CHAPTER 13

Service Discovery

The *Service Discovery* pattern provides a stable endpoint through which consumers of a service can access the instances providing the service. For this purpose, Kubernetes provides multiple mechanisms, depending on whether the service consumers and producers are located on or off the cluster.

Problem

Applications deployed on Kubernetes rarely exist on their own, and usually they have to interact with other services within the cluster or systems outside the cluster. The interaction can be initiated internally within the service or through external stimulus. Internally initiated interactions are usually performed through a polling consumer: either after startup or later, an application connects to another system and starts sending and receiving data. Typical examples are an application running within a Pod that reaches a file server and starts consuming files, or a message that connects to a message broker and starts receiving or sending messages, or an application that uses a relational database or a key-value store and starts reading or writing data.

The critical distinction here is that the application running within the Pod decides at some point to open an outgoing connection to another Pod or external system and starts exchanging data in either direction. In this scenario, we don't have an external stimulus for the application, and we don't need any additional setup in Kubernetes.

To implement the patterns described in Chapter 7, "Batch Job", or Chapter 8, "Periodic Job", we often use this technique. In addition, long-running Pods in DaemonSets or ReplicaSets sometimes actively connect to other systems over the network. The more common use case for Kubernetes workloads occurs when we have long-running services expecting external stimulus, most commonly in the form of incoming HTTP connections from other Pods within the cluster or external systems. In these cases,

service consumers need a mechanism for discovering Pods that are dynamically placed by the scheduler and sometimes elastically scaled up and down.

It would be a significant challenge if we had to track, register, and discover endpoints of dynamic Kubernetes Pods ourselves. That is why Kubernetes implements the *Service Discovery* pattern through different mechanisms, which we explore in this chapter.

Solution

If we look at the "Before Kubernetes Era," the most common mechanism of service discovery was through client-side discovery. In this architecture, when a service consumer had to call another service that might be scaled to multiple instances, the service consumer would have a discovery agent capable of looking at a registry for service instances and then choosing one to call. Classically, that would be done, for example, either with an embedded agent within the consumer service (such as a ZooKeeper client, Consul client, or Ribbon) or with another colocated process looking up the service in a registry, as shown in Figure 13-1.

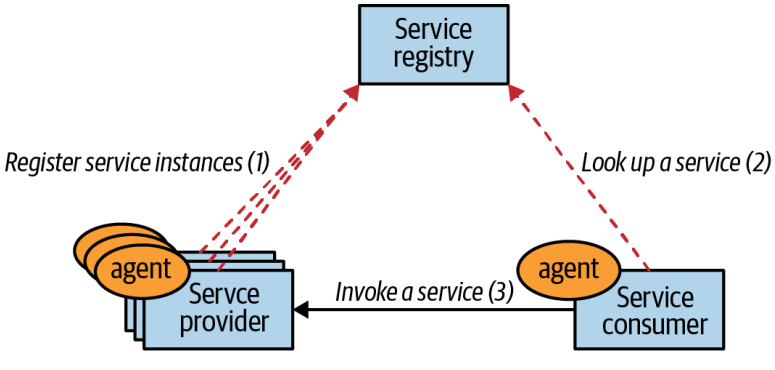

Figure 13-1. Client-side service discovery

In the "Post Kubernetes Era," many of the nonfunctional responsibilities of distributed systems such as placement, health checks, healing, and resource isolation are moving into the platform, and so is service discovery and load balancing. If we use the definitions from service-oriented architecture (SOA), a service provider instance still has to register itself with a service registry while providing the service capabilities, and a service consumer has to access the information in the registry to reach the service.

In the Kubernetes world, all that happens behind the scenes so that a service consumer calls a fixed virtual Service endpoint that can dynamically discover service instances implemented as Pods. Figure 13-2 shows how registration and lookup are embraced by Kubernetes.

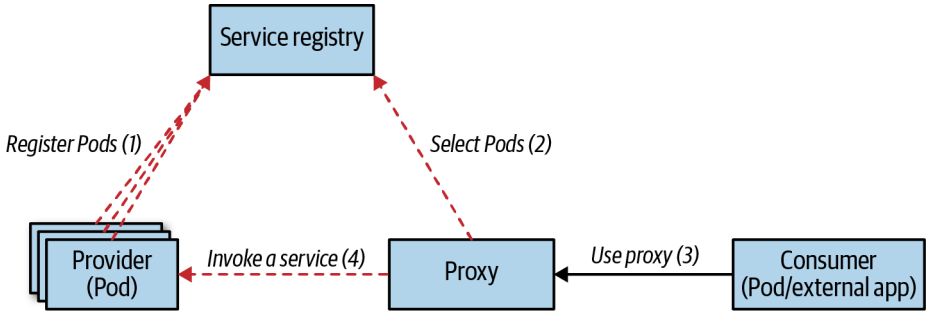

Figure 13-2. Server-side service discovery

At first glance, *Service Discovery* may seem like a simple pattern. However, multiple mechanisms can be used to implement this pattern, which depends on whether a service consumer is within or outside the cluster and whether the service provider is within or outside the cluster.

Internal Service Discovery

Let's assume we have a web application and want to run it on Kubernetes. As soon as we create a Deployment with a few replicas, the scheduler places the Pods on the suitable nodes, and each Pod gets a cluster-internal IP address assigned before starting up. If another client service within a different Pod wishes to consume the web application endpoints, there isn't an easy way to know the IP addresses of the service provider Pods in advance.

