Microservice Concepts –

Service to Service Communication.

Security in Microservices.

Revisit

1. Created Eureka Server – It’s a Separate Spring Boot Project with Eureka Discovery Server dependency (Spring Cloud Discovery) – [Service Status Check, it helps to communicate with all the registered service by using their name.

<http://localhost:8081/api/v1/customers> -- Testing the service individually

<http://customer-service/api/v1/customers> -- accessing the service using it’s name which is registered with eureka [server.port=8761, register-with-eureka, fetch-registry =false]

Different Service Discovery options – Eureka, ZooKeeper, Consul.

1. We Created micro-service each is a separate Spring boot project. [customer-service, accout-service, transaction-service]
2. Eureka Discovery Client. – web, devtools, data-jpa, h2,mysql, Lombok, validation
3. Pom.xml (Project Object Model) – This is the back bone of any maven project. [groupId, artifactId]

Security – Spring Security [It’s a separate module in Spring framework – It simplifies the process of securing the application]

1. It creates a sample user with default user name as “user” and generate a dynamic password which will be displayed in the console at the time of starting the application.
2. It creates 2 end points /login, /logout
3. Login screen, and Logout confirmation screen also created automatically.
4. It also creates & manages session
5. Default authentication mechanism of spring security is – usercredentials authentication

Authentication & Authorization

* Allowing the user based on some details/credential (checking the details provided by the user is valid/invalid) – Credentials based authentication
* Role based authorization

Popular security methods

1. JWT – Token based authentication & authorization
2. OAuth
3. Credentials based authentication

Adding sample data to your micro-service

1. Use schema.sql [create queries] and data.sql [insert queries] file - ddl-auto = create-drop or none [ resources folder]
2. Sites/AI – mockaroo <https://www.mockaroo.com/>
3. Generate using Code

Service to Service Communication

1. Synchronous Communication (using Request/Response) – Blocking operations
2. Asynchronous Communication (using Event Driven Arch) – Non-Blocking operation

1. [Synchronous Communication](https://www.google.com/search?sca_esv=38b8b35f2fe0540c&sxsrf=AE3TifN8-X-u551t9l5SmiBvEzffgrYGtw%3A1754453029073&q=Synchronous+Communication&sa=X&ved=2ahUKEwiztMrspvWOAxVlh1YBHY3RHzwQxccNegQIOBAC&mstk=AUtExfA7mIoMXl72wgu2YK0qD-S6bSL3DAtFtAP6PVusi0fslfzs4pcbPGBGbMRcxlrCWFUb0312yVCqiDE-TKlM2z7-8eNMdlhrxargaHrCSps9smPZbocP1ZWOT8ZwSCT2PNrdFqEvOf2wsbhh06qnGdZ4bEFxoudqs8AdOCOxajKJ1pc&csui=3):

* **Request-Response:**

One service sends a request to another and waits for a response before continuing. This is often implemented using REST APIs over HTTP, or gRPC.

* **Advantages:**

Simple to implement, easy to understand, suitable for scenarios where immediate responses are needed.

* **Disadvantages:**

Can lead to tight coupling, single points of failure, and potential performance bottlenecks if not managed carefully.

2. [Asynchronous Communication](https://www.google.com/search?sca_esv=38b8b35f2fe0540c&sxsrf=AE3TifN8-X-u551t9l5SmiBvEzffgrYGtw%3A1754453029073&q=Asynchronous+Communication&sa=X&ved=2ahUKEwiztMrspvWOAxVlh1YBHY3RHzwQxccNegUIgwEQAg&mstk=AUtExfA7mIoMXl72wgu2YK0qD-S6bSL3DAtFtAP6PVusi0fslfzs4pcbPGBGbMRcxlrCWFUb0312yVCqiDE-TKlM2z7-8eNMdlhrxargaHrCSps9smPZbocP1ZWOT8ZwSCT2PNrdFqEvOf2wsbhh06qnGdZ4bEFxoudqs8AdOCOxajKJ1pc&csui=3):

* **Event-Driven:**

Services communicate by publishing and subscribing to events, often using a message broker (like Kafka, RabbitMQ). One service publishes an event, and multiple services can react to it independently.

