KAPITEL 3

Ambassador

Im vorigen Kapitel wurde das Sidecar-Muster vorgestellt, bei dem ein Container einen bestehenden Container erweitert, um ihm mehr Funktionalität zu verschaffen. In diesem Kapitel geht es um das Ambassador-Muster. Hier kümmert sich ein Ambassador-Container um die Interaktionen zwischen den Anwendungs-Containern und dem Rest der Welt. Wie bei anderen Single-Node-Mustern befinden sich die beiden Container in einer symbiotischen Verbindung und sie sind auf einer einzelnen Maschine geschedult. Ein Diagramm dieses Musters sehen Sie in Abbildung 3-1.

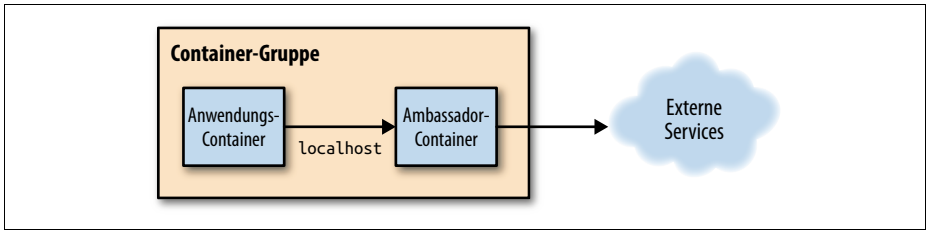

Abbildung 3-1: Generisches Ambassador-Muster

Das Ambassador-Muster bietet zwei Vorteile. Zum einen entsteht wie bei den anderen Single-Node-Mustern schon ein inhärenter Wert aus dem Erstellen modularer, wiederverwendbarer Container. Die Separation of Concerns sorgt dafür, dass sich die Container einfacher bauen und warten lassen. Und so kann auch der Ambassador-Container von vielen verschiedenen Anwendungs-Containern wiederverwendet werden. Das beschleunigt die Anwendungs-Entwicklung, weil der Code des Containers an vielen Stellen eingesetzt werden kann. Zusätzlich ist die Implementierung konsistenter und von höherer Qualität, weil sie einmal gebaut und in vielen Situationen genutzt wird.

Im Rest des Kapitels stelle ich eine Reihe von Beispielen für den Einsatz des Ambassador-Musters vor, indem ich eine Reihe realer Anwendungen implementiere.

Mit einem Ambassador einen Service per Sharding aufteilen

Manchmal werden die Daten, die Sie in einem Storage Layer ablegen wollen, für eine einzelne Maschine zu groß. In solchen Situationen müssen Sie Ihren Storage Layer *sharden*. Dabei wird er in mehrere getrennte Stücke (»Scherben«) aufgeteilt, die jeweils von einer eigenen Maschine betreut werden. Dieses Kapitel fokussiert sich auf ein Single-Node-Muster für das Anpassen eines bestehenden Service, damit dieser mit einem Sharded Service irgendwo in den Weiten des Webs kommunizieren kann. Es geht nicht darum, wie der Sharded Service selbst aufgesetzt wird. Sharding und ein Multi-Node-Entwurfsmuster für einen Sharded Service werden detailliert in Kapitel 6 beschrieben. Ein Diagramm solch eines Service sehen Sie in Abbildung 3-2.

