OpenShift IN ACTION

Jamie Duncan
John Osborne
Foreword by Jim Whitehurst

SAMPLE CHAPTER

MANNING

OpenShift in Action

by Jamie Duncan
and John Osborne

Chapter 8

brief contents

8 Stateful applications

This chapter covers

- Enabling a headless service
- Clustering applications
- Configuring sticky sessions
- Handling graceful shutdowns
- Working with stateful sets

In chapter 7, you created persistent storage for the Image Uploader pods, which allowed data to persist past the lifecycle of a single pod. When a pod failed, a new pod spun up in its place and mounted the existing persistent volume locally. Persistent storage in OpenShift allows many stateful applications to run in containers. Many other stateful applications still have requirements that are unsatisfied by persistent storage alone: for instance, many workloads distribute data through replication, which requires application-level clustering. In OpenShift, this type of data replication requires direct pod-to-pod networking without going through the service layer. It is also very common for stateful applications such as databases to have their own custom load balancing and discovery algorithms which require direct pod-to-pod access. Other common requirements for stateful applications include the ability to support sticky sessions as well as implement a predictable graceful shutdown.

One of the main goals of the OpenShift container platform is to be a world-class platform for stateless and stateful applications. In order to support stateful applications, a variety of tools are available to make virtually any application container-native. This chapter will walk you through the most popular tools, including headless services, sticky sessions, pod-discovery techniques, and stateful sets, just to name a few. At the end of the chapter, you'll walk through the power of the stateful set, which brings many stateful applications to life on OpenShift.

8.1 Enabling a headless service

A good example of application clustering in everyday life is demonstrated by Amazon's virtual shopping cart. Amazon customers browse for items and add them to a virtual shopping cart so they can potentially be purchased later. If an Amazon user is signed in to their account, their virtual shopping cart will be persisted permanently because the data is stored in a database. But for users who aren't signed in to an account, the shopping cart is temporary. The temporary cart is implemented as in-memory cache in Amazon's datacenter. By taking advantage of in-memory caching, end users get fast performance, which results in a better user experience. One downside of using in-memory caching is that if a server crashes, the data is lost. A common solution to this problem is data replication: when an application puts data in memory, that data can be replicated to many different caches, which results in fast performance and redundancy.

Before applications can replicate data among one another, they need a way to dynamically find each other. In chapter 6, this concept was covered through the use of service discovery, in which pods use an OpenShift service object. The OpenShift service object provides a stable IP and port that can be used to access one or more pods running the same workload. For most use cases, having a stable IP and port to access one or more replicated pods is all that's required. But many types of applications, such as those that replicate data, require the ability to find all the pods in a service and access each one directly on demand.

One working solution would be to use a single service object for each pod, giving the application a stable IP and port for each pod. Although this works nicely, it isn't ideal because it can generate many service objects, which can become difficult to manage. A better solution is to implement a *headless service* and discover the application pods using an application-specific discovery mechanism. A headless service is a service object that doesn't load-balance or proxy between backend pods. It's implemented by setting the `spec.clusterIP` field to `None` in the service API object.

Headless services are most often used for applications that need to access specific pods directly without going through the service proxy. Two common examples of headless services are clustered databases and applications that have client-side load-balancing logic built-in to the code. Later in this chapter, we'll explore an example of a headless service using MongoDB, a popular NoSQL database.

TIP One common traditional approach to discovery has been to use network broadcasting or multicasting, which is blocked by most public cloud providers such as Amazon Web Services (AWS) and Azure. OpenShift also blocks multicasting by default. Fortunately, OpenShift 3.6 and higher allow users to enable multicasting between pods. Because OpenShift tunnels this traffic over its software-defined network (SDN), this solution can also work on all public cloud providers. You can learn more at http://mng.bz/L33O.

8.1.1 Application clustering with WildFly

In this section, you'll deploy a classic example of application-level clustering in OpenShift using WildFly, a popular application server for Java-based application runtimes. You'll be deploying new applications as part of this chapter, so create a new stateful-apps project as follows:

```
oc new-project stateful-apps
```

It's important to note that this new example uses *cookies* stored in your browser to track your session. Cookies are small pieces of data that servers ask your browser to hold to make your experience better. In this case, a cookie will be stored in your browser with a simple unique identifier: a randomly generated string called `JSESSIONID`. When the user initially accesses the web application, the server will reply with a cookie containing the `JSESSIONID` field and a unique identifier as the value. Subsequent access to the application will use `JSESSIONID` to look up all information about the user's session, which is stored in a replication cache. It doesn't matter which pod is accessed—the user experience will be the same (see figure 8.1).

The WildFly application that you'll deploy will replicate the user data among all pods in its service. The application will track which user the request comes from by checking the `JSESSIONID` that's passed from the browser cookie. Because the user data will be replicated, the end user will have a consistent experience even if some pods die and new pods are accessed. Run the following command to install the WildFly application template and see this in action:

```
oc create -f \
    https://raw.githubusercontent.com/OpenShiftInAction/chapter8/master/
➥ wildfly-template.yaml \
    -n stateful-apps
```

To process the template through the OpenShift console, navigate to the stateful-apps project. Click Add to Project, and enter `wildfly-oia-s2i` in the Service Catalog search box. Keep the default values, and click create to process the template.

The build may take up to a couple of minutes because it's pulling in a lot of dependencies. Watch for the pod to be running and ready before you proceed. When the pod is running and ready, choose Applications > Routes to see the route you just created. Click that route: you'll see the application shown in figure 8.2. If you don't see the application, go back and look at the pod logs to make sure the application is fully deployed.