* **Advantages:**

Loose coupling, increased resilience, improved scalability, better suited for complex, event-driven systems.

* **Disadvantages:**

Increased complexity, requires managing message brokers, potential for eventual consistency issues.

3. Common Protocols/Technologies:

* [**REST**](https://www.google.com/search?sca_esv=38b8b35f2fe0540c&sxsrf=AE3TifN8-X-u551t9l5SmiBvEzffgrYGtw%3A1754453029073&q=REST&sa=X&ved=2ahUKEwiztMrspvWOAxVlh1YBHY3RHzwQxccNegUIuQEQAQ&mstk=AUtExfA7mIoMXl72wgu2YK0qD-S6bSL3DAtFtAP6PVusi0fslfzs4pcbPGBGbMRcxlrCWFUb0312yVCqiDE-TKlM2z7-8eNMdlhrxargaHrCSps9smPZbocP1ZWOT8ZwSCT2PNrdFqEvOf2wsbhh06qnGdZ4bEFxoudqs8AdOCOxajKJ1pc&csui=3)**:**

A widely used architectural style for building web services, often implemented using HTTP for request-response interactions.

* [**gRPC**](https://www.google.com/search?sca_esv=38b8b35f2fe0540c&sxsrf=AE3TifN8-X-u551t9l5SmiBvEzffgrYGtw%3A1754453029073&q=gRPC&sa=X&ved=2ahUKEwiztMrspvWOAxVlh1YBHY3RHzwQxccNegUIwwEQAQ&mstk=AUtExfA7mIoMXl72wgu2YK0qD-S6bSL3DAtFtAP6PVusi0fslfzs4pcbPGBGbMRcxlrCWFUb0312yVCqiDE-TKlM2z7-8eNMdlhrxargaHrCSps9smPZbocP1ZWOT8ZwSCT2PNrdFqEvOf2wsbhh06qnGdZ4bEFxoudqs8AdOCOxajKJ1pc&csui=3)**:**

A high-performance RPC framework developed by Google, efficient for internal service-to-service communication.

* **Message Brokers:**

Platforms like Kafka, RabbitMQ, and others facilitate asynchronous communication by decoupling services through message queues.

* [**GraphQL**](https://www.google.com/search?sca_esv=38b8b35f2fe0540c&sxsrf=AE3TifN8-X-u551t9l5SmiBvEzffgrYGtw%3A1754453029073&q=GraphQL&sa=X&ved=2ahUKEwiztMrspvWOAxVlh1YBHY3RHzwQxccNegUIuAEQAQ&mstk=AUtExfA7mIoMXl72wgu2YK0qD-S6bSL3DAtFtAP6PVusi0fslfzs4pcbPGBGbMRcxlrCWFUb0312yVCqiDE-TKlM2z7-8eNMdlhrxargaHrCSps9smPZbocP1ZWOT8ZwSCT2PNrdFqEvOf2wsbhh06qnGdZ4bEFxoudqs8AdOCOxajKJ1pc&csui=3)**:**

A query language for APIs and a runtime for fulfilling those queries with existing data. Can support both synchronous and asynchronous communication styles.