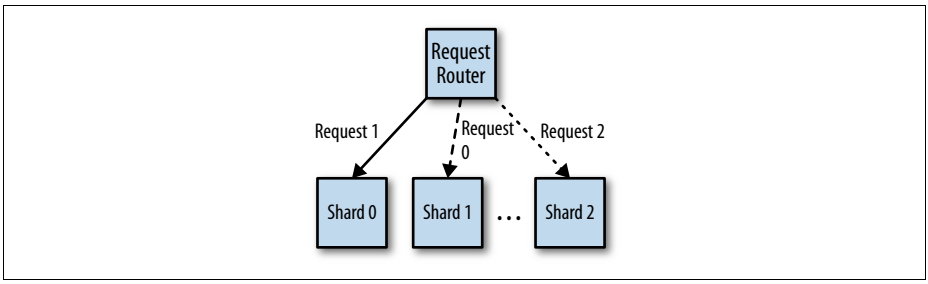

Abbildung 3-2: Ein generischer Sharded Service

Beim Deployen eines Sharded Service taucht unter anderem die Frage auf, wie man ihn mit dem Frontend- oder Middleware-Code zusammenführt, der die Daten speichert. Es muss dafür ganz klar eine Logik geben, die einen bestimmten Request an einen bestimmten Shard leitet, aber häufig ist es schwierig, solch einen Sharded Client in bestehendem Quellcode einzusetzen, wenn Letzterer davon ausgeht, dass er sich mit einem Single Storage Backend verbindet. Zudem erschweren Sharded Services es, Konfigurationsdaten für Entwicklungsumgebungen (in denen es oft nur einen Single Storage Shard gibt) und Produktivumgebungen (meist mit vielen Storage Shards) gemeinsam zu nutzen.

Ein Ansatz ist, die Shard-Logik in den Sharded Service selbst einzubauen. Dabei besitzt dieser ebenfalls einen zustandslosen Load Balancer, der den Verkehr an den passenden Shard weiterleitet. Somit handelt es sich bei diesem Load Balancer letztendlich auch um einen verteilten Ambassador-Service. Ein Ambassador auf Client-Seite wird damit überflüssig, dafür lässt sich der Sharded Service nur mit mehr Aufwand deployen. Die Alternative ist, einen Single-Node-Ambassador auf Seite des Clients zu integrieren, um den Verkehr an den passenden Shard zu leiten. Damit wird zwar das Deployen des Clients etwas komplizierter, aber es vereinfacht das Deployen des Sharded Service. Wie immer bei solchen Abwägungen hängt es von den Besonderheiten Ihrer spezifischen Anwendung ab, wel-

ches Vorgehen sinnvoller ist. Dabei ist unter anderem zu berücksichtigen, wo sich die Team-Grenzen in Ihrer Architektur befinden, aber auch, wo Sie selbst Code schreiben, statt nur fertige Software zu deployen. Letztendlich handelt es sich bei beiden Ansätzen um valide Alternativen. Der folgende Abschnitt beschreibt, wie Sie das Single-Node-Ambassador-Muster für ein clientseitiges Sharding nutzen.

Beim Anpassen einer bestehenden Anwendung an ein Sharded Backend können Sie einen Ambassador-Container einsetzen, der die Logik zum Weiterleiten von Requests an den entsprechenden Storage Shard besitzt. Damit muss sich Ihr Frontend oder die Middleware-Anwendung nur mit etwas verbinden, was wie ein Single-Storage-Backend auf localhost aussieht. Aber dieser Server ist in Wirklichkeit ein *Sharding Ambassador Proxy*, der alle Requests von Ihrem Anwendungscode erhält, diese an den passenden Storage Shard weiterleitet und das Ergebnis dann an Ihre Anwendung zurückgibt. Dieser Einsatz eines Ambassadors ist in Abbildung 3-2 dargestellt.

Als Ergebnis des Einsatzes des Ambassador-Musters auf einen Sharded Service erhalten Sie eine Separation of Concerns zwischen dem Anwendungs-Container, der nur weiß, dass er mit einem Storage Service reden muss, den er auf localhost findet, und dem Sharding Ambassador Proxy, der nur den Code enthält, der zum Umsetzen des richtigen Shardings notwendig ist. Wie bei allen guten Single-Node-Mustern lässt sich dieser Ambassador von vielen verschiedenen Anwendungen wiederverwenden. Oder es kann – wie Sie im folgenden Beispiel sehen werden – eine fertige Open-Source-Implementierung für den Ambassador zum Einsatz kommen, der die Entwicklung des gesamten verteilten Systems beschleunigt.

Aus der Praxis: Einen Sharded Redis implementieren

Redis ist ein schneller Key/Value-Store, der als Cache oder für einen persistenteren Storage zum Einsatz kommen kann. In diesem Beispiel werden wir ihn als Cache nutzen. Dabei beginnen wir mit dem Deployen eines Sharded Redis Service auf ein Kubernetes-Cluster. Dazu nutzen wir das API-Objekt `StatefulSet`, da wir damit eindeutige DNS-Namen für jeden Shard erhalten, die wir dann beim Konfigurieren des Proxys nutzen können.