This challenge is what the Kubernetes Service resource addresses. It provides a constant and stable entry point for a collection of Pods offering the same functionality. The easiest way to create a Service is through `kubectl expose`, which creates a Service for a Pod or multiple Pods of a Deployment or ReplicaSet. The command creates a virtual IP address referred to as the `clusterIP`, and it pulls both Pod selectors and port numbers from the resources to create the Service definition. However, to have full control over the definition, we create the Service manually, as shown in Example 13-1.

Example 13-1. A simple Service

```
apiVersion: v1
kind: Service
metadata:
  name: random-generator
spec:
  selector:                  ❶
    app: random-generator
  ports:
  - port: 80                 ❷
    targetPort: 8080         ❸
    protocol: TCP
```

❶ Selector matching Pod labels.

❷ Port over which this Service can be contacted.

❸ Port on which the Pods are listening.

The definition in this example will create a Service named `random-generator` (the name is important for discovery later) and `type: ClusterIP` (which is the default) that accepts TCP connections on port 80 and routes them to port 8080 on all the matching Pods with the selector `app: random-generator`. It doesn't matter when or how the Pods are created—any matching Pod becomes a routing target, as illustrated in Figure 13-3.

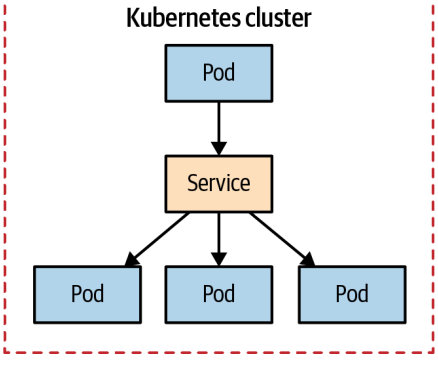

Figure 13-3. Internal service discovery

The essential points to remember here are that once a Service is created, it gets a `clusterIP` assigned that is accessible only from within the Kubernetes cluster (hence the name), and that IP remains unchanged as long as the Service definition exists. However, how can other applications within the cluster figure out what this dynamically allocated `clusterIP` is? There are two ways:

Discovery through environment variables
: When Kubernetes starts a Pod, its environment variables get populated with the details of all Services that exist up to that moment. For example, our `random-generator` Service listening on port 80 gets injected into any newly starting Pod, as the environment variables shown in Example 13-2 demonstrate. The application running that Pod would know the name of the Service it needs to consume and can be coded to read these environment variables. This lookup is a simple mechanism that can be used from applications written in any language and is also easy to emulate outside the Kubernetes cluster for development and testing purposes. The main issue with this mechanism is the temporal dependency on Service creation. Since environment variables cannot be injected into already-running Pods, the Service coordinates are available only for Pods started after the Service is created in Kubernetes. That requires the Service to be defined before starting the Pods that depend on the Service—or if this is not the case, the Pods need to be restarted.

Example 13-2. Service-related environment variables set automatically in Pod

```
RANDOM_GENERATOR_SERVICE_HOST=10.109.72.32
RANDOM_GENERATOR_SERVICE_PORT=80
```

Discovery through DNS lookup
: Kubernetes runs a DNS server that all the Pods are automatically configured to use. Moreover, when a new Service is created, it automatically gets a new DNS entry that all Pods can start using. Assuming a client knows the name of the Service it wants to access, it can reach the Service by a fully qualified domain name (FQDN) such as `random-generator.default.svc.cluster.local`. Here, `random-generator` is the name of the Service, `default` is the name of the namespace, `svc` indicates it is a Service resource, and `cluster.local` is the cluster-specific suffix. We can omit the cluster suffix if desired, and the namespace as well when accessing the Service from the same namespace.

The DNS discovery mechanism doesn't suffer from the drawbacks of the environment-variable-based mechanism, as the DNS server allows lookup of all Services to all Pods as soon as a Service is defined. However, you may still need to use the environment variables to look up the port number to use if it is a nonstandard one or unknown by the service consumer.

Here are some other high-level characteristics of the Service with `type: ClusterIP` that other types build upon:

Multiple ports
: A single Service definition can support multiple source and target ports. For example, if your Pod supports both HTTP on port 8080 and HTTPS on port 8443, there is no need to define two Services. A single Service can expose both ports on 80 and 443, for example.

Session affinity
: When there is a new request, the Service randomly picks a Pod to connect to by default. That can be changed with `sessionAffinity: ClientIP`, which makes all requests originating from the same client IP stick to the same Pod. Remember that Kubernetes Services performs L4 transport layer load balancing, and it cannot look into the network packets and perform application-level load balancing such as HTTP cookie-based session affinity.

Readiness probes
: In Chapter 4, "Health Probe", you learned how to define a `readinessProbe` for a container. If a Pod has defined readiness checks, and they are failing, the Pod is removed from the list of Service endpoints to call even if the label selector matches the Pod.

Virtual IP
: When we create a Service with `type: ClusterIP`, it gets a stable virtual IP address. However, this IP address does not correspond to any network interface and doesn't exist in reality. It is the kube-proxy that runs on every node that picks this new Service and updates the iptables of the node with rules to catch the network packets destined for this virtual IP address and replaces it with a selected Pod IP address. The rules in the iptables do not add ICMP rules, but only the protocol specified in the Service definition, such as TCP or UDP. As a consequence, it is not possible to `ping` the IP address of the Service as that operation uses the ICMP.

