Microservices

Design Patterns

→ Design patterns provide reusable solutions to commonly recurring problems.
→ In microservices, they address challenges related to distributed system deployment, scaling, communication, and more.

Key Microservice Patterns

1) Single Service Pattern
- Each functionality or module as a standalone service
- Ensures single responsibility, easier to scale and maintain.
- In an e-commerce site, Order Management, User Profile, and Inventory could be 3 different microservice.
- Should not be used when services have very high inter-dependencies.

2) API Gateway Pattern
- A service that provides a single entry point for certain group of microservices.
- Ideal for external consumers, providing a unified API experience.
- Public clients might use a different API Gateway than web clients in a large-scale application, allowing for specialized optimization.
- When you have very few services, direct access might be simpler.

3) Saga Pattern
- Instead of a single transaction that encompasses multiple services, each service involved in the workflow about a saga.
- If a service in the saga fails, compensating transactions run to correct the situation.
- Handling long-running business processes and ensuring data consistency.
- An online placement system where payment, inventory check, and delivery scheduling are different sagas.
- Should not be used when simple, simpler three atomic transaction suffice.

4) Circuit Breaker Pattern
- Prevents system failures by opening the circuit if downstream service calls fail a certain number of times.
- Ensuring system resilience

- A recommendation system falls back to default recommendation if the personalized model fails.
- For non-critical services where occasional failures don't significantly impact user experience.

5) Event Sourcing Pattern
- Stores all changes to the application state as a sequence of events.
- All reconstructing system state, rather event replay.
- An audit log system that logs all user actions to replay them on case of issues.
- Where the overhead of event storage and processing outweigh the benefit.

Optimization Techniques

1) Service Mesh
- A dedicated layer for handling service-to-service communication in microservices architecture, or Kubernetes the complexity of these communications into a configurable infrastructure layer.
- Traffic Management: Load balancing, rate limiting, and failover.
  Security: mTLS for encrypted service-to-service communication
  Observability: Metrics, monitoring, and tracing for services.
- Example: Istio
- Practical Use-Cases:
  - Travel Booking Platform:
    Imagine a platform like Expedia, with services like flight booking, hotel reservations, car rentals, etc.
    Istio can help manage traffic between the services, ensure secure communication, and provide insight into bottlenecks or issues.
  - Netflix
    Scenario: Suppose Netflix wants to roll out a new recommendation algorithm. With a service mesh in place, they can easily do a canary release.
    How Service Mesh Helps:
    With service mesh, Netflix can direct a small portion of user traffic to service running the new algorithm.
    Mesh provides telemetry data to monitor how the new algorithm is performing.
    If an anomaly is detected, traffic can be immediately reverted back to the old version.
  - Amazon
    Scenario: During Black Friday, Amazon's order and inventory services are under extreme load.
    How service Mesh Helps:
    → Service mesh helps in auto-scaling instances of these services based on the demand.
    → The traffic management feature ensures requests are distributed evenly.
    → If a service overload fails, service mesh reroutes the traffic to healthy instances, ensuring high availability.

- Service Mesh vs. API Gateway

Category	Service Mesh	API Gateway
1. Scope	Primarily concerned with service-to-service communications within a particular environment (e.g., within Kubernetes cluster). It deals with the east-west traffic flow.	Sits at the edge of the system and manages external requests coming into our application stack. It's responsible for request routing, composition, and API exposure. It deals with the north-south traffic flow.
2. Functionality	Focuses on load balancing, retries, failures, and monitoring, without needing any changes to the underlying code.	Can transform requests, handle protocol translation, manage quotas, do rate limiting, and often have built-in developer portals.
3. Complexity	More granular, as it operates at the service level.	Operates at a higher, service aggregate level.

Istio with Spring Microservices

↳ Istio provides a lot of functions on top of Kubernetes and integrate well with Spring Boot.

↳ Steps:

