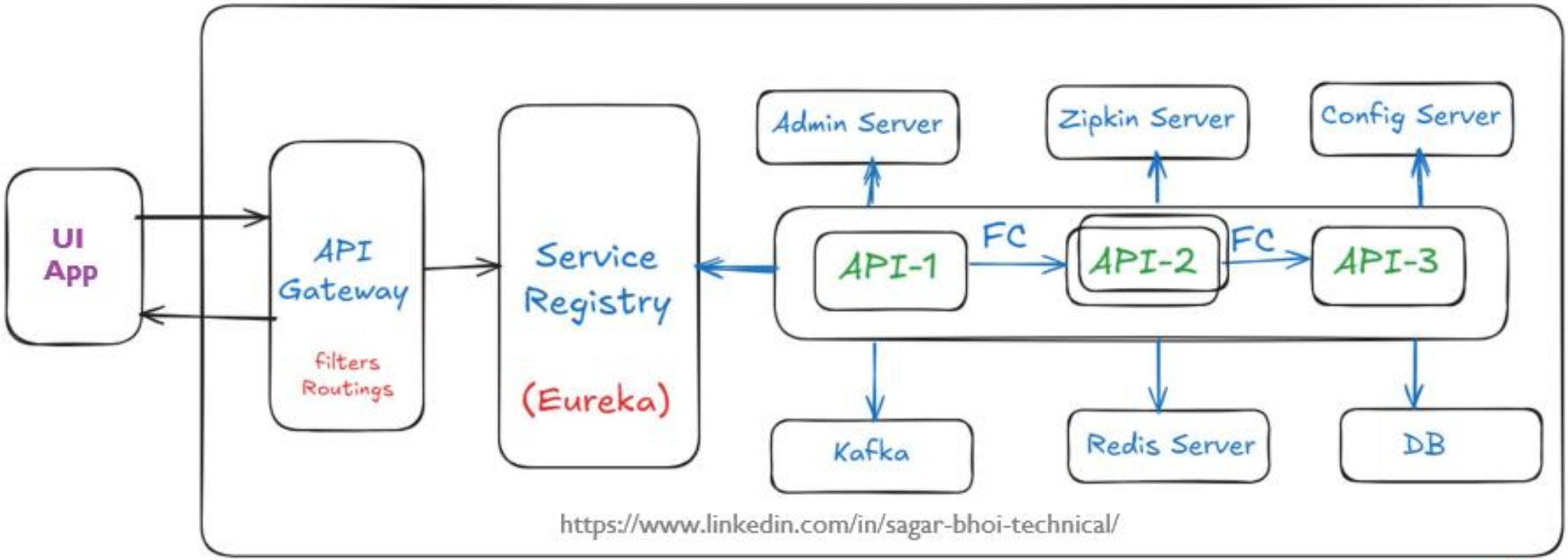
Microservices Architecture



Next ➡ Dive deeper into microservices components.

Swipe Right

API Gateway

1) Serves as a single entry point for all backend services.

2) Acts as a mediator between the frontend and microservices.

Supports:

Filters: For preprocessing and postprocessing requests (e.g., authentication, logging, header manipulation).

Routing: For directing requests to specific backend services based on URL paths or headers.

3) Helps enforce cross-cutting concerns like security, rate limiting, and logging in a centralized way.

4) Improves scalability and security by isolating backend services from direct external access.

5) Can transform requests and responses (e.g., modify payloads or add metadata) before they reach downstream services.

Service Registry (Eureka Server)

1) Acts as a centralized registry to maintain all service metadata such as name, URL, health status, etc.

2) Also known as Service Discovery.

3) Provides a user interface to view registered APIs.

4) Enables client-side load balancing when integrated with Ribbon or Spring Cloud LoadBalancer.

5) Helps services discover each other without hardcoding URLs,making the system more dynamic and scalable.

6) Automatically registers and de-registers services as they come up or go down, supporting resilience and high availability.

Admin Server (Spring Boot Admin)

1) Used to monitor and manage all microservices in one place.

2) Offers an intuitive UI to access Actuator endpoints of all services.

3) Displays detailed metrics like memory usage, thread count, and health status for each registered service.

4) Supports real-time updates via server-sent events (SSE) to track service status live.

5) Allows remote shutdown, logging level changes, and other management features for registered services.

6) Integrates easily with Spring Boot applications using minimal configuration.

Zipkin Server

1) Enables distributed tracing of requests across microservices.

2) Helps in tracking API execution paths and diagnosing performance issues.

3) Comes with a user-friendly dashboard for trace visualization.

4) Captures key data like service name, timestamp, duration, and span ID for each request.

5) Supports integration with Sleuth to automatically generate trace and span IDs in Spring Boot apps.

6) Useful for identifying bottlenecks, latency issues, and failures in service communication.

Config Server

1) Externalizes configuration properties from the application code.

2) Supports centralized management of configuration across all environments.

3) Helps in achieving loose coupling between code and configuration.

4) Enables dynamic refresh of configuration without restarting services (when used with Spring Cloud Bus).

5) Supports multiple backend sources like Git, filesystem, or even Vault for secure property storage.

6) Promotes consistency and ease of configuration management across microservices in large-scale systems.

Feign Client

1) A declarative REST client used for inter-service communication.

2) Simplifies calling other microservices within the application using annotations.

3) Automatically integrates with Eureka for service name-based resolution (load-balanced calls).

4) Reduces boilerplate code by eliminating the need to manually use RestTemplate or WebClient.

5) Supports request interceptors, retries, fallbacks (with Resilience4j/Hystrix), and custom configurations.

6) Enhances readability and maintainability by abstracting HTTP communication into simple interfaces.

Ex:

```
@FeignClient(name = "product-service") // "product-service" is the service name
public interface ProductClient {                                    registered in Eureka

    @GetMapping("/api/products/{id}")
    Product getProductById(@PathVariable("id") Long id);
}
```

Apache Kafka

1) Serves as a message broker and distributed streaming platform.

2) Follows the publish-subscribe (pub-sub) model to handle asynchronous communication.

3) Enables decoupling of services by allowing producers and consumers to operate independently.

4) Provides high throughput, fault-tolerant, and scalable message processing.

5) Stores streams of records in topics, allowing real-time and replayable event processing.

6) Commonly used for event-driven architecture, logging, analytics, and data pipelines.

Redis Server

1) Acts as an in-memory data store and caching layer.

2) Stores data in key-value pairs to reduce frequent database calls and enhance performance.

3) Supports various data structures like strings, hashes, lists, sets, and sorted sets.

4) Offers built-in support for data expiration (TTL), making it ideal for session storage and temporary data.

5) Helps in reducing latency and improving scalability in high-traffic applications.

6) Can be used for caching, pub-sub messaging, distributed locks, and rate limiting.