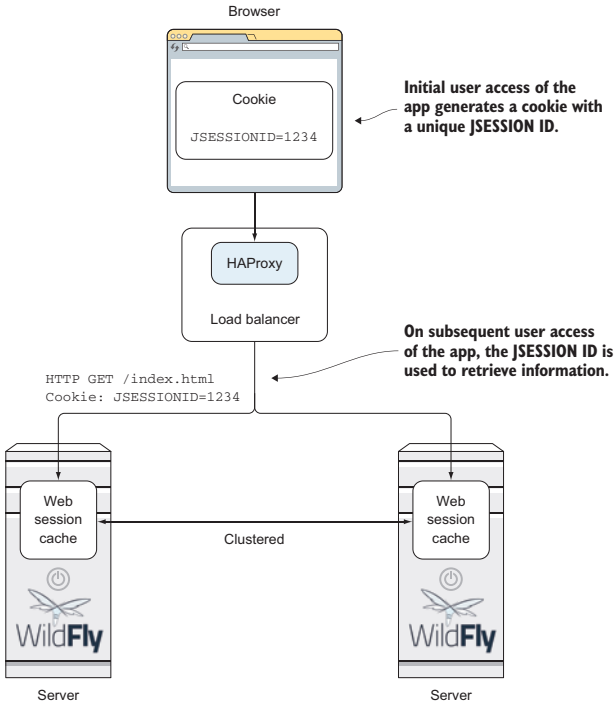


Figure 8.1 An application replicating cached user data

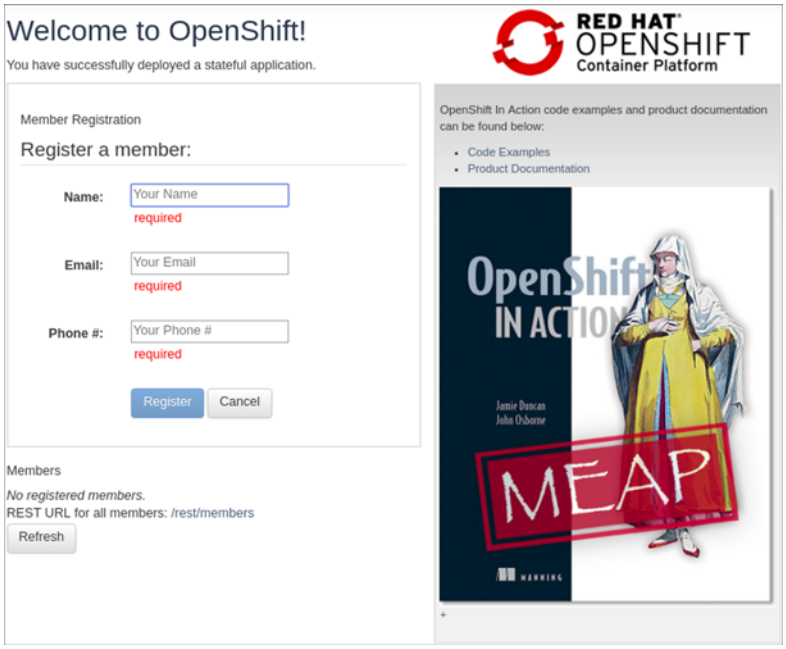


Figure 8.2 Home page for the WildFly application

NOTE Back in chapter 4, you learned that readiness probes can be used to ensure that a pod is ready to receive traffic before traffic is routed to it. The default readiness probe for the WildFly image only makes sure the application server is running as expected—it doesn't check for the application you deployed. The best practice is to create a readiness probe that's specific to each application. Because you haven't done that yet in this exercise, it's possible for traffic to be routed to the pod before the application is fully deployed.

Now that the application is deployed, let's explore application clustering with WildFly on OpenShift. To demonstrate this, you'll do the following:

1. Add data to your session by registering users on the application page.
2. Make a note of the pod name.
3. Scale the service to two replicated pods. The WildFly application will then automatically replicate your session data in memory between pods.
4. Delete the original pod.
5. Verify that your session data is still active.

8.1.2 Querying the OpenShift API server from a pod

Before you get started, you need to modify some permissions for the default service account in your project, which will be responsible for running the application pods. From the stateful-apps project, run the following command to add the view role to the default service account. The view role will allow the pods running in the project to query the OpenShift API server directly. In this case, the application will take advantage of the ability to query OpenShift API server to find other WildFly application pods in the project. These other instances will send pod-to-pod traffic directly to each other and use their own application-specific service-discovery and load-balancing features for communication:

```
oc policy \
    add-role-to-user \
    view \
    system:serviceaccount:$(oc project -q):default \
    -n $(oc project -q)
```

Start by registering a few users in the WildFly application page. Choose Applications > Pods, and write down your pod name. Then, click the Overview tab in the left panel and scale up to two pods by clicking the up arrow. Once the second pod starts running, the data that you generated in the first step will be sent directly from the first pod to the second pod.

The two pods discovered each other with the help of a WildFly-specific discovery mechanism designed for Kubernetes. The implementation is called *KUBE_PING*, and it's part of the JGroups project. When the second pod was started, it queried the OpenShift API for all the pods in the current project. The API server then returned a list of pods in the current project. The KUBE_PING code in the WildFly server

filtered the list of pods for those with special ports labeled `ping`. If any of the pods in the result set returned from the API server match the filter, then the JGroups code in WildFly will attempt to join any existing clusters among the pods in the list.

> **NOTE** JGroups (www.jgroups.org) is a popular open source toolkit for reliable messaging written in Java and popular with Java application servers. WildFly uses JGroups under the covers to send messages back and forth between other application servers.

> **TIP** By default, the WildFly pods have several ports exposed. The main port that is used for direct pod-to-pod traffic is 8888. On that port is an embedded HTTP server that's used to send messages to and receive messages from other pods. If you examine the pod object, you'll notice that the port also has a matching name of `ping`, which is used as metadata about the port.

Take a moment to examine the result set from the pod perspective by navigating to any of the pods in the OpenShift console and clicking the Terminal tab. Then run this command to query the API server for a list of pods in the project matching the label `application=wildfly-app`:

```
curl -k -X GET \
  -H "Authorization: Bearer $(cat /var/run/secrets/kubernetes.io/serviceac
  ➥ count/token)" \
  https://$KUBERNETES_PORT_443_TCP_ADDR:$KUBERNETES_SERVICE_PORT_HTTPS/api
  ➥ /v1/namespaces/stateful-apps/pods?labelSelector=application%3Dwildfly-app
```

KUBE_PING also uses two environment variables that were automatically generated for you in the OpenShift template that you first used. Navigate to the Environment tab on the current page, and you'll see the `OPENSHIFT_KUBE_PING_NAMESPACE` and `OPENSHIFT_KUBE_PING_LABELS` variables set automatically, as shown in figure 8.3.