Choosing ClusterIP
: During Service creation, we can specify an IP to use with the field `.spec.clusterIP`. It must be a valid IP address and within a predefined range. While not recommended, this option can turn out to be handy when dealing with legacy

applications configured to use a specific IP address, or if there is an existing DNS entry we wish to reuse.

Kubernetes Services with `type: ClusterIP` are accessible only from within the cluster; they are used for discovery of Pods by matching selectors and are the most commonly used type. Next, we will look at other types of Services that allow discovery of endpoints that are manually specified.

Manual Service Discovery

When we create a Service with `selector`, Kubernetes tracks the list of matching and ready-to-serve Pods in the list of endpoint resources. For Example 13-1, you can check all endpoints created on behalf of the Service with `kubectl get endpoints random-generator`. Instead of redirecting connections to Pods within the cluster, we could also redirect connections to external IP addresses and ports. We can do that by omitting the `selector` definition of a Service and manually creating endpoint resources, as shown in Example 13-3.

Example 13-3. Service without selector

```
apiVersion: v1
kind: Service
metadata:
  name: external-service
spec:
  type: ClusterIP
  ports:
  - protocol: TCP
    port: 80
```

Next, in Example 13-4, we define an endpoint resource with the same name as the Service and containing the target IPs and ports.

Example 13-4. Endpoints for an external service

```
apiVersion: v1
kind: Endpoints
metadata:
  name: external-service    ❶
subsets:
  - addresses:
    - ip: 1.1.1.1
    - ip: 2.2.2.2
    ports:
    - port: 8080
```

❶ Name must match the Service that accesses these endpoints.

This Service is also accessible only within the cluster and can be consumed in the same way as the previous ones, through environment variables or DNS lookup. The difference is that the list of endpoints is manually maintained and those values usually point to IP addresses outside the cluster, as demonstrated in Figure 13-4.

While connecting to an external resource is this mechanism's most common use, it is not the only one. Endpoints can hold IP addresses of Pods but not virtual IP addresses of other Services. One good thing about the Service is that it allows you to add and remove selectors and point to external or internal providers without deleting the resource definition that would lead to a Service IP address change. So service consumers can continue using the same Service IP address they first pointed to while the actual service provider implementation is migrated from on-premises to Kubernetes without affecting the client.

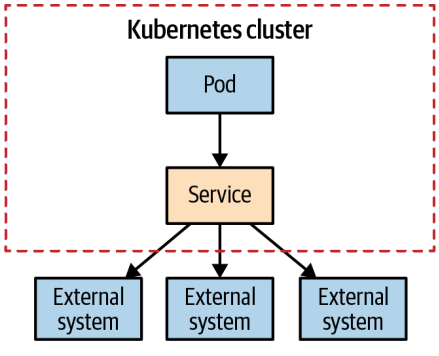

Figure 13-4. Manual service discovery

In this category of manual destination configuration, there is one more type of Service, as shown in Example 13-5.

Example 13-5. Service with an external destination

```
apiVersion: v1
kind: Service
metadata:
  name: database-service
spec:
  type: ExternalName
  externalName: my.database.example.com
  ports:
  - port: 80
```

This Service definition does not have a `selector` either, but its type is `ExternalName`. That is an important difference from an implementation point of view. This Service definition maps to the content pointed by `externalName` using DNS only, or more

specifically, `database-service.<namespace>.svc.cluster.local` will now point to `my.database.example.com`. It is a way of creating an alias for an external endpoint using DNS CNAME rather than going through the proxy with an IP address. But fundamentally, it is another way of providing a Kubernetes abstraction for endpoints located outside the cluster.

Service Discovery from Outside the Cluster

The service discovery mechanisms discussed so far in this chapter all use a virtual IP address that points to Pods or external endpoints, and the virtual IP address itself is accessible only from within the Kubernetes cluster. However, a Kubernetes cluster doesn't run disconnected from the rest of the world, and in addition to connecting to external resources from Pods, very often the opposite is also required—external applications wanting to reach to endpoints provided by the Pods. Let's see how to make Pods accessible for clients living outside the cluster.

The first method to create a Service and expose it outside of the cluster is through `type: NodePort`. The definition in Example 13-6 creates a Service as earlier, serving Pods that match the selector `app: random-generator`, accepting connections on port 80 on the virtual IP address and routing each to port 8080 of the selected Pod. However, in addition to all of that, this definition also reserves port 30036 on all the nodes and forwards incoming connections to the Service. This reservation makes the Service accessible internally through the virtual IP address, as well as externally through a dedicated port on every node.

Example 13-6. Service with type NodePort

```
apiVersion: v1
kind: Service
metadata:
  name: random-generator
spec:
  type: NodePort              ❶
  selector:
    app: random-generator
  ports:
  - port: 80
    targetPort: 8080
    nodePort: 30036           ❷
    protocol: TCP
```

❶ Open port on all nodes.

❷ Specify a fixed port (which needs to be available) or leave this out to get a randomly selected port assigned.

While this method of exposing services (illustrated in Figure 13-5) may seem like a good approach, it has drawbacks.

