Using Ambassador to Build Cloud-Native Applications

Steve Flanders

CloudNativeCon China, Shanghai 2019
About Me

Steve Flanders
Head of Product and Experience at Omnition
@smflanders | https://sflanders.net
Agenda

● Background
  ○ The move from monolith to microservice-based architecture
  ○ An introduction to API gateways
  ○ Kubernetes and what it provides

● Ambassador
  ○ What is it?
  ○ How does it work?
  ○ What does it provide?

● Cloud-Native Applications
  ○ API gateway use-cases
  ○ Real world examples
  ○ Configuration examples
  ○ Lessons learned
Background
Monolith to microservices

API Gateway
Kubernetes and what it provides

From https://kubernetes.io/: Kubernetes (K8s) is an open-source system for automating deployment, scaling, and management of containerized applications.

```yaml
---
apiVersion: v1
kind: Service
metadata:
  name: ambassador
spec:
  type: LoadBalancer
  externalTrafficPolicy: Local
  ports:
  - port: 80
    targetPort: 8080
  selector:
    service: ambassador
```
Ambassador
What is it?

Ambassador
- An open source, Kubernetes-native microservices API gateway built on the Envoy Proxy.

Envoy
- An open source edge and service proxy, designed for cloud-native applications.

Put another way: Envoy is a “data plane” and Ambassador is a “control plane”
How does it work?

1. Service owner defines configuration in Kubernetes manifests.
2. Kubernetes API notifies Ambassador of manifest changes.
3. Ambassador parses the change and transforms the configuration to a semantic intermediate representation. Envoy configuration is generated from this IR.
4. The new configuration is passed to Envoy via the gRPC-based Aggregated Discovery Service (ADS) API.
5. Traffic flows through the reconfigured Envoy, without dropping any connections.
What does it provide?

Support for:

● Self-Service via Kubernetes Annotations
● Kubernetes-Native Architecture
● Istio Integration
● Flexible Canary Deployments

Features:

● gRPC and HTTP/2 Support
● Authentication
● Rate Limiting
● Integrated Diagnostics
How is it different?

Alternatives fall in three basic categories:

1. Hosted API gateways (e.g. Amazon API gateway)
2. Traditional API gateways (e.g. Kong)
3. L7 proxies (e.g. Traefik, NGINX, HAProxy, or Envoy, or Ingress controllers built on these proxies)

Ambassador differences:

- No vendor lock-in (e.g. hosted gateways)
- No dependency on external database (e.g. Kong)
- Self service and Kubernetes native
- Leverages Envoy
Cloud-Native Applications
Use-case 1: Edge (North/South) Routing

- Ability to control/route ingress traffic
- Offload requirements such as
  - Authentication (e.g. require all ingress traffic to be authenticated)
  - Encryption (e.g. TLS termination and pass-through)
  - Retries and timeouts
Real World Example: Monolith to microservices
Real World Example: Monolith to microservices
Real World Example: Monolith to microservices
Real World Example: SaaS Service

- Ingestion
- Query
- Auth
- Data
- Ingest Path
- Query Path
- SaaS Service
Understanding TLS

Client
root cert

Encrypted Traffic

cert chain
public key

Server

Unencrypted Traffic

Client
root cert

Encrypted Traffic

Encrypted Traffic

Server

cert chain
private key

self-signed cert chain
self-signed private key
Understanding TLS and SNI

End-user (Internet) → Kubernetes cluster

- Auth Service
  - Datawire site service
  - Shared service
  - Ambassador site service

HOSTS
- www.datawire.io
- www.getambassador.io

TLS cert

Stored as Kubernetes secrets
Use-case 2: Internal (North/South) Routing

- Ability to control/route multi-tenant traffic
- Offload requirements such as
  - Mapping (e.g., based on headers)
  - Retries and timeouts
Real World Example: SaaS Service

Ingestion

Query

Auth

Query Path

Ingest Path

Data

Query Path

Ingest Path

Data

SaaS Service
Real World Example: SaaS Service

Ingestion → Auth → Ingest Path

Query → Auth → Query Path

Data

Query Path

Ingest Path

Data

SaaS Service
Use-case 3: Internal (East/West) Routing

- Ability to control/route any traffic
- Offload requirements such as
  - Service discovery
  - Load balancing
  - Access control
Real World Example: SaaS Service

- Analytics Service
- Ingest Service
- Query Service
- Billing Service
- Auth Service

K8s node 1

K8s node 2
Real World Example: Service Mesh

Service Mesh: A network of services that allows communication between different applications. It provides a way to route traffic, secure communication, and manage applications at scale.

