting to the port exposed by the service and not the target port.
- Try connecting to the pod IP directly to confirm your pod is accepting connections on the correct port.
- If you can't even access your app through the pod's IP, make sure your app isn't only binding to localhost.

This should help you resolve most of your service-related problems. You'll learn much more about how services work in chapter 11. By understanding exactly how they're implemented, it should be much easier for you to troubleshoot them.

## 5.8 Summary

In this chapter, you've learned how to create Kubernetes Service resources to expose the services available in your application, regardless of how many pod instances are providing each service. You've learned how Kubernetes

- Exposes multiple pods that match a certain label selector under a single, stable IP address and port
- Makes services accessible from inside the cluster by default, but allows you to make the service accessible from outside the cluster by setting its type to either `NodePort` or `LoadBalancer`
- Enables pods to discover services together with their IP addresses and ports by looking up environment variables
- Allows discovery of and communication with services residing outside the cluster by creating a Service resource without specifying a selector, by creating an associated Endpoints resource instead
- Provides a DNS CNAME alias for external services with the `ExternalName` service type
- Exposes multiple HTTP services through a single Ingress (consuming a single IP)

- Uses a pod container's readiness probe to determine whether a pod should or shouldn't be included as a service endpoint
- Enables discovery of pod IPs through DNS when you create a headless service

Along with getting a better understanding of services, you've also learned how to

- Troubleshoot them
- Modify firewall rules in Google Kubernetes/Compute Engine
- Execute commands in pod containers through `kubectl exec`
- Run a bash shell in an existing pod's container
- Modify Kubernetes resources through the `kubectl apply` command
- Run an unmanaged ad hoc pod with `kubectl run --generator=run-pod/v1`