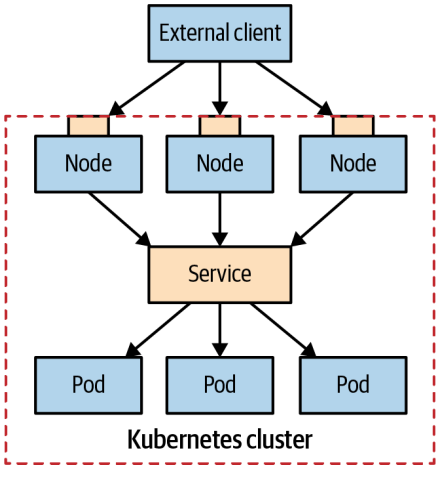

Figure 13-5. Node port service discovery

Let's see some of its distinguishing characteristics:

Port number
: Instead of picking a specific port with `nodePort: 30036`, you can let Kubernetes pick a free port within its range.

Firewall rules
: Since this method opens a port on all the nodes, you may have to configure additional firewall rules to let external clients access the node ports.

Node selection
: An external client can open connection to any node in the cluster. However, if the node is not available, it is the responsibility of the client application to connect to another healthy node. For this purpose, it may be a good idea to put a load balancer in front of the nodes that picks healthy nodes and performs failover.

Pods selection
: When a client opens a connection through the node port, it is routed to a randomly chosen Pod that may be on the same node where the connection was open or a different node. It is possible to avoid this extra hop and always force Kubernetes to pick a Pod on the node where the connection was opened by adding `externalTrafficPolicy: Local` to the Service definition. When this option is set, Kubernetes does not allow you to connect to Pods located on other nodes, which can be an issue. To resolve that, you have to either make sure there

are Pods placed on every node (e.g., by using daemon services) or make sure the client knows which nodes have healthy Pods placed on them.

Source addresses
: There are some peculiarities around the source addresses of packets sent to different types of Services. Specifically, when we use type `NodePort`, client addresses are source NAT'd, which means the source IP addresses of the network packets containing the client IP address are replaced with the node's internal addresses. For example, when a client application sends a packet to node 1, it replaces the source address with its node address, replaces the destination address with the Pod's address, and forwards the packet to node 2, where the Pod is located. When the Pod receives the network packet, the source address is not equal to the original client's address but is the same as node 1's address. To prevent this from happening, we can set `externalTrafficPolicy: Local` as described earlier and forward traffic only to Pods located on node 1.

Another way to perform Service Discovery for external clients is through a load balancer. You have seen how a `type: NodePort` Service builds on top of a regular Service with `type: ClusterIP` by also opening a port on every node. The limitation of this approach is that we still need a load balancer for client applications to pick a healthy node. The Service type `LoadBalancer` addresses this limitation.

In addition to creating a regular Service, and opening a port on every node, as with `type: NodePort`, it also exposes the service externally using a cloud provider's load balancer. Figure 13-6 shows this setup: a proprietary load balancer serves as a gateway to the Kubernetes cluster.

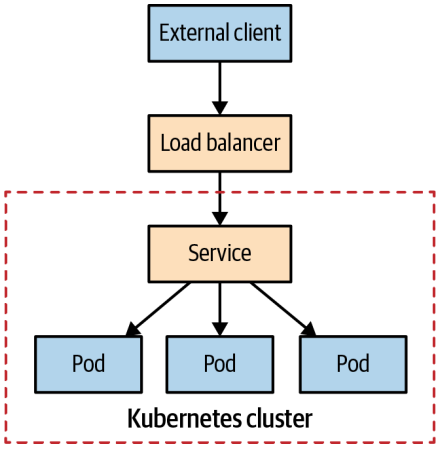

Figure 13-6. Load balancer service discovery

So this type of Service works only when the cloud provider has Kubernetes support and provisions a load balancer. We can create a Service with a load balancer by specifying the type `LoadBalancer`. Kubernetes then will add IP addresses to the `.spec` and `.status` fields, as shown in Example 13-7.

Example 13-7. Service of type LoadBalancer

```
apiVersion: v1
kind: Service
metadata:
  name: random-generator
spec:
  type: LoadBalancer
  clusterIP: 10.0.171.239          ❶
  loadBalancerIP: 78.11.24.19
  selector:
    app: random-generator
  ports:
  - port: 80
    targetPort: 8080
status:                            ❷
  loadBalancer:
    ingress:
    - ip: 146.148.47.155
```

❶ Kubernetes assigns `clusterIP` and `loadBalancerIP` when they are available.

❷ The `status` field is managed by Kubernetes and adds the Ingress IP.

With this definition in place, an external client application can open a connection to the load balancer, which picks a node and locates the Pod. The exact way that load-balancer provisioning and service discovery are performed varies among cloud providers. Some cloud providers will allow you to define the load-balancer address and some will not. Some offer mechanisms for preserving the source address, and some replace that with the load-balancer address. You should check the specific implementation provided by your cloud provider of choice.

Yet another type of Service is available: *headless* services, for which you don't request a dedicated IP address. You create a headless service by specifying `clusterIP None` within the Service's `spec` section. For headless services, the backing Pods are added to the internal DNS server and are most useful for implementing Services to StatefulSets, as described in detail in Chapter 12, "Stateful Service".