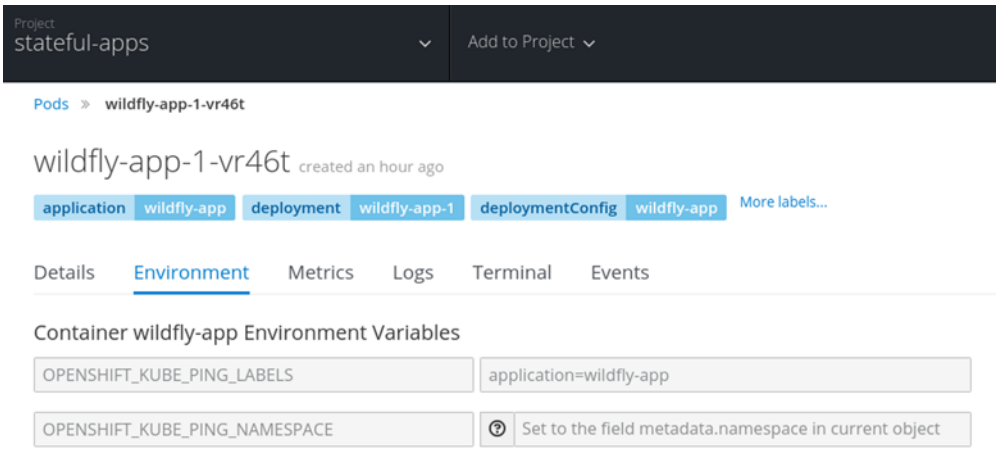


Figure 8.3 WildFly clustering environment variables

8.1.3 Verifying WildFly data replication

Now that two pods are successfully clustered together, delete the original pod from the OpenShift console by choosing Actions > Delete, as shown in figure 8.4. The OpenShift replication controller (RC) will notice that a pod has been deleted and will spin up a new one in its place to ensure that there are still two replicas. If clustering is working properly, the original data you entered will still be available, even though it was originally stored in memory in a pod that no longer exists. Double-check by refreshing the application in your browser. If your data is no longer there, go back and make sure you ran the `oc policy add-role-to-user` command properly from the stateful-apps project. If that doesn't resolve the issue, look at the pod logs for any noticeable errors.

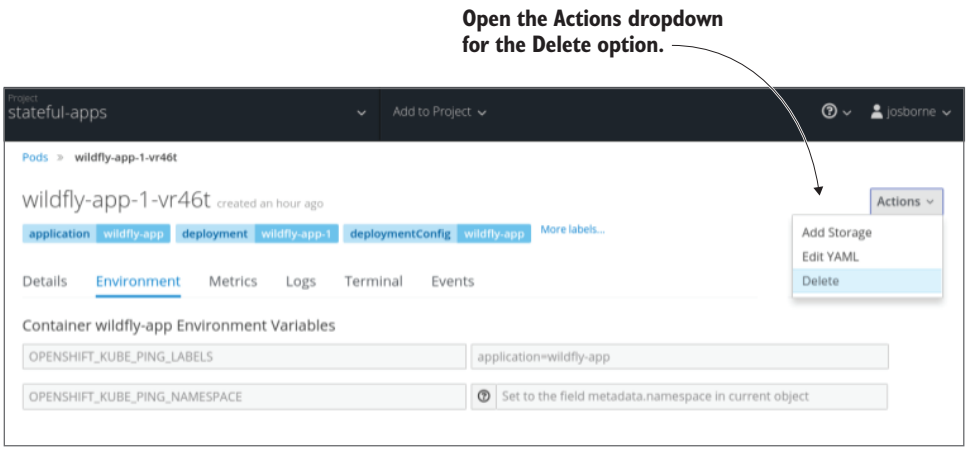


Figure 8.4 Delete a WildFly application pod

8.1.4 Other use cases for direct pod access

A Java application server that needs to cluster applications is just one common use case for direct pod discovery and access. Another example is an application that has its own load-balancing or routing mechanism, such as a *sharded database*. A sharded database is one in which large datasets are stored in many small databases as opposed to one large database. Many sharded databases have intelligence built into their clients and drivers that allows for direct access to the correct shard without querying or guessing where the data resides. Sharded databases work well on OpenShift and have been implemented using MongoDB as well as Infinispan, among others.

A typical sharded-database implementation may include creating the service object as a headless service. Once a headless service object is created, DNS can be used as another service-discovery mechanism. A DNS query for a given headless service will return A records for all the pods in the service. (More information on DNS and A records is available in chapter 10.) Applications can then implement custom logic to determine which pod to access.

One popular application that uses DNS queries to determine which instances to access is *Apache Kafka*, a fast open source messaging broker. Most implementations of Apache Kafka on OpenShift and other Kubernetes-based platforms use headless services so the messaging brokers can access each other directly to send and replicate messages. The brokers find each other using DNS queries, which are made possible by implementing a headless service.

Other common use cases for direct access include more mundane IT workloads such as software agents that are used for backups and monitoring. Backup agents are often run with many traditional database workloads and implement features such as scheduled snapshots and point-in-time recovery of data. A monitoring agent often provides features such as real-time alerting and visualization of an application. Often these agents may either run locally embedded as instrumented code in the application or communicate through direct network access. For many use cases, direct network access is required because the agents may communicate with more than one application across many servers. In these scenarios, the agents require consistent, direct access to applications in order to fulfill their daily functions.

8.2 Demonstrating sticky sessions

In the WildFly example, data is replicated between two WildFly server instances. A cookie with a unique identifier is generated automatically by the application and stored in your browser. By using a cookie, the application can track which end user is accessing the application. This approach works well but has several drawbacks. The most obvious is that if the WildFly server didn't support application clustering or didn't have a discovery mechanism that works in OpenShift, the application would produce uneven user experiences. Without application clustering, if there were two application pods—one with user data and one without user data—then the user would see their data only 50% of the time because requests are sent in a round-robin manner between pods in a service.

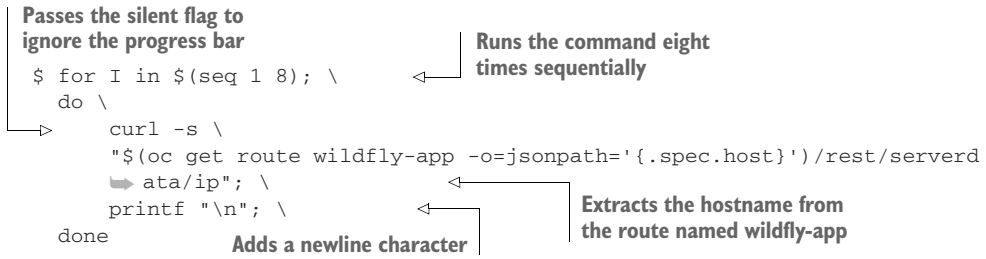
One common solution to this problem is to use *sticky sessions*. In the context of OpenShift, enabling sticky sessions ensures that a user making requests into the cluster will consistently receive responses from the same pod for the duration of their session.

This added consistency helps ensure a smooth user experience and allows many applications that temporarily store data locally in the container to be run in OpenShift. By default in OpenShift, sticky sessions are implemented using cookies for HTTP-based routes and some types of HTTPS-based routes. The OpenShift router can reuse existing cookies or create new cookies. The WildFly application you created earlier created its own cookie, so the router will use that cookie for the sticky-session implementation. If cookies are disabled or can't be used for the route, sticky sessions are implemented using a load-balancing scheme called *source* that uses the client IP address as part of its implementation.

