Microservices

Introduction

Microservices is an architectural approach in software development where complex applications are broken down into smaller, independently deployable services. Each microservice focuses on a specific business capability, communicating with other microservices through well-defined APIs or messaging systems.

This modular structure allows development teams to work on individual services concurrently, enabling greater flexibility, scalability, and technology diversity. Overall, microservices offer a way to build agile, adaptable, and scalable applications that can meet the demands of modern, complex software ecosystems.

Monolithic vs Microservice Architecture

Monolithic and microservices architectures are different approaches to designing and structuring software applications. Each has advantages and disadvantages, and the choice between them depends on various factors such as the project's complexity, scalability requirements, development