4. Other Important Patterns:

* [**API Gateway**](https://www.google.com/search?sca_esv=38b8b35f2fe0540c&sxsrf=AE3TifN8-X-u551t9l5SmiBvEzffgrYGtw%3A1754453029073&q=API+Gateway&sa=X&ved=2ahUKEwiztMrspvWOAxVlh1YBHY3RHzwQxccNegUItwEQAQ&mstk=AUtExfA7mIoMXl72wgu2YK0qD-S6bSL3DAtFtAP6PVusi0fslfzs4pcbPGBGbMRcxlrCWFUb0312yVCqiDE-TKlM2z7-8eNMdlhrxargaHrCSps9smPZbocP1ZWOT8ZwSCT2PNrdFqEvOf2wsbhh06qnGdZ4bEFxoudqs8AdOCOxajKJ1pc&csui=3)**:** Acts as a single entry point for clients, routing requests to the appropriate microservices.
* [**Circuit Breaker**](https://www.google.com/search?sca_esv=38b8b35f2fe0540c&sxsrf=AE3TifN8-X-u551t9l5SmiBvEzffgrYGtw%3A1754453029073&q=Circuit+Breaker&sa=X&ved=2ahUKEwiztMrspvWOAxVlh1YBHY3RHzwQxccNegUItgEQAQ&mstk=AUtExfA7mIoMXl72wgu2YK0qD-S6bSL3DAtFtAP6PVusi0fslfzs4pcbPGBGbMRcxlrCWFUb0312yVCqiDE-TKlM2z7-8eNMdlhrxargaHrCSps9smPZbocP1ZWOT8ZwSCT2PNrdFqEvOf2wsbhh06qnGdZ4bEFxoudqs8AdOCOxajKJ1pc&csui=3)**:** A pattern that prevents cascading failures by temporarily disabling communication with a failing service.
* [**Saga**](https://www.google.com/search?sca_esv=38b8b35f2fe0540c&sxsrf=AE3TifN8-X-u551t9l5SmiBvEzffgrYGtw%3A1754453029073&q=Saga&sa=X&ved=2ahUKEwiztMrspvWOAxVlh1YBHY3RHzwQxccNegUIswEQAQ&mstk=AUtExfA7mIoMXl72wgu2YK0qD-S6bSL3DAtFtAP6PVusi0fslfzs4pcbPGBGbMRcxlrCWFUb0312yVCqiDE-TKlM2z7-8eNMdlhrxargaHrCSps9smPZbocP1ZWOT8ZwSCT2PNrdFqEvOf2wsbhh06qnGdZ4bEFxoudqs8AdOCOxajKJ1pc&csui=3)**:** A pattern for managing distributed transactions across multiple microservices.
* [**CQRS**](https://www.google.com/search?sca_esv=38b8b35f2fe0540c&sxsrf=AE3TifN8-X-u551t9l5SmiBvEzffgrYGtw%3A1754453029073&q=CQRS&sa=X&ved=2ahUKEwiztMrspvWOAxVlh1YBHY3RHzwQxccNegUIowEQAQ&mstk=AUtExfA7mIoMXl72wgu2YK0qD-S6bSL3DAtFtAP6PVusi0fslfzs4pcbPGBGbMRcxlrCWFUb0312yVCqiDE-TKlM2z7-8eNMdlhrxargaHrCSps9smPZbocP1ZWOT8ZwSCT2PNrdFqEvOf2wsbhh06qnGdZ4bEFxoudqs8AdOCOxajKJ1pc&csui=3)**(Command Query Responsibility Segregation):** Separates read and write operations for data management.
* [**Event Sourcing**](https://www.google.com/search?sca_esv=38b8b35f2fe0540c&sxsrf=AE3TifN8-X-u551t9l5SmiBvEzffgrYGtw%3A1754453029073&q=Event+Sourcing&sa=X&ved=2ahUKEwiztMrspvWOAxVlh1YBHY3RHzwQxccNegUIqAEQAQ&mstk=AUtExfA7mIoMXl72wgu2YK0qD-S6bSL3DAtFtAP6PVusi0fslfzs4pcbPGBGbMRcxlrCWFUb0312yVCqiDE-TKlM2z7-8eNMdlhrxargaHrCSps9smPZbocP1ZWOT8ZwSCT2PNrdFqEvOf2wsbhh06qnGdZ4bEFxoudqs8AdOCOxajKJ1pc&csui=3)**:** Stores data as a sequence of events, rather than the current state.

5. Choosing the Right Pattern:

The best communication pattern depends on the specific requirements of your microservices architecture. Consider factors like:

* **Coupling:** How tightly coupled do you need the services to be?
* **Scalability:** How much scalability do you need?
* **Resilience:** How resilient should the system be to failures?
* **Performance:** What are the performance requirements of the application?
* **Complexity:** How complex is the overall system?

RestTemplate – Synchronous Communication mechanism used in micro-service

Database per Service.

Centralised Logging –

Logging – ELK

Monitoring – Observability – Prometheus

# 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.