Components:
- Query Service
- Billing Service
- Auth Service
- Istio
- SaaS service

Istio and Envoy are key components in a service mesh, providing traffic management, security, and observability features.
Use-case 4: Traffic Shadowing

- Ability to test code/releases
- Leverage real traffic/load
- Minimize duplicated resources
Use-case 5: Software Development Testing with Telepresence (paid)

- Ability to test in production
- Ability to leverage production without impacting production

1. Jane sends request to Ambassador.
2. Ambassador authenticates request.
3. Telepresence proxies request to Jane’s laptop.
4. Bob, who works with Jane, only sees Jane’s committed code.
Configuration
Options by version

- < 0.50.0
  - Configmaps
  - Annotations

- >= 0.50.0
  - Annotations

- >= 0.70.0
  - Annotations
  - CRDs
Encryption

apiVersion: v1
kind: Service
metadata:
  annotations:
    getambassador.io/config: |
      ---
      apiVersion: ambassador/v0
      kind: Module
      name: tls
      ambassador_id: myID
      config:
        server:
          enabled: True
      ...

 CRDs (0.70.0 and NEWER)

apiVersion: getambassador.io/v1
kind: Module
metadata:
  name: tls
  namespace: default
spec:
  ambassador_id: myID
  config:
    server:
      enabled: True
      ...

Annotations (0.70.0 and older)
Authentication

---

```yaml
apiVersion: v1
kind: Service
metadata:
  annotations:
    getambassador.io/config: |
      ---
      apiVersion: ambassador/v0
      ambassador_id: myID
      kind: AuthService
      name: authentication
      auth_service: "auth:3000"
      path_prefix: "/auth/api/check"
      ---
      apiVersion: ambassador/v0
      kind: Mapping
      ambassador_id: myID
      prefix: /auth/
      rewrite: /auth/
      service: auth:3000
```

---

```yaml
apiVersion: getambassador.io/v1
kind: AuthService
metadata:
  name: authentication
  namespace: default
spec:
  ambassador_id: myID
  auth_service: "auth:3000"
  path_prefix: "/auth/api/check"
---

apiVersion: getambassador.io/v1
kind: Mapping
metadata:
  name: auth-mapping
  namespace: default
spec:
  ambassador_id: myID
  prefix: /auth/
  rewrite: /auth/
  service: auth:3000
```
Mapping

apiVersion: v1
type: Service
metadata:
  annotations:
    getambassador.io/config: |
      ---
    apiVersion: ambassador/v0
    type: Mapping
    name: omnition-mapping
    ambassador_id: myID
    prefix: /
    host: omnition.io
    service: web.default.svc.cluster.local

CRDs (0.7.0 and NEWER)

apiVersion: getambassador.io/v1
type: Mapping
metadata:
  name: omnition-mapping
  namespace: default
spec:
  ambassador_id: myID
  prefix: /
  host: omnition.io
  service: web.default.svc.cluster.local

...
Tracing

Annotations (0.70.0 and older)

```yaml
apiVersion: v1
kind: Service
metadata:
  annotations:
    getambassador.io/config: |
      ---
      apiVersion: ambassador/v0
      kind: TracingService
      name: tracing
      ambassador_id: myID
      service: collector.default:9411
      driver: zipkin

...```

CRDs (0.70.0 and NEWER)

```yaml
apiVersion: getambassador.io/v1
kind: TracingService
metadata:
  name: tracing
  namespace: default
spec:
  ambassador_id: myID
  service: collector.default:9411
  driver: zipkin
...```
Lessons Learned
Dedicate Ambassadors per responsibility

- While this is possible and supported it is not ideal (e.g. availability, performance, etc)
- In some circumstances this is not possible (e.g. exposing 80 and 443 from the same Ambassador instance)
Dedicate Ambassadors per responsibility

- Different Ambassador “clusters” can be created via unique ambassador_id
- Each can be configured and scaled independently
Understand timeouts

AWS ELB settings:

- `service.beta.kubernetes.io/aws-load-balancer-healthcheck-interval`: "5"
- `service.beta.kubernetes.io/aws-load-balancer-healthcheck-timeout`: "3"
- `service.beta.kubernetes.io/aws-load-balancer-healthcheck-unhealthy-threshold`: "2"
- `service.beta.kubernetes.io/aws-load-balancer-healthcheck-healthy-threshold`: "2"

Ambassador timeout settings:

- Request timeout: `timeout_ms` (default = 3000ms)
- Idle timeout: `idle_timeout_ms` (default = 300000ms)
- Connect timeout: `connect_timeout_ms`
Understand service discovery and load balancing

- Typically resolve to k8s service via DNS
  - Use: <service>
  - Use: <service>.<namespace>
  - Not: <service>.<namespace>.svc.cluster.local
- Alternatives
  - K8s endpoint routing
  - Consul routing

- K8s service load balancing
- Alternatives
  - Round robin
  - Ring hash
  - Maglev
HTTP 503s

Converter Service

Converter Pod

Converter Pod

Converter Pod
Others

- Great start, but work/money required
  - How will you handle authentication?
  - How will you handle circuit breaking?
  - How will you handle per-service retries?

- Store configuration with your service
  - Exception for authentication service
  - Everything through CI/CD

- Edge routing is much harder than internal
  - Encryption: end-to-end or terminated?
  - Authentication: which headers?

- Observability - critical, but immature
  - Tracing differs between releases (Envoy)
  - Metrics are not as granular (Envoy)
  - Diagnostic UI is basic and ugly
Questions?