TIP For more information on how to disable cookies, visit http://mng.bz/OYn9. For more information on load-balancing schemes such as the source scheme, visit http://mng.bz/1wMh.

8.2.1 Toggling sticky sessions

Let's see how sticky sessions work by toggling cookies on and off using the Linux `curl` command-line tool, which can make HTTP requests to a server eight times and print the results. The WildFly application you've deployed has a couple of REST endpoints that haven't been explored yet. The first endpoint can be used to print out the pod IP and hostname. Enter the following command to print out that information from eight sequential HTTP requests to the wildfly-app route:




```
$ for I in $(seq 1 8); \
  do \
      curl -s \
      "$(oc get route wildfly-app -o=jsonpath='{.spec.host}')/rest/serverd
      ata/ip"; \
      printf "\n"; \
  done
```

```
{"hostname":"wildfly-app-7-x4q1k","ip":"10.128.1.138"}
{"hostname":"wildfly-app-7-xrwsz","ip":"10.128.1.137"}
{"hostname":"wildfly-app-7-x4q1k","ip":"10.128.1.138"}
{"hostname":"wildfly-app-7-xrwsz","ip":"10.128.1.137"}
{"hostname":"wildfly-app-7-x4q1k","ip":"10.128.1.138"}
{"hostname":"wildfly-app-7-xrwsz","ip":"10.128.1.137"}
{"hostname":"wildfly-app-7-x4q1k","ip":"10.128.1.138"}
{"hostname":"wildfly-app-7-xrwsz","ip":"10.128.1.137"}
```

The output prints the hostname and IP address of each pod four times, alternating back and forth between pods in a round-robin pattern. This is as you'd expect because the `curl` command doesn't provide a cookie for the OpenShift router to track the origins of each request. Fortunately, `curl` can save cookies locally in a text file that can be used for future HTTP requests to the server. Use the following command to grab the cookie from the WildFly application and save it to a local file called cookie.txt:

```
$ curl -o /dev/null \
    --cookie-jar cookies.txt \
    $(oc get route wildfly-app \
    -o=jsonpath='{.spec.host}')/rest/serverdata/ip
```

Now that you've saved the cookie locally, send eight more requests to the wildfly-app route, using the cookie that's saved locally:


```
$ for I in $(seq 1 8); \
  do \
      curl -s \
      --cookie cookies.txt \
```

```
    "$(oc get route wildfly-app -o=jsonpath='{.spec.host}')/rest/serverd
    ➥ ata/ip"; \
    printf "\n"; \
done
```

```
{"hostname":"wildfly-app-7-x4q1k","ip":"10.128.1.138"}
{"hostname":"wildfly-app-7-x4q1k","ip":"10.128.1.138"}
{"hostname":"wildfly-app-7-x4q1k","ip":"10.128.1.138"}
{"hostname":"wildfly-app-7-x4q1k","ip":"10.128.1.138"}
{"hostname":"wildfly-app-7-x4q1k","ip":"10.128.1.138"}
{"hostname":"wildfly-app-7-x4q1k","ip":"10.128.1.138"}
{"hostname":"wildfly-app-7-x4q1k","ip":"10.128.1.138"}
{"hostname":"wildfly-app-7-x4q1k","ip":"10.128.1.138"}
```

Sticky sessions are now working! Unlike in the previous command, the eight requests aren't sent in a round-robin pattern. Each request is sent to the same pod consistently by using a cookie.

TIP Modern browsers automatically enable cookies for you, unless you've disabled this functionality.

LIMITATIONS OF USING COOKIES

One limitation of using cookies for load balancing is that they don't work for HTTPS connections that use the *passthrough* routing. In passthrough routing, there's an encrypted connection from the client—typically a browser—all the way to the application pod. In this scenario, cookies won't work, because the connection is encrypted from the client to the application; there's no way for the routing layer to view the cookie. To solve this problem, OpenShift uses the client IP address to implement sticky sessions. But this option has a couple of drawbacks.

First, many client IP addresses get translated using *Network Address Translation* (NAT) before reaching their destination. When a request is translated using NAT, it replaces the often-private IP address of the client with that of a public IP address. This frequently makes the client IP the same for all users on a particular home or business network. Imagine a scenario in which you ran three pods to run an application for everyone in your office, but everyone in your office was being routed to the same pod because the requests all appeared to show the same source IP address.

Second, OpenShift uses an internal hashing schema based on the client IP address and the number of pods to determine its load-balancing schema. When the number of replicas changes, such as when you're using autoscaling, it's possible to lose sticky sessions.

For the rest of this chapter, you won't need two instances of the WildFly application pod. So, scale back down to a single pod:

```
$ oc scale dc/wildfly-app --replicas=1
deploymentconfig "wildfly-app" scaled
```

TIP Previously, we mentioned that the WildFly application has two REST endpoints, but only one was covered. Another endpoint prints out the HTTP headers associated with the user and is available at http://wildfly-app-stateful-apps.apps.192.168.122.101.nip.io/rest/clientdata.

8.3 Shutting down applications gracefully

So far in this chapter, you've learned how to use sticky sessions to ensure that users have a consistent experience in OpenShift. You've also learned how to use custom load balancing and service discovery in OpenShift services. To demonstrate custom load balancing, you deployed an application that keeps user data in memory and replicates its data to other pods.

When looking at clustering, you entered data and then scaled up to two pods that replicated the data you entered. You then killed the original pod and verified that your data was still there.

This approach worked well but in a controlled and limited capacity. Imagine a scenario in which autoscaling was enabled and the pods were spinning up and down more quickly. How would you know the application data had been replicated before a particular pod was killed—or even which pod was killed? OpenShift has several ways to solve this issue.

8.3.1 Setting a grace period for application cleanup

The easiest and most straightforward solution is to use a *grace period* for the pod to gracefully shut down. Normally, when OpenShift deletes a pod, it sends the pod a Linux `TERM` signal, often abbreviated *SIGTERM*. The SIGTERM acts as a notification to the process that it needs to finish what it's doing and then exit. One caveat is that the application needs custom code to catch the signal and handle the shutdown sequence. Fortunately, many application servers have this code built in. If the container doesn't exit within a given grace period, OpenShift sends a Linux `Kill` signal (*SIGKILL*) that immediately terminates the application.