Application Layer Service Discovery

Unlike the mechanisms discussed so far, Ingress is not a service type but a separate Kubernetes resource that sits in front of Services and acts as a smart router and entry point to the cluster. Ingress typically provides HTTP-based access to Services through externally reachable URLs, load balancing, TLS termination, and name-based virtual hosting, but there are also other specialized Ingress implementations. For Ingress to work, the cluster must have one or more Ingress controllers running. A simple Ingress that exposes a single Service is shown in Example 13-8.

Example 13-8. An Ingress definition

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: random-generator
spec:
  defaultBackend:
    service:
      name: random-generator
      port:
        number: 8080
```

Depending on the infrastructure Kubernetes is running on, and the Ingress controller implementation, this definition allocates an externally accessible IP address and exposes the `random-generator` Service on port 80. But this is not very different from a Service with `type: LoadBalancer`, which requires an external IP address per Service definition. The real power of Ingress comes from reusing a single external load balancer and IP to service multiple Services and reduce the infrastructure costs. A simple fan-out configuration for routing a single IP address to multiple Services based on HTTP URI paths looks like Example 13-9.

Example 13-9. A definition for Nginx Ingress controller

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: random-generator
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:                          ❶
  - http:
      paths:
      - path: /                   ❷
        pathType: Prefix
        backend:
          service:
            name: random-generator
            port:
              number: 8080
      - path: /cluster-status     ❸
        pathType: Exact
        backend:
          service:
            name: cluster-status
            port:
              number: 80
```

❶ Dedicated rules for the Ingress controller for dispatching requests based on the request path.

❷ Redirect every request to Service `random-generator`…

❸ … except `/cluster-status`, which goes to another Service.

Since every Ingress controller implementation is different, apart from the usual Ingress definition, a controller may require additional configuration, which is passed through annotations. Assuming the Ingress is configured correctly, the preceding definition would provision a load balancer and get an external IP address that services two Services under two different paths, as shown in Figure 13-7.

Ingress is the most powerful and at the same time most complex service discovery mechanism on Kubernetes. It is most useful for exposing multiple services under the same IP address and when all services use the same L7 (typically HTTP) protocol.

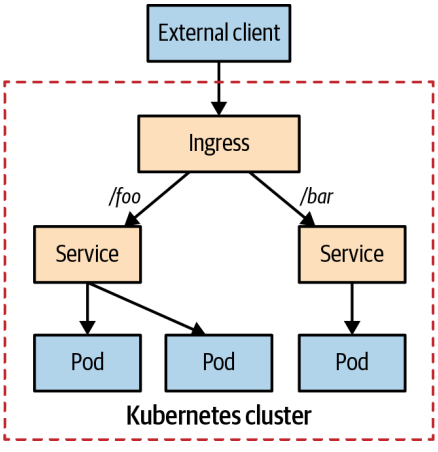

Figure 13-7. Application layer service discovery

OpenShift Routes

Red Hat OpenShift is a popular enterprise distribution of Kubernetes. Besides being fully compliant with Kubernetes, OpenShift provides some additional features. One of these features is Routes, which are very similar to Ingress. They are so similar, in fact, the differences might be difficult to spot. First of all, Routes predates the introduction of the Ingress object in Kubernetes, so Routes can be considered a kind of predecessor of Ingress.

However, some technical differences still exist between Routes and Ingress objects:

- A Route is picked up automatically by the OpenShift-integrated HAProxy load balancer, so there is no requirement for an extra Ingress controller to be installed.
- You can use additional TLS termination modes like re-encryption or pass-through for the leg to the Service.
- Multiple weighted backends for splitting traffic can be used.
- Wildcard domains are supported.

Having said all that, you can use Ingress on OpenShift too. So you have the choice when using OpenShift.

Discussion

In this chapter, we covered the favorite service discovery mechanisms on Kubernetes. Discovery of dynamic Pods from within the cluster is always achieved through the Service resource, though different options can lead to different implementations. The Service abstraction is a high-level cloud native way of configuring low-level details such as virtual IP addresses, iptables, DNS records, or environment variables. Service discovery from outside the cluster builds on top of the Service abstraction and focuses on exposing the Services to the outside world. While a `NodePort` provides the basics of exposing Services, a highly available setup requires integration with the platform infrastructure provider.

Table 13-1 summarizes the various ways service discovery is implemented in Kubernetes. This table aims to organize the various service discovery mechanisms in this chapter from more straightforward to more complex. We hope it can help you build a mental model and understand them better.

Table 13-1. Service Discovery mechanisms

Name	Configuration	Client type	Summary
ClusterIP	`type: ClusterIP` `.spec.selector`	Internal	The most common internal discovery mechanism
Manual IP	`type: ClusterIP` `kind: Endpoints`	Internal	External IP discovery
Manual FQDN	`type: ExternalName` `.spec.externalName`	Internal	External FQDN discovery
Headless Service	`type: ClusterIP` `.spec.clusterIP: None`	Internal	DNS-based discovery without a virtual IP
NodePort	`type: NodePort`	External	Preferred for non-HTTP traffic
LoadBalancer	`type: LoadBalancer`	External	Requires supporting cloud infrastructure
Ingress	`kind: Ingress`	External	L7/HTTP-based smart routing mechanism

This chapter gave a comprehensive overview of all the core concepts in Kubernetes for accessing and discovering services. However, the journey does not stop here. With the *Knative* project, new primitives on top of Kubernetes have been introduced, which help application developers with advanced serving and eventing.