1) Setup Istio on Kubernetes

```
$ curl -L https://istio.io/downloadIstio | sh -
Downloading 1.22.0 from
https://github.com/istio/istio/releases/download/1.22.0/istio-1.22.0-osx.tar.gz ...
Istio 1.22.0 Download complete!

Add the istio-1.22.0 bin directory to your path; for instance, run:
  export PATH="$PATH:/Users/sam/istio-1.22.0/bin"

The command, istioctl, is now available in your bin directory.
Refer to https://istio.io/docs/setup/kubernetes/ to install Istio in your cluster.
$ istioctl install --set profile=demo -y
✔ Istio core installed
✔ Istiod installed
✔ Egressgateway installed
✔ Ingressgateway installed
✔ Installation complete
Thank you for installing Istio 1.22.0.  Enjoy!
$ kubectl get pod -n istio-system
NAME                                    READY   STATUS    RESTARTS   AGE
istio-egressgateway-6d9d8b7b9-ldqzq     1/1     Running   0          2m
istio-ingressgateway-5f7d8c9b4-mnpqr    1/1     Running   0          2m
istiod-7b4d9c8f6-xkrtv                  1/1     Running   0          3m
$ kubectl get svc -n istio-system
NAME                   TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)                                      AGE
istio-egressgateway    ClusterIP      10.96.12.34    <none>        80/TCP,443/TCP,15443/TCP                     2m
istio-ingressgateway   LoadBalancer   10.96.45.67    <pending>     15021:31021/TCP,80:31080/TCP,443:31443/TCP   2m
istiod                 ClusterIP      10.96.78.90    <none>        15010/TCP,15012/TCP,443/TCP,15014/TCP        3m
$ kubectl get ns --show-labels
NAME              STATUS   AGE   LABELS
default           Active   5d    kubernetes.io/metadata.name=default
istio-system      Active   2m    kubernetes.io/metadata.name=istio-system,istio-injection=disabled
kube-system       Active   5d    kubernetes.io/metadata.name=kube-system
kube-public       Active   5d    kubernetes.io/metadata.name=kube-public
kube-node-lease   Active   5d    kubernetes.io/metadata.name=kube-node-lease
$ kubectl label namespace default istio-injection=enabled
namespace/default labeled
$ kubectl get ns default --show-labels
NAME      STATUS   AGE   LABELS
default   Active   5d    kubernetes.io/metadata.name=default,istio-injection=enabled
$ kubectl apply -f sample-app.yaml
deployment.apps/spring-boot-service created
service/spring-boot-service created
deployment.apps/spring-boot-client created
service/spring-boot-client created
$ kubectl get pod -n default
NAME                                 READY   STATUS    RESTARTS   AGE
spring-boot-client-7c9d4f8b6-2xqrt   2/2     Running   0          30s
spring-boot-client-7c9d4f8b6-9mnpq   2/2     Running   0          30s
spring-boot-client-7c9d4f8b6-kvwzx   2/2     Running   0          30s
spring-boot-service-5b8c7d9f4-4hjkl  2/2     Running   0          30s
spring-boot-service-5b8c7d9f4-7rstu  2/2     Running   0          30s
spring-boot-service-5b8c7d9f4-vwxyz  2/2     Running   0          30s
$ kubectl describe pod spring-boot-service-5b8c7d9f4-4hjkl -n default | grep -A 5 Containers:
  Containers:
    spring-boot-service:
      Container ID:  containerd://a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6q7r8s9t0u1v2w3x4y5z6
      Image:         docker.io/sample/spring-boot-service:latest
      Image ID:      docker.io/sample/spring-boot-service@sha256:8f4a2b9c7d3e1f6a5b0c4d8e2f7a9b3c6d1e5f0a4b8c2d7e9f3a6b1c5d0e4f8a
      Port:          8080/TCP
      Host Port:     0/TCP
      State:         Running
$ kubectl exp
```

2) Enable Your Namespace to Auto-Inject Istio sidecars proxies

3) Deploy Spring Boot Microservice
Istio sidecar will be automatically added.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-spring-boot-service
  labels:
    app: spring-boot
    version: v1
spec:
  replicas: 3
  selector:
    matchLabels:
      app: spring-boot
      version: v1
  template:
    metadata:
      labels:
      app: spring-boot
      version: v1
      annotations:
      sidecar.istio.io/inject: true
    spec:
      containers:
      - name: spring-boot
        image: my-spring-boot-image:latest
        ports:
        - containerPort: 8080
```

4) Istio Traffic Management
For example, to redirect 80% of traffic to v1 of our service and 20% to v2:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: my-spring-boot-route
spec:
  hosts:
  - my-spring-boot-service
  http:
  - route:
    - destination:
        host: my-spring-boot-service
        subset: v1
      weight: 80
    - destination:
        host: my-spring-boot-service
        subset: v2
      weight: 20
```

↳ Istio offers a multitude of features like circuit breaking, retries, fault injection, etc.

→ Possible Use-Cases of Service Mesh
1) Canary Deployments
2) Fault Injection
3) Blue-Green Deployments

2) Containerization (Docker) & Orchestration (Kubernetes)

Practical Use-Case
For platforms like Amazon, there's a need for high availability during sales. Docker ensures consistent application environments, while Kubernetes scales services as per demand.

Possible Use-Cases
1) Auto scaling
2) Self-healing
3) Rolling Updates & Rollbacks

3) Centralized Logging (e.g., ELK Stack)
→ A platform for examining, analyzing, and visualizing log data in real-time.
- Elasticsearch: Search and analytics engine
- Logstash: Data processing pipeline
- Kibana: Visualization layer

Practical Use-Case
Streaming platforms like Netflix benefit from centralized logging as it helps in quick error detection, user activity tracking, and performance monitoring when some stream-related issues are multiple microservices at play.

Other Common Use-Cases
1) Monitoring
2) Debugging
3) Auditing