In this section, you'll deploy a new application to demonstrate how OpenShift grace periods work. In the same stateful-apps project that you're already in, run the following command to build and deploy the application:

```
$ oc new-app \
    -l app=graceful \
    --context-dir=dockerfile-graceful-shutdown \
    https://github.com/OpenShiftInAction/chapter8
...
--> Creating resources with label app=graceful ...
    imagestream "centos" created
    imagestream "chapter8" created
    buildconfig "chapter8" created
    deploymentconfig "chapter8" created
--> Success
    Build scheduled, use 'oc logs -f bc/chapter8' to track its progress.
    Run 'oc status' to view your app.
```

Learning more about source-to-image

Many of the applications so far in this book have used the OpenShift source-to-image (S2I) technology to build an application container image from source code. This approach extends the traditional Dockerfile approach, which uses standard Linux commands to build the container image. Both approaches are first-class citizens in OpenShift and have various advantages and disadvantages. One advantage of the Dockerfile approach is that it allows for the most extensibility and customization. S2I has a few major advantages:

- Easy-to-use.
- Customizable for most use cases.
- Removes the need to build and maintain a Dockerfile.
- Improves performance. In a Dockerfile, every command is a layer in the container image, whereas the entire S2I build processes a single layer that improves the size and speed of the container image.
- Allows platform administrators to limit how the images are built.

You can learn more about S2I at https://docs.openshift.org/latest/creating_images/s2i.html.

TIP Many of the objects created in the examples are called chapter8 because we didn't use a template for deployment. The name is based on the Git repository when using S2I.

The application may take a minute or so to build, because it may need to pull down a new base image to build the application. Once the application is successfully built and running, delete it with a grace period of 10 seconds:

```
$ oc delete pod -l app=graceful --grace-period=10
pod "dockerfile-graceful-shutdown-demo-1-1cbv1" deleted
```

When you run `delete` with a grace period of 10 seconds, OpenShift sends a SIGTERM signal immediately to the pod and then forcibly kills it in 10 seconds if it hasn't exited by itself. Quickly, run the following command to see this plays out in the logs for the pod:

```
$ oc logs -f \
    $(oc get pods -l app=graceful -o=jsonpath='{.items[].metadata.name}')
pid is 1
Waiting for SIGTERM, sleeping for 5 seconds now...
Waiting for SIGTERM, sleeping for 5 seconds now...
Waiting for SIGTERM, sleeping for 5 seconds now...
...
Caught SIGTERM! Gracefully shutting down now
Gracefully shutting down for 0 seconds
Gracefully shutting down for 1 seconds
Gracefully shutting down for 2 seconds
Gracefully shutting down for 3 seconds
```

```
Gracefully shutting down for 4 seconds
Gracefully shutting down for 5 seconds
Gracefully shutting down for 6 seconds
Gracefully shutting down for 7 seconds
Gracefully shutting down for 8 seconds
Gracefully shutting down for 9 seconds
```

The process that's running is a simple bash script that waits for a SIGTERM signal and then prints a message to standard out until it's killed. In this case, the pod was given a grace period of 10 seconds, and the pod printed logs for approximately 10 seconds before it was forcibly killed. By default, the grace period is set to 30 seconds. If you have an important container that you never want to be killed, you must set the `terminationGracePeriodSeconds` field in the deployment config to -1.

TIP In a container, the main process runs as process ID (PID) 1. This is important when handling Linux signals because only PID 1 receives the signal. Although most containers have a single process, many containers have multiple processes. In this scenario, the main process needs to catch the signal and notify the other process in the container. systemd can also be used as a seamless solution. For containers with multiple processes that all need to handle Linux signals, it's best to use systemd for this implementation. A CentOS base container that's available to build is available at https://hub.docker.com/r/centos/systemd.

TIP You can find a full list of Linux signals at https://en.wikipedia.org/wiki/Signal_(IPC).

You no longer need the graceful app demo, so delete all the resources it created:

```
$ oc delete all -l app=graceful
buildconfig "chapter8" deleted
imagestream "centos" deleted
imagestream "chapter8" deleted
deploymentconfig "chapter8" deleted
pod "chapter8-1-1j1zx" deleted
```

8.3.2 Using container lifecycle hooks

Although catching basic Linux signals such as SIGTERM is a best practice, many applications aren't equipped to handle Linux signals. A nice way to externalize the logic from the application is to use a *preStop* hook and one of two *container lifecycle hooks* available in OpenShift. Container lifecycle hooks allow users to take predetermined actions during a container management lifecycle event. The two events available in OpenShift are as follows:

- `PreStop`—Executes a handler before the container is terminated. This event is blocking, meaning it must finish before the pod is terminated.
- `PostStart`—Executes a handler immediately after the container is started.

Similar to readiness probes and liveness probes, the handler can be a command (often a script) that is executed in the container, or it can be an HTTP call to an endpoint exposed by the container.

TIP Container lifecycle hooks can be used in conjunction with pod grace periods. If preStop hooks are used, they take precedence over pod deletion. SIGTERM won't be sent to the container until the preStop hook finishes executing.

Using patch to set a preStop hook

To implement a preStop hook that calls the CLI tooling to initiate a graceful shutdown, you can use the `oc patch` command. This command can be used to update object fields. Here's an example of adding a preStop hook to the wildfly-app deployment config:

```
oc patch dc wildfly-app \
    -p '{"spec": {"template": {"spec": {"containers": [{"name": "wildfly-a
    ➥ pp","lifecycle": {"preStop": {"exec": {"command":
    ➥ ["/jboss-cli.sh", "
    ➥ --connect", "command=:shutdown[timeout=10]"]}}}}]}}}}'
```

CHOOSING THE BEST GRACEFUL SHUTDOWN METHOD

Container lifecycle hooks and Linux signal handling are often used together, but in many cases users decide which method to use for their application. The main benefit of using Linux signal handling is that the application will always behave the same way, no matter where the image is run. It guarantees consistent and predictable shutdown behavior because the behavior is coded in the application. Sending SIGTERM signals on delete is fundamental not only to all Kubernetes platforms but also to the standalone docker engine. If the user handles the SIGTERM signal in their application, the image will behave consistently even if it's moved outside of OpenShift. Because preStop hooks need to be explicitly added to the deployment, deployment config, or template, there's no guarantee that the image will behave the same way in other environments.

Many applications, such as third-party applications, don't handle SIGTERM properly, and the end user can't easily modify the code. In this case, a preStop hook must be used. A good example is *NGINX*, a popular and lightweight HTTP server. When NGINX is sent a SIGTERM, it exits immediately. Rather than forking NGINX and adding code to handle the Linux SIGTERM signal, an easy solution is to add a preStop hook that gracefully shuts down NGINX from the command line. A general rule to follow is that if you control the code, you should code your application to handle SIGTERM. If you don't control the code, use a preStop hook if needed.