In the context of the *Service Discovery* pattern, the *Knative Serving* subproject is of particular interest as it introduces a new Service resource with the same kind as the Services introduced here (but with a different API group). Knative Serving provides support for application revision but also for a very flexible scaling of services behind a load balancer. We give a short shout-out to Knative Serving in "Knative" on page 317, but a full discussion of Knative is beyond the scope of this book. In "More

Information" on page 333, you will find links that point to detailed information about Knative.

More Information

- Service Discovery Example
- Kubernetes Service
- DNS for Services and Pods
- Debug Services
- Using Source IP
- Create an External Load Balancer
- Ingress
- Kubernetes NodePort Versus LoadBalancer Versus Ingress? When Should I Use What?
- Kubernetes Ingress Versus OpenShift Route

CHAPTER 14

Self Awareness

Some applications need to be self-aware and require information about themselves. The *Self Awareness* pattern describes the Kubernetes *downward API* that provides a simple mechanism for introspection and metadata injection to applications.

Problem

For the majority of use cases, cloud native applications are stateless and disposable without an identity relevant to other applications. However, sometimes even these kinds of applications need to have information about themselves and the environment they are running in. That may include information known only at runtime, such as the Pod name, Pod IP address, and the hostname on which the application is placed. Or, other static information defined at Pod level such as the specific resource requests and limits, or some dynamic information such as annotations and labels that could be altered by the user at runtime.

For example, depending on the resources made available to the container, you may want to tune the application thread-pool size, or change the garbage collection algorithm or memory allocation. You may want to use the Pod name and the hostname while logging information, or while sending metrics to a central server. You may want to discover other Pods in the same namespace with a specific label and join them into a clustered application. For these and other use cases, Kubernetes provides the downward API.

Solution

The requirements that we've described and the following solution are not specific only to containers but are present in any dynamic environment where the metadata of resources changes. For example, AWS offers Instance Metadata and User Data services that can be queried from any EC2 instance to retrieve metadata about the EC2 instance itself. Similarly, AWS ECS provides APIs that can be queried by the containers and retrieve information about the container cluster.

The Kubernetes approach is even more elegant and easier to use. The *downward API* allows you to pass metadata about the Pod to the containers and the cluster through environment variables and files. These are the same mechanisms we used for passing application-related data from ConfigMaps and Secrets. But in this case, the data is not created by us. Instead, we specify the keys that interest us, and Kubernetes populates the values dynamically. Figure 14-1 gives an overview of how the downward API injects resource and runtime information into interested Pods.

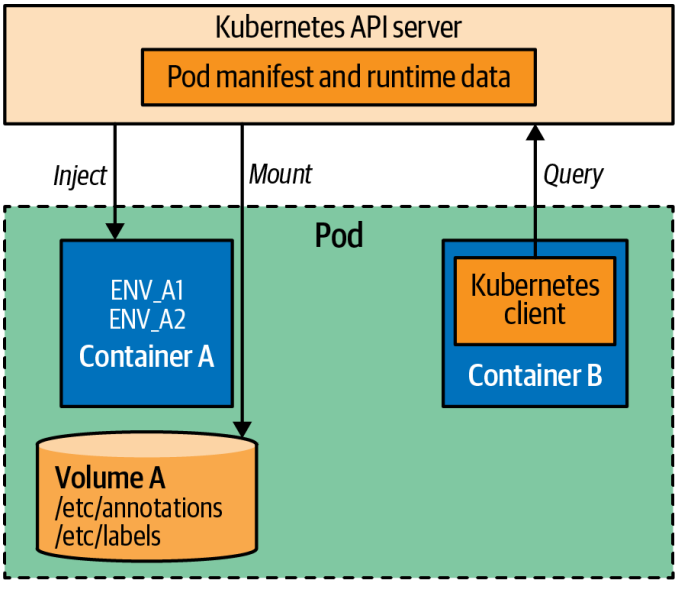

Figure 14-1. Application introspection mechanisms

The main point here is that with the downward API, the metadata is injected into your Pod and made available locally. The application does not need to use a client and interact with the Kubernetes API and can remain Kubernetes-agnostic. Let's see how easy it is to request metadata through environment variables in Example 14-1.

