```
Ready:              False
Restart Count:      6
```

As you can see, the "`I've had enough`" message the process wrote to the file /var/termination-reason is shown in the container's `Last State` section. Note that this mechanism isn't limited only to containers that crash. It can also be used in pods that run a completable task and terminate successfully (you'll find an example in the file termination-message-success.yaml).

This mechanism is great for terminated containers, but you'll probably agree that a similar mechanism would also be useful for showing app-specific status messages of running, not only terminated, containers. Kubernetes currently doesn't provide any such functionality and I'm not aware of any plans to introduce it.

> **NOTE** If the container doesn't write the message to any file, you can set the `terminationMessagePolicy` field to `FallbackToLogsOnError`. In that case, the last few lines of the container's log are used as its termination message (but only when the container terminates unsuccessfully).

17.4.6 Handling application logs

While we're on the subject of application logging, let's reiterate that apps should write to the standard output instead of files. This makes it easy to view logs with the `kubectl logs` command.

> **TIP** If a container crashes and is replaced with a new one, you'll see the new container's log. To see the previous container's logs, use the `--previous` option with `kubectl logs`.

If the application logs to a file instead of the standard output, you can display the log file using an alternative approach:

```
$ kubectl exec <pod> cat <logfile>
```

This executes the `cat` command inside the container and streams the logs back to kubectl, which prints them out in your terminal.

COPYING LOG AND OTHER FILES TO AND FROM A CONTAINER

You can also copy the log file to your local machine using the `kubectl cp` command, which we haven't looked at yet. It allows you to copy files from and into a container. For example, if a pod called `foo-pod` and its single container contains a file at `/var/log/foo.log`, you can transfer it to your local machine with the following command:

```
$ kubectl cp foo-pod:/var/log/foo.log foo.log
```

To copy a file from your local machine into the pod, specify the pod's name in the second argument:

```
$ kubectl cp localfile foo-pod:/etc/remotefile
```

This copies the file localfile to /etc/remotefile inside the pod's container. If the pod has more than one container, you specify the container using the `-c containerName` option.

USING CENTRALIZED LOGGING

In a production system, you'll want to use a centralized, cluster-wide logging solution, so all your logs are collected and (permanently) stored in a central location. This allows you to examine historical logs and analyze trends. Without such a system, a pod's logs are only available while the pod exists. As soon as it's deleted, its logs are deleted also.

Kubernetes by itself doesn't provide any kind of centralized logging. The components necessary for providing a centralized storage and analysis of all the container logs must be provided by additional components, which usually run as regular pods in the cluster.

Deploying centralized logging solutions is easy. All you need to do is deploy a few YAML/JSON manifests and you're good to go. On Google Kubernetes Engine, it's even easier. Check the Enable Stackdriver Logging checkbox when setting up the cluster. Setting up centralized logging on an on-premises Kubernetes cluster is beyond the scope of this book, but I'll give you a quick overview of how it's usually done.

You may have already heard of the ELK stack composed of ElasticSearch, Logstash, and Kibana. A slightly modified variation is the EFK stack, where Logstash is replaced with FluentD.

When using the EFK stack for centralized logging, each Kubernetes cluster node runs a FluentD agent (usually as a pod deployed through a DaemonSet), which is responsible for gathering the logs from the containers, tagging them with pod-specific information, and delivering them to ElasticSearch, which stores them persistently. ElasticSearch is also deployed as a pod somewhere in the cluster. The logs can then be viewed and analyzed in a web browser through Kibana, which is a web tool for visualizing ElasticSearch data. It also usually runs as a pod and is exposed through a Service. The three components of the EFK stack are shown in the following figure.

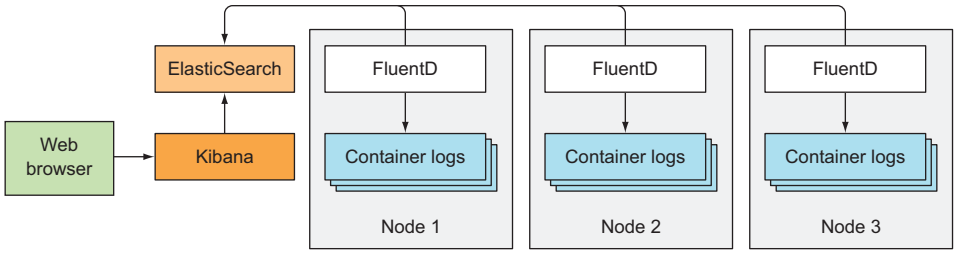

Figure 17.10 Centralized logging with FluentD, ElasticSearch, and Kibana

NOTE In the next chapter, you'll learn about Helm charts. You can use charts created by the Kubernetes community to deploy the EFK stack instead of creating your own YAML manifests.

HANDLING MULTI-LINE LOG STATEMENTS

The FluentD agent stores each line of the log file as an entry in the ElasticSearch data store. There's one problem with that. Log statements spanning multiple lines, such as exception stack traces in Java, appear as separate entries in the centralized logging system.

To solve this problem, you can have the apps output JSON instead of plain text. This way, a multiline log statement can be stored and shown in Kibana as a single entry. But that makes viewing logs with `kubectl logs` much less human-friendly.

The solution may be to keep outputting human-readable logs to standard output, while writing JSON logs to a file and having them processed by FluentD. This requires configuring the node-level FluentD agent appropriately or adding a logging sidecar container to every pod.

17.5 Best practices for development and testing

We've talked about what to be mindful of when developing apps, but we haven't talked about the development and testing workflows that will help you streamline those processes. I don't want to go into too much detail here, because everyone needs to find what works best for them, but here are a few starting points.

17.5.1 Running apps outside of Kubernetes during development

When you're developing an app that will run in a production Kubernetes cluster, does that mean you also need to run it in Kubernetes during development? Not really. Having to build the app after each minor change, then build the container image, push it to a registry, and then re-deploy the pods would make development slow and painful. Luckily, you don't need to go through all that trouble.

You can always develop and run apps on your local machine, the way you're used to. After all, an app running in Kubernetes is a regular (although isolated) process running on one of the cluster nodes. If the app depends on certain features the Kubernetes environment provides, you can easily replicate that environment on your development machine.

I'm not even talking about running the app in a container. Most of the time, you don't need that—you can usually run the app directly from your IDE.

CONNECTING TO BACKEND SERVICES

In production, if the app connects to a backend Service and uses the `BACKEND_SERVICE_HOST` and `BACKEND_SERVICE_PORT` environment variables to find the Service's coordinates, you can obviously set those environment variables on your local machine manually and point them to the backend Service, regardless of if it's running outside or inside a Kubernetes cluster. If it's running inside Kubernetes, you can always (at least temporarily) make the Service accessible externally by changing it to a `NodePort` or a `LoadBalancer`-type Service.