8.4 Native API object support for stateful applications with stateful sets

So far in this chapter, you've learned that OpenShift has many capabilities to support stateful applications:

- Implementing custom load balancing
- Implementing custom service discovery
- Obtaining DNS A records on a per-pod-basis using a headless service
- Configuring sticky sessions
- Handling a controlled startup and shutdown sequence by handling Linux signals and container lifecycle events

These let users make traditional workloads first-class citizens on OpenShift, but some applications also require even more predictable startup and shutdown sequencing as well as predictable storage and networking-identifying information. Imagine a scenario with the WildFly application in which data replication is critical to the user experience, but a massive scaling event destroys too many pods at one time while replication is happening. How will the application recover? Where will the data be replicated to?

To solve this problem, OpenShift has a special object called a *stateful set* (known as a *pet set* in older versions of OpenShift). A stateful set is a powerful tool in the OpenShift users' toolbox to facilitate many traditional workloads in a modern environment. A stateful set object is used in place of a replication controller as the underlying implementation to ensure replicas in a service, but it does so in a more controlled way.

A replication controller can't control the order of how pods are created or destroyed. Normally, if a user configures a deployment to go from one to five replicas in OpenShift, that task is passed to an RC that starts four new pods all at once. The order in which they're started and marked as ready (successfully completing a readiness probe) is completely random.

8.4.1 Deterministic sequencing of startup and shutdown order with stateful sets

A stateful set brings a deterministic sequential order to pod creation and deletion. Each pod that's created also has an ordinal index number (starting at 0) associated with it. The ordinal index indicates the startup order. For instance, if the previous WildFly application was using a stateful set with three replicas, the pods would be started and named in this order: wildfly-app-0, wildfly-app-1, and wildfly-app-2. A stateful set also ensures that each pod is running and ready (has passed the readiness probe) before the next pod is started. In the previous scenario, wildfly-app-2 wouldn't be started until wildfly-app-1 was running and ready.

The reverse is also true. An RC or replica will delete pods at random when a command is given to reduce the number of replicas. A stateful set can also be used for a controlled shutdown sequence: it starts with the pod that has the highest ordinal index (n-1 replicas) and works backward to meet the new replica requirement. A pod won't be shut down until the previous pod has been fully terminated.

This controlled shutdown sequence can be critical for many stateful applications. In the case of the WildFly application, user data is being shared between a number of pods. When the WildFly application is shut down gracefully, a data-synchronization process may occur between the remaining pods in the application cluster. This process will

often be interrupted without the use of a stateful set because the pods are shut down in parallel. By using a predictable, one-at-a-time shutdown sequence, the application is less likely to lose data, which results in a better user experience.

8.4.2 Examining a stateful set

To see how stateful sets work, first create a new project:

```
$ oc new-project statefulset
Now using project "statefulset" on server "https://ocp-1.192.168.122.100.n
➥ ip.io:8443".
```

Now, import the template for this chapter's stateful set example:

```
$ oc create \
 -f https://raw.githubusercontent.com/OpenShiftInAction/chapter8/master/st
 ➥ atefulsets/mongodb-statefulset-replication-emptydir.yaml \
 -n statefulset
template "mongodb-statefulset-replication-emptydir" created
```

Adding the -n <namespace> tag to the end of the command means this template is available only in the statefulset project.

After you've installed the MongoDB template, go the statefulset project via the OpenShift console and click Add to Project. Filter for the template you just installed by typing `statefulset` in the OpenShift service catalog, as shown in figure 8.5. Once you select the template, you can modify the parameters for the MongoDB installation. None of the parameters are required because the OpenShift template will generate random values for anything left empty. Because this template has everything you need for the example already filled in, scroll to the bottom and click Create.

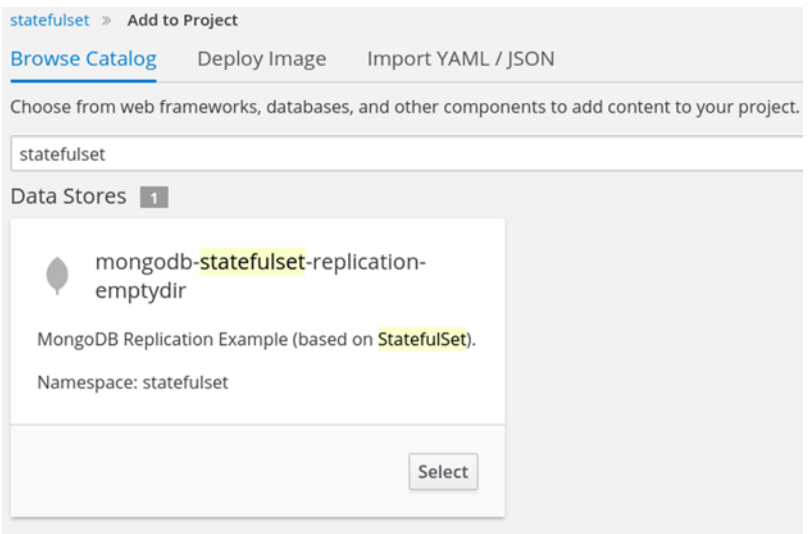


Figure 8.5 Find the MongoDB stateful set template in the service catalog.

TIP The template makes reference to a *replica set*. That's the MongoDB term for replicated MongoDB instances. This template doesn't create an OpenShift replica set object, which is similar to a replication controller.

Next, navigate to the stateful set in the OpenShift console by choosing Applications > Stateful Sets and selecting the mongodb stateful set object that was created. The details page for the stateful set object will open. Examine the two pods that the template created: as shown in figure 8.6, the bottom of the screen shows that the two pod names have an ordinal index associated with them. As mentioned earlier, that index also determines the startup and shutdown sequence.

