# Microservices Deep Dive: API Gateway, Data, Security, Monitoring

# Module 1: API Gateway – Architecture & Patterns

## 1.1 Introduction to API Gateway

* **Definition:** Acts as a single entry point for client requests into microservices, centralizing cross-cutting concerns.
* **Benefits:**
  + Simplifies client interactions.
  + Offloads authentication, logging, etc.
  + Enables API composition & versioning.
  + Enhances security and resilience.

## 1.2 Core Architecture

* **Reverse Proxy/Gateway Routing:** Forwards requests to correct microservice.
* **Request Aggregation:** Combines results from several service calls into a single response.
* **Cross-Cutting Concerns Offloading:** Deals with authentication, SSL, rate limiting, caching.

## 1.3 Common Patterns

| **Pattern** | **Description** | **Example Tools** |
| --- | --- | --- |
| Centralized Edge Gateway | Single entry point for all requests | Zuul, Spring Cloud Gateway |
| Two-Tier Gateway | Separates client-facing from back-end gateway for scalability/security | Ocelot, API Gateway |
| Microgateway | Smaller proxies for individual services | Istio, Linkerd |
| Sidecar Gateway/Service Mesh | Lightweight proxies alongside each service, typically with service mesh | Istio, Envoy |

## 1.4 Implementation Demos

* **Spring Cloud Gateway Example:** Route, predicate, filter configuration.
* **Zuul Example:** Routing with Netflix OSS stack, integrating with service discovery.

## 1.5 Real-world Considerations

* **Performance (caching, rate limiting, load balancing).**
* **Security (token validation, SSL termination, IP allowlisting).**
* **Observability (logging/tracing at the Gateway layer).**

# Module 2: Data Management in Microservices

## 2.1 Data Management Challenges

* **Why is data hard in microservices?**
  + Distributed ownership of data.
  + Difficulties maintaining consistency.
  + Need for independent schema evolution.

## 2.2 Event Sourcing

* **Pattern Overview:** State is stored as a series of events rather than as the current state.
* **Benefits:** Full history/audit trail, atomic updates, enables CQRS.
* **Drawbacks:** Complex queries for current state, increased storage needs.

## 2.3 Command Query Responsibility Segregation (CQRS)

* **Pattern Overview:** Segregates write (command) models from read (query) models.
* **Read Model:** Denormalized and optimized for queries, fast and scalable.
* **Write Model:** Only handles commands/events, storing actions in event store.
* **Use-cases:** High read/write load, flexibility in data models per use-case.

## 2.4 Eventual Consistency & Distributed Transactions

* **Saga Pattern:** Choreography and orchestration to coordinate distributed operations.
* **Handling Inconsistency:** Retries, idempotency keys, compensating transactions.
* **Messaging Platforms:** Kafka, RabbitMQ for event-driven consistency.

## 2.5 Practical Considerations

* **Choosing schemas, synchronizing events, handling duplication.**
* **Database technologies:** NoSQL for read models, event stores for write models.

# Module 3: Security in Microservices

## 3.1 Security in Distributed Systems

* **API as the attack surface.**
* **Centralized vs. distributed security enforcement.**

## 3.2 OAuth2 & JWT

* **OAuth2:** Authorization framework allowing limited delegated access.
  + Client, resource owner, authorization server, resource server.
* **JWT:** Self-contained, signed token for stateless authentication. Contains user data, expiry, roles in claims.
* **Usage in Microservices:** Token-based, decentralized validation, single sign-on.

## 3.3 Implementing Security

* **Spring Security:**
  + Secures REST endpoints, configures authentication/authorization.
  + Integrates with OAuth2/JWT, centralizes user management.
* **API Gateway Role:** Verifies tokens, forwards verified requests, protects downstream services.

## 3.4 Common Practices

* **Role-based access control.**
* **Service-to-service authentication (mutual TLS, JWT).**
* **Securing service discovery and configuration servers.**

## 3.5 Hands-On Demo Ideas

* Protecting microservices using JWT with Spring Security.
* Integrating Keycloak or Auth0 for OAuth2 flows.
* Customizing authorization via claims.

# Module 4: Monitoring & Logging in Microservices

## 4.1 Why Observability Matters

* Distributed nature means traditional logging=insufficient.
* Must trace requests, spot issues, monitor performance.

## 4.2 Centralized Logging

* **ELK Stack (Elasticsearch, Logstash, Kibana):**
  + Aggregates, indexes, and visualizes logs across all services.
  + Structured logging (e.g., JSON), log collectors (Logstash, Fluentd), visualization (Kibana).
* **Best Practices:**
  + Uniform log format.
  + Avoid logging sensitive information.
  + Use correlation IDs.

## 4.3 Metrics & Monitoring

* **Prometheus:** Metrics collection and alerting.
* **Grafana:** Visual dashboards for real-time metrics.
* **Service Health Checks:** Expose endpoints for liveness/readiness.

## 4.4 Distributed Tracing

* **Tools:** Zipkin, Jaeger for end-to-end tracing across services.
* **Concept:** Propagate trace IDs, visualize request flows, bottlenecks.

## 4.5 Putting It All Together

* Integrating logs, metrics, and traces for a comprehensive observability stack.
* Demo: Visualizing a trace of a request across three microservices using Zipkin.