Das `StatefulSet` für Redis sieht so aus:

```
apiVersion: apps/v1beta1
kind: StatefulSet
metadata:
  name: sharded-redis
spec:
  serviceName: "redis"
  replicas: 3
  template:
    metadata:
      labels:
        app: redis
    spec:
```

```
      terminationGracePeriodSeconds: 10
      containers:
      - name: redis
        image: redis
        ports:
        - containerPort: 6379
          name: redis
```

Sichern Sie dies in einer Datei namens *redis-shards.yaml* und deployen Sie sie mit `kubectl create -f redis-shards.yaml`. Damit werden drei Container erstellt, in denen Redis läuft. Mit `kubectl get pods` können Sie diese anzeigen, dabei sollte es die Einträge `sharded-redis-[0,1,2]` geben.

Natürlich reicht es nicht aus, diese Replicas laufen zu lassen – wir brauchen auch Namen, über die wir sie ansprechen können. Dafür nutzen wir einen Kubernetes-`Service`, der DNS-Namen für die erzeugten Replicas erstellt. Dieser sieht so aus:

```
apiVersion: v1
kind: Service
metadata:
  name: redis
  labels:
    app: redis
spec:
  ports:
  - port: 6379
    name: redis
  clusterIP: None
  selector:
    app: redis
```

Speichern Sie dies in einer Datei namens *redis-service.yaml* und deployen Sie das Ganze mit `kubectl create -f redis-service.yaml`. Sie sollten nun DNS-Einträge für `sharded-redis-0.redis`, `sharded-redis-1.redis` und so weiter erhalten. Mit diesen Namen können wir nun `twemproxy` konfigurieren. Dabei handelt es sich um einen schlanken, sehr performanten Proxy für `memcached` und `Redis`, der ursprünglich bei Twitter entwickelt wurde, mittlerweile Open Source ist und bei GitHub zur Verfügung steht *(https://github.com/twitter/twemproxy)*. Konfigurieren wir `twemproxy` so, dass er auf die von uns erstellten Replicas verweist:

```
redis:
  listen: 127.0.0.1:6379
  hash: fnv1a_64
  distribution: ketama
  auto_eject_hosts: true
  redis: true
  timeout: 400
  server_retry_timeout: 2000
  server_failure_limit: 1
  servers:
    - sharded-redis-0.redis:6379:1
    - sharded-redis-1.redis:6379:1
    - sharded-redis-2.redis:6379:1
```

In dieser Konfiguration können Sie sehen, dass wir das Redis-Protokoll an `localhost:6379` nutzen, sodass der Anwendungs-Container auf den Ambassador zugreifen kann. Wir werden diesen Pod mit einem `ConfigMap`-Objekt von Kubernetes deployen, den wir wie folgt erstellen:

```
kubectl create configmap --from-file=nutcracker.yaml
```

Schließlich sind alle Vorbereitungen getroffen und wir können unser Ambassador-Beispiel deployen. Dazu definieren wir einen Pod wie folgt:

```
apiVersion: v1
kind: Pod
metadata:
  name: ambassador-example
spec:
  containers:
    # Hier kommt der Anwendungs-Container hin, z. B.
    # - name: nginx
    #   image: nginx
    # Dies ist der Ambassador-Container
    - name: twemproxy
      image: ganomede/twemproxy
      command:
      - nutcracker
      - -c
      - /etc/config/nutcracker.yaml
      - -v
      - 7
      - -s
      - 6222
      volumeMounts:
      - name: config-volume
        mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: twem-config
```

Dieser Pod definiert den Ambassador, der Anwendungs-Container des Benutzers lässt sich dann hinzufügen, um den Pod zu vervollständigen.

Einen Ambassador zum Service Brokering einsetzen

Beim Versuch, eine Anwendung über mehrere Umgebungen hinweg laufen lassen zu können (zum Beispiel in der Public Cloud, direkt in einem Datacenter oder in einer Private Cloud), ist eine der größten Herausforderungen an das Service Discovery und die Konfiguration. Um zu verstehen, was das heißt, stellen Sie sich ein Frontend vor, das auf einer MySQL-Datenbank zum Abspeichern der Daten aufbaut. In der Public Cloud könnte dieser MySQL-Service als Software-as-a-Service (SaaS) bereitgestellt sein, während es in einer Private Cloud eventuell notwendig ist, eine neue virtuelle Maschine oder einen Container mit MySQL zu starten.

Daher ist es zum Bauen einer portablen Anwendung notwendig, dass sie weiß, wie sie ihre Umgebung untersuchen und den passenden MySQL-Service zum Verbinden finden kann. Dieser Prozess wird als *Service Discovery* bezeichnet und das System, das das Suchen und Verlinken erledigt, nennt man meist einen *Service Broker*. Wie in vorigen Beispielen ermöglicht es das Ambassador-Muster einem System, die Logik des Anwendungs-Containers von der Logik des Service Broker Ambassadors zu trennen. Die Anwendung verbindet sich einfach immer mit einer Instanz des Service (zum Beispiel MySQL), die auf localhost läuft. Es liegt in der Verantwortung des Service Broker Ambassadors, seine Umgebung zu untersuchen und die passende Verbindung weiterzuleiten. Dieser Prozess ist in Abbildung 3-3 dargestellt.

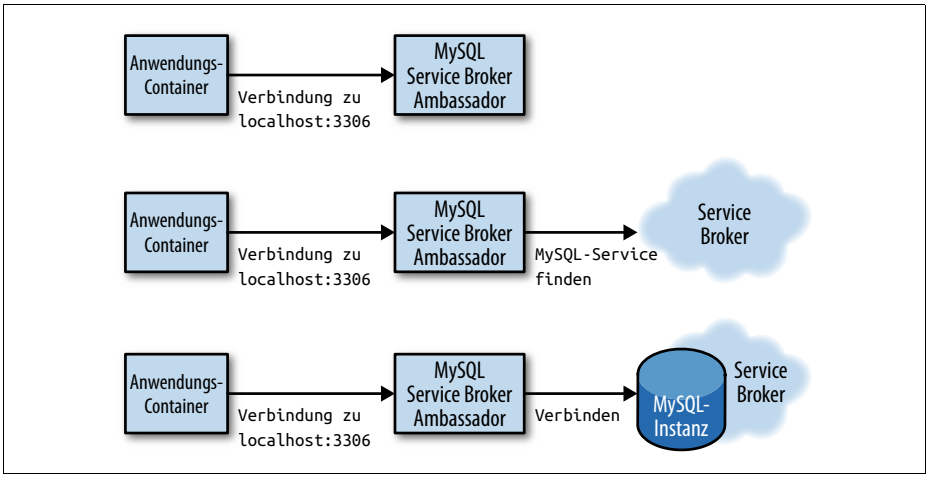

Abbildung 3-3: Ein Service Broker Ambassador, der einen MySQL-Service erstellt

Mit einem Ambassador experimentieren oder Requests splitten

Eine letzte Beispielanwendung des Ambassador-Musters soll zeigen, wie man Experimente oder andere Formen des Request Splittings durchführen kann. In vielen Produktiv-Systemen ist es von Vorteil, solche Splits vornehmen zu können. Dabei wird ein gewisser Anteil aller Requests nicht vom eigentlichen Produktiv-Service behandelt, sondern an eine andere Implementierung des Service umgeleitet. Meist dient das zum Durchführen von Experimenten mit neuen Beta-Versionen des Service, um herauszufinden, ob die neue Version der Software zuverlässig oder aus Performancesicht vergleichbar mit der aktuell deployten Version ist.

Zudem wird Request Splitting manchmal genutzt, um den Verkehr sowohl an das Produktiv-System als auch an eine neuere, noch nicht deployte Version zu leiten. Die Responses vom Produktiv-System werden an den Anwender zurückgeschickt,

während die vom neuen Service ignoriert werden. Dann dient das Splitting dazu, eine produktive Last auf der neuen Version des Service zu simulieren, ohne Auswirkungen auf bestehende Produktiv-Benutzer befürchten zu müssen.

Basierend auf den vorigen Beispielen ist recht klar, wie ein Request-Splitting-Ambassador mit einem Anwendungs-Container interagieren kann, um das Request-Splitting zu implementieren. Wie zuvor verbindet sich der Anwendungs-Container mit dem Service einfach auf localhost, während der Ambassador-Container die Requests empfängt, sie sowohl an das Produktiv- wie auch das Experimental-System weiterleitet und dann die Produktiv-Responses zurückliefert, als ob er die Arbeit selbst erledigt hätte.

Diese Separation of Concerns sorgt dafür, dass der Code in jedem Container schlank und fokussiert bleibt, während das modulare Faktorieren der Anwendung sicherstellt, dass der Request-Splitting-Ambassador für eine Vielzahl verschiedener Anwendungen und Situationen eingesetzt werden kann.

Aus der Praxis: 10%-Experimente umsetzen

Um unser Request-Splitting-Experiment zu implementieren, werden wir den Webserver nginx nutzen. Dabei handelt es sich um einen mächtigen, umfangreich ausgestatteten Open-Source-Server. Um ngninx als Ambassador zu konfigurieren, werden wir die folgende Konfiguration nutzen (beachten Sie, dass dies für HTTP ist, sich aber für HTTPS auch einfach anpassen lässt).

```
worker_processes  5;
error_log  error.log;
pid        nginx.pid;
worker_rlimit_nofile 8192;

events {
  worker_connections  1024;
}

http {
    upstream backend {
        ip_hash;
        server web weight=9;
        server experiment;
    }

    server {
        listen localhost:80;
        location / {
            proxy_pass http://backend;
        }
    }
}
```

Wie beim zuvor vorgestellten Sharded Service ist es möglich, das Experimental-Framework als eigenen Microservice zu deployen, statt es als Teil Ihres Client-Pods umzusetzen. Natürlich erhalten Sie so einen weiteren Service, der gewartet, skaliert und überwacht werden muss. Wenn es sich bei diesen Experimenten um eine langfristige Komponente in Ihrer Architektur handelt, kann sich das lohnen. Werden sie nur gelegentlich genutzt, ist ein clientseitiger Ambassador vermutlich sinnvoller.

Sie werden sehen, dass ich in dieser Konfiguration IP-Hashing nutze. Das ist wichtig, weil so sichergestellt ist, dass der Anwender nicht zwischen Experiment und eigentlichem Service hin und her wechselt. Dadurch hat er eine konsistente Reaktion durch die Anwendung.

Der Parameter `weight` wird genutzt, um 90% des Verkehrs auf die bestehende Anwendung zu leiten, während 10% an das Experiment gehen.

Wie in anderen Beispielen werden wir diese Konfiguration als `ConfigMap`-Objekt in Kubernetes deployen:

```
kubectl create configmaps --from-file=nginx.conf
```

Ich gehe hier davon aus, dass Sie einen Service mit dem Namen `web` und einen namens `experiment` definiert haben. Wenn nicht, müssen Sie sie zuerst anlegen, bevor Sie versuchen, den Ambassador-Container zu erzeugen, da es nginx nicht mag, gestartet zu werden, wenn die Ziel-Services nicht gefunden werden. Hier ein paar Beispiel-Konfigurationen für die Services:

```
# Dies ist der Service 'experiment'
apiVersion: v1
kind: Service
metadata:
  name: experiment
  labels:
    app: experiment
spec:
  ports:
  - port: 80
    name: web
  selector:
    # Ändern Sie diesen Selektor, damit er zu den Labels
    # Ihrer Anwendung passt
    app: experiment
---
# Dies ist der Service 'web'
apiVersion: v1
kind: Service
metadata:
  name: web
  labels:
    app: web
```

```
spec:
  ports:
  - port: 80
    name: web
  selector:
    # Ändern Sie diesen Selektor, damit er zu den Labels
    # Ihrer Anwendung passt
    app: web
```

Nun deployen wir nginx selbst als Ambassador-Container in einem Pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: experiment-example
spec:
  containers:
    # Hier kommt der Anwendungs-Container hin, z. B.
    # - name: some-name
    #   image: some-image
    # Dies ist der Ambassador-Container
    - name: nginx
      image: nginx
      volumeMounts:
      - name: config-volume
        mountPath: /etc/nginx
  volumes:
    - name: config-volume
      configMap:
        name: experiment-config
```

Sie können einen zweiten (oder dritten oder vierten) Container zum Pod hinzufügen, um die Vorteile des Ambassadors wirklich auszureizen.