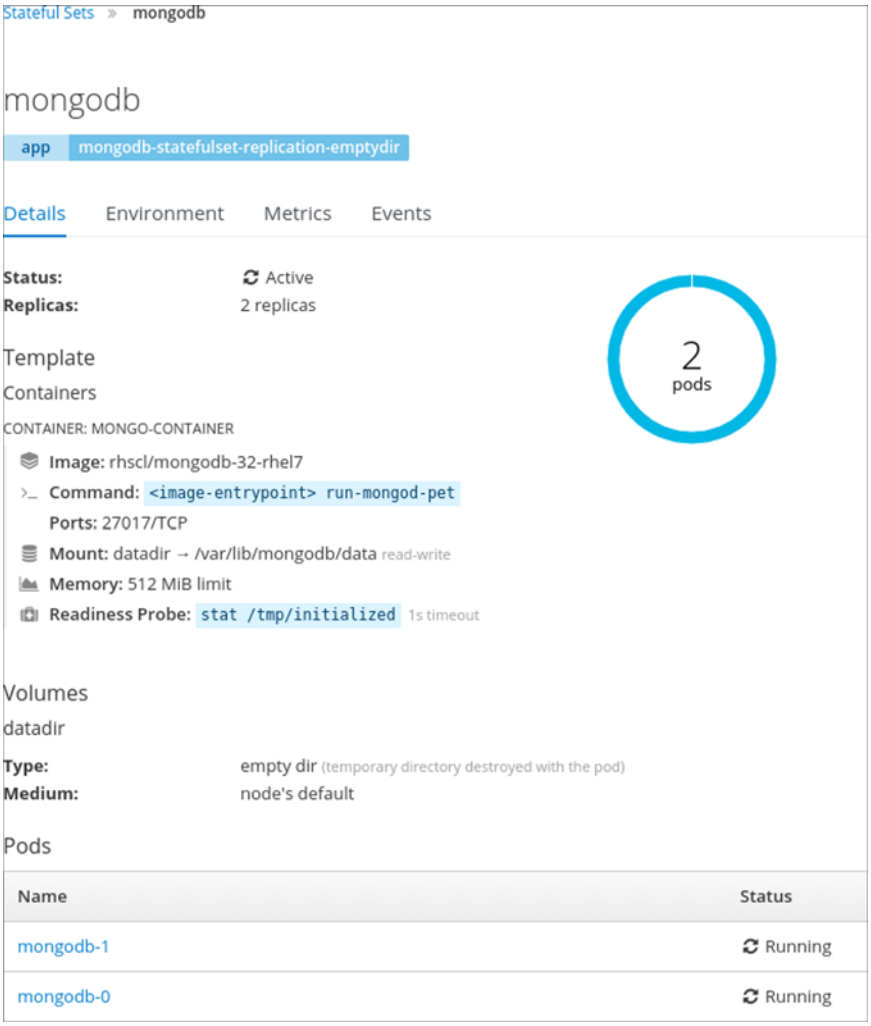


Figure 8.6 Example mongodb stateful set in the console

From the command line, modify the stateful set to spin up a third replica:

```
$ oc scale statefulsets mongodb --replicas=3
statefulset "mongodb" scaled
```

Unlike previous use of the `scale` command, this time you need to explicitly state that you're scaling a stateful set. In the OpenShift console, notice that the new pod that was created has a deterministic pod name with the ordinal index associated with it: mongodb-2.

Similar to the WildFly application, the three MongoDB pods are replicating data to each other. To check that this replication is fully functional, click any of the pods on the bottom of the mongodb stateful set Details page, and then click the Terminal tab. Any commands executed here will execute in the pod. First log in to mongodb as the admin user, as shown in figure 8.7; then check the status of the MongoDB replica set by typing `rs.status()` after a successful login.

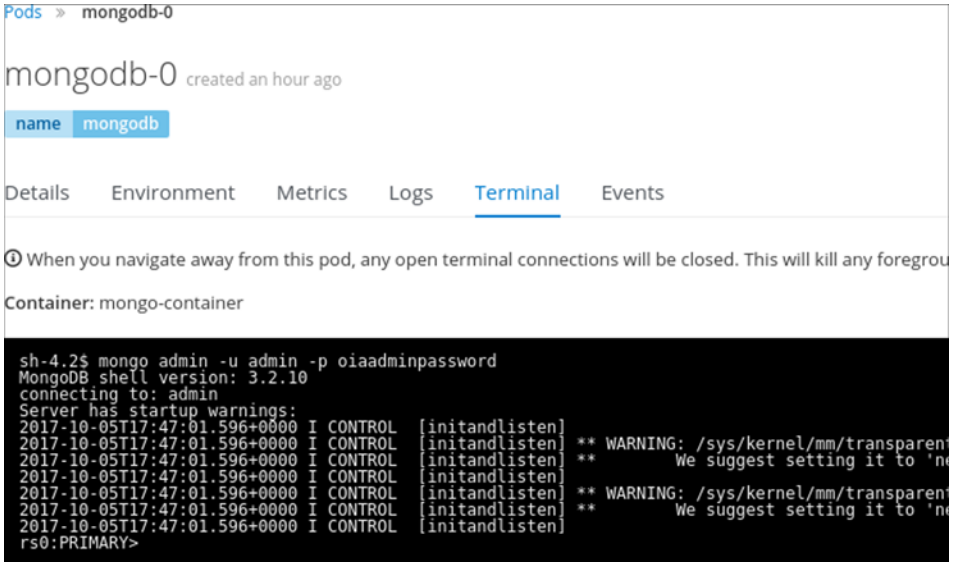


Figure 8.7 Log in as the mongo admin user.

> **TIP** The MongoDB replica set is now fully functional. You can learn more and see an architectural diagram of how this works at https://docs.mongodb.com/manual/replication.

8.4.3 Predictable network identity

Stateful sets also provide a consistent host-naming scheme for each pod in the set. Each predictable hostname is also associated with a predictable DNS entry. Examine the pod hostname for mongodb-0 by executing this command:

```
$ for statefulpod in $(oc get pods -l name=mongodb -o=jsonpath='{.items[*]
  ➥ .metadata.name}'); \
  do \
      oc exec $statefulpod cat /etc/hostname; \
  done
mongodb-0
mongodb-1
mongodb-2
```

The stateful set also ensures a DNS entry for each pod running in the stateful set. This can be found by executing the `dig` command using the DNS entry name for each pod. Find the IP addresses by executing the following command from one of the OpenShift nodes. Because the command relies on the OpenShift-provided DNS, it must be run from within the OpenShift environment to work properly:

```
$ for statefulpod in $(oc get pods -l name=mongodb -o=jsonpath='{.items[*]
  ➥ .metadata.name}'); \
   do \
       dig +short $statefulpod.mongodb-internal.statefulset.svc.cluster.lo
       ➥ cal; \
   done
10.128.1.164
10.128.1.165
10.128.1.166
```

TIP When you're using stateful sets, the pod hostname in DNS is listed in the format <pod name>.<service name>.<namespace>.svc.cluster.local.