Example 14-1. Environment variables from downward API

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    env:
    - name: POD_IP
      valueFrom:
        fieldRef:                              ❶
          fieldPath: status.podIP
    - name: MEMORY_LIMIT
      valueFrom:
        resourceFieldRef:
          containerName: random-generator      ❷
          resource: limits.memory
```

❶ The environment variable `POD_IP` is set from the properties of this Pod and comes into existence at Pod startup time.

❷ The environment variable `MEMORY_LIMIT` is set to the value of the memory resource limit of this container; the actual limit declaration is not shown here.

In this example, we use `fieldRef` to access Pod-level metadata. The keys shown in Table 14-1 are available for `fieldRef.fieldPath` both as environment variables and `downwardAPI` volumes.

Table 14-1. Downward API information available in `fieldRef.fieldPath`

Name	Description
`spec.nodeName`	Name of node hosting the Pod
`status.hostIP`	IP address of node hosting the Pod
`metadata.name`	Pod name
`metadata.namespace`	Namespace in which the Pod is running
`status.podIP`	Pod IP address
`spec.serviceAccountName`	ServiceAccount that is used for the Pod
`metadata.uid`	Unique ID of the Pod
`metadata.labels['key']`	Value of the Pod's label *key*
`metadata.annotations['key']`	Value of the Pod's annotation *key*

As with `fieldRef`, we use `resourceFieldRef` to access metadata specific to a container's resource specification belonging to the Pod. This metadata is specific to a container and is specified with `resourceFieldRef.container`. When used as an environment variable, by default the current container is used. Possible keys for `resourceFieldRef.resource` are shown in Table 14-2. Resource declarations are explained in Chapter 2, "Predictable Demands".

Table 14-2. Downward API information available in `resourceFieldRef.resource`

Name	Description
`requests.cpu`	A container's CPU request
`limits.cpu`	A container's CPU limit
`requests.memory`	A container's memory request
`limits.memory`	A container's memory limit
`requests.hugepages-<size>`	A container's hugepages request (e.g., `requests.hugepages-1Gi`)
`limits.hugepages-<size>`	A container's hugepages limit (e.g., `limits.hugepages-1Gi`)
`requests.ephemeral-storage`	A container's ephemeral-storage request
`limits.ephemeral-storage`	A container's ephemeral-storage limit

A user can change certain metadata such as labels and annotations while a Pod is running. Unless the Pod is restarted, environment variables will not reflect such a change. But `downwardAPI` volumes can reflect updates to labels and annotations. In addition to the individual fields described previously, `downwardAPI` volumes can capture all Pod labels and annotations into files with `metadata.labels` and `metadata.annotations` references. Example 14-2 shows how such volumes can be used.

Example 14-2. Downward API through volumes

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    volumeMounts:
    - name: pod-info                    ❶
      mountPath: /pod-info
  volumes:
  - name: pod-info
    downwardAPI:
      items:
      - path: labels                    ❷
        fieldRef:
          fieldPath: metadata.labels
```

```
  - path: annotations                    ❸
    fieldRef:
      fieldPath: metadata.annotations
```

❶ Values from the downward API can be mounted as files into the Pod.

❷ The file `labels` contain all labels, line by line, in the format `name=value`. This file gets updated when labels are changing.

❸ The `annotations` file holds all annotations in the same format as the labels.

With volumes, if the metadata changes while the Pod is running, it is reflected in the volume files. But it is still up to the consuming application to detect the file change and read the updated data accordingly. If such a functionality is not implemented in the application, a Pod restart still might be required.

Discussion

Often, an application needs to be self-aware and have information about itself and the environment in which it is running. Kubernetes provides nonintrusive mechanisms for introspection and metadata injection. One of the downsides of the downward API is that it offers a fixed number of keys that can be referenced. If your application needs more data, especially about other resources or cluster-related metadata, it has to be queried on the API Server. This technique is used by many applications that query the API Server to discover other Pods in the same namespace that have certain labels or annotations. Then the application may form a cluster with the discovered Pods and sync state. It is also used by monitoring applications to discover Pods of interest and then start instrumenting them.

Many client libraries are available for different languages to interact with the Kubernetes API Server to obtain more self-referring information that goes beyond what the downward API provides.

More Information

- Self Awareness Example
- AWS EC2: Instance Metadata and User Data
- Expose Pod Information to Containers Through Files
- Expose Pod Information to Containers Through Environment Variables
- Downward API: Available Fields

PART III
Structural Patterns

Container images and containers are similar to classes and objects in the object-oriented world. Container images are the blueprint from which containers are instantiated. But these containers do not run in isolation; they run in other abstractions called Pods, where they interact with other containers.

The patterns in this category are focused on structuring and organizing containers in a Pod to satisfy different use cases. Pods provide unique runtime capabilities. The forces that affect containers in Pods result in the patterns discussed in the following chapters:

- Chapter 15, "Init Container", introduces a lifecycle for initialization-related tasks, decoupled from the main application responsibilities.
- Chapter 16, "Sidecar", describes how to extend and enhance the functionality of a preexisting container without changing it.
- Chapter 17, "Adapter", takes a heterogeneous system and makes it conform to a consistent unified interface that can be consumed by the outside world.
- Chapter 18, "Ambassador", describes a proxy that decouples access to external services.

CHAPTER 15

Init Container

The *Init Container* pattern enables separation of concerns by providing a separate lifecycle for initialization-related tasks distinct from the main application containers. In this chapter, we look closely at this fundamental Kubernetes concept that is used in many other patterns when initialization logic is required.

Problem

Initialization is a widespread concern in many programming languages. Some languages have it covered as part of the language, and some use naming conventions and patterns to indicate a construct as the initializer. For example, in the Java programming language, to instantiate an object that requires some setup, we use the constructor (or static blocks for fancier use cases). Constructors are guaranteed to run as the first thing within the object, and they are guaranteed to run only once by the managing runtime (this is just an example; we don't go into detail here on the different languages and corner cases). Moreover, we can use the constructor to validate preconditions such as mandatory parameters. We also use constructors to initialize the instance fields with incoming arguments or default values.

Init containers are similar but are at the Pod level rather than at the Java class level. So if you have one or more containers in a Pod that represent your main application, these containers may have prerequisites before starting up. These may include special permissions setup on the filesystem, database schema setup, or application seed data installation. Also, this initializing logic may require tools and libraries that cannot be included in the application image. For security reasons, the application image may not have permissions to perform the initializing activities. Alternatively, you may want to delay the startup of your application until an external dependency is satisfied. For all these kinds of use cases, Kubernetes uses init containers as implementation

of this pattern, which allow separation of initializing activities from the main application duties.