Because this example also contains a headless service, there are DNS A records for the pods associated with the headless service. Ensure that the pod IPs in DNS match the previous listing by running this command from one of the OpenShift nodes:

```
$  dig +search +short mongodb-internal.statefulset.svc.cluster.local
10.128.1.164
10.128.1.165
10.128.1.166
```

8.4.4 Consistent persistent storage mappings

Pods running as part of a stateful set can also have their own persistent volume claims (PVCs) associated with each pod. But unlike a normal PVC, they remain associated with a pod and its ordinal index as long as the stateful set exists. In the previous example, you deployed an ephemeral stateful set without persistent storage.

Imagine that the previous example was using persistent storage, and the pods were writing log files that included the pod hostname. You wouldn't want the PVC to later be mapped to the volume of a different pod with a different hostname because it would be hard to make sense of those log files for debugging and auditing purposes. Stateful sets solve this problem by providing a consistent mapping through the use of a *volume claim template*, which is a template the PVC associates with each pod. If a pod

dies or is rescheduled to a different node, then the PVC will be mapped only to the new pod that starts in its place with the same hostname as the old pod. Providing a separate and dedicated persistent volume claim for each pod in the stateful set is crucial for many different types of stateful applications which cannot use the typical deployment config model of sharing the same PVC across many application instances.

NOTE A similar MongoDB stateful set example with persistent storage is available at http://mng.bz/LCaI.

8.4.5 Stateful set limitations

Under normal circumstances, pods controlled by a stateful set shouldn't need to be deleted manually. But there are a few scenarios in which a pod being controlled by a stateful set could be deleted by an outside force. For instance, if the kubelet or node is unresponsive, then the API server may remove the pod after a given amount of time and restart it somewhere else in the cluster. A pod could also exit accidentally or could be manually removed by a user. In those cases, it's likely that the ordinal index will be broken. New pods will be created with the same hostnames and DNS entries as the old pods, but the IP addresses may be different. For this reason, any application that relies on hardcoded IP addresses isn't a good fit for stateful sets. If the application can't be modified to use DNS or hostnames instead of IP addresses, you should use a single service per pod for a stable IP address.

Another limitation is that all the pods in a stateful set are replicas of each other, which of course makes sense when you want to scale. But that won't help any situation in which disparate applications need to be started in a particular order. A classic example is a Java or .NET application that throws errors if a database is unavailable. Once the database is started, then the application also needs to be restarted to refresh its connections. In that scenario, a stateful set wouldn't help the order between the two disparate services.

8.4.6 Stateful applications without native solutions

One of the reasons OpenShift has gained so much market adoption is that traditional IT workloads work just as well as modern stateless applications. Yet there's still work to be done. One of the biggest promises of using containers is that applications will behave the same way between environments. Containers start from well-known image binaries that contain the application and the configuration it needs to run. If a container dies, a new one is started from the previous image binary that's identical to how the previous container was started. One major problem with this model occurs for applications that are changed on the fly and store their information in a way that makes it difficult to re-create.

A good example of this issue can be seen with WordPress, an extremely popular blogging application that was designed many years before containers become popular. In a given WordPress workflow, a blogger might go to the admin portion of their website, add some text, and then save/publish it. WordPress saves all that text in a database,

along with any HTML and styling. When the blogger has completed this action, the container has drifted from its original image. Container drift is normal for most applications; but in the case of WordPress, if the container crashed, the blog would be lost. Persistent storage can be used to ensure that the data is persisted. When a new WordPress pod starts, it could map to the database and would have all the blogs available.

But promoting such a snapshot of a database among various environments is a major challenge. There are many examples of using an event-driven workflow that can be triggered to export and import a database after a blogger publishes content, but it's not easy, nor is it native to the platform. Containers start from well-known immutable container images, but engineering a reverse workflow in which images are created from running container instances is more error-prone and rigid. Other examples that have worked—with some engineering—include applications that open a large number of ports, applications the rely on hardcoded IP addresses, and other legacy applications that rely on older Linux technologies.

As OpenShift continues to evolve, the ecosystem of support for OpenShift and other Kubernetes-based offerings is also evolving. OpenShift keeps adding features to support even more stateful applications, and many companies are putting significant engineering resources behind making more cloud-native applications. This will further the process of making traditional IT workloads first-class citizens on OpenShift.

8.5 Summary

- OpenShift supports many different types of stateful applications.
- There are many ways in OpenShift to do custom service discovery.
- Custom load balancing can be implemented with a headless service.
- The OpenShift router supports session affinity with sticky sessions.
- Pods are sent Linux SIGTERM signals before being shut down.
- You can add a configurable grace period parameter before a pod is forcibly removed with SIGKILL.
- Container lifecycle events allow for actions to be taken when containers are started and stopped.
- A stateful set object allows a consistent startup and shutdown sequence.
- Pods can obtain their own hostname and consistent PVC mapping with a stateful set.
- Applications that need hardcoded IP addresses can use a single service per pod.

SOFTWARE DEVELOPMENT/LINUX

OpenShift IN ACTION

Duncan • Osborne

Containers let you package everything into one neat place, and with Red Hat OpenShift you can build, deploy, and run those packages all in one place! Combining Docker and Kubernetes, OpenShift is a powerful platform for cluster management, scaling, and upgrading your enterprise apps.

OpenShift in Action is a full reference to Red Hat OpenShift that breaks down this robust container platform so you can use it day-to-day. Starting with how to deploy and run your first application, you'll go deep into OpenShift. You'll discover crystal-clear explanations of namespaces, cgroups, and SELinux, learn to prepare a cluster, and even tackle advanced details like software-defined networks and security, with real-world examples you can take to your own work. It doesn't matter why you use OpenShift—by the end of this book you'll be able to handle every aspect of it, inside and out!

What's Inside

- Written by lead OpenShift architects
- Rock-solid fundamentals of Docker and Kubernetes
- Keep mission-critical applications up and running
- Manage persistent storage

For DevOps engineers and administrators working in a Linux-based distributed environment.

Jamie Duncan is a cloud solutions architect for Red Hat, focusing on large-scale OpenShift deployments. John Osborne is a principal OpenShift architect for Red Hat.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/openshift-in-action

"The first holistic view of OpenShift in print … a soup-to-nuts approach that combines both the developer and operator perspectives."

—From the Foreword by Jim Whitehurst, Red Hat

"At last, a much-needed guide to OpenShift! An excellent read crammed with practical hands-on exercises."

—Michael Bright, Containous

"The definitive guide to the base technologies of the containers era."

—Ioannis Sermetziadis Numbrs Personal Finance

"An essential resource. Gives a clear picture of a complex ecosystem."

—Bruno Vernay, Schneider Electric

ISBN-13: 978-1-61729-483-9
ISBN-10: 1-61729-483-7
54499
9 781617 294839

MANNING $44.99 / Can $59.99 [INCLUDING eBOOK]