Solution

Init containers in Kubernetes are part of the Pod definition, and they separate all containers in a Pod into two groups: init containers and application containers. All init containers are executed in a sequence, one by one, and all of them have to terminate successfully before the application containers are started up. In that sense, init containers are like constructor instructions in a Java class that help object initialization. Application containers, on the other hand, run in parallel, and the startup order is arbitrary. The execution flow is demonstrated in Figure 15-1.

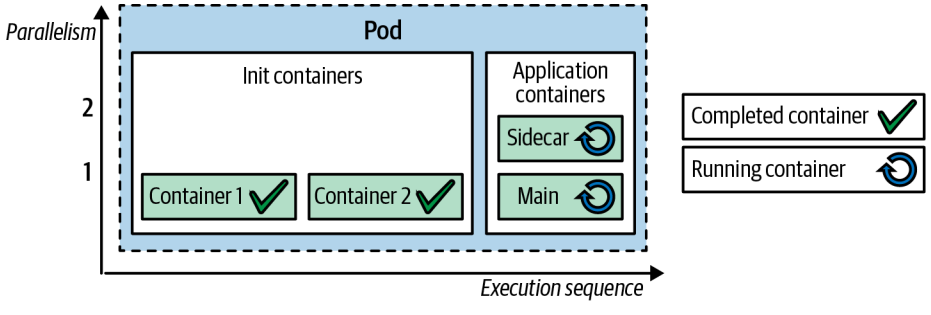

Figure 15-1. Init and application containers in a Pod

Typically, init containers are expected to be small, run quickly, and complete successfully, except when an init container is used to delay the start of a Pod while waiting for a dependency, in which case it may not terminate until the dependency is satisfied. If an init container fails, the whole Pod is restarted (unless it is marked with `RestartNever`), causing all init containers to run again. Thus, to prevent any side effects, making init containers idempotent is a good practice.

On one hand, init containers have all of the same capabilities as application containers: all of the containers are part of the same Pod, so they share resource limits, volumes, and security settings and end up placed on the same node. On the other hand, they have slightly different lifecycle, health-checking, and resource-handling semantics. There is no `livenessProbe`, `readinessProbe`, or `startupProbe` for init containers, as all init containers must terminate successfully before the Pod startup processes can continue with application containers.

Init containers also affect the way Pod resource requirements are calculated for scheduling, autoscaling, and quota management. Given the ordering in the execution of all containers in a Pod (first, init containers run a sequence, then all application

containers run in parallel), the effective Pod-level request and limit values become the highest values of the following two groups:

- The highest init container request/limit value
- The sum of all application container values for request/limit

A consequence of this behavior is that if you have init containers with high resource demands and application containers with low resource demands, the Pod-level request and limit values affecting the scheduling will be based on the higher value of the init containers, as demonstrated in Figure 15-2.

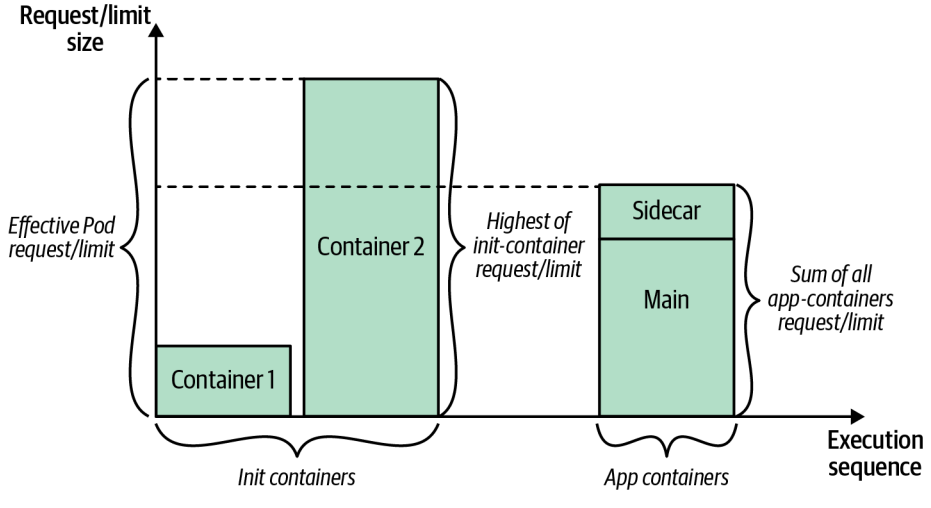

Figure 15-2. Effective Pod request/limit calculation

This setup is not resource-efficient. Even if init containers run for a short period of time and there is available capacity on the node for the majority of the time, no other Pod can use it.

Moreover, init containers enable separation of concerns and allow you to keep containers single-purposed. An application container can be created by the application engineer and focus on the application logic only. A deployment engineer can author an init container and focus on configuration and initialization tasks only. We demonstrate this in Example 15-1, which has one application container based on an HTTP server that serves files.

The container provides a generic HTTP-serving capability and does not make any assumptions about where the files to serve might come from for the different use cases. In the same Pod, an init container provides Git client capability, and its sole purpose is to clone a Git repo. Since both containers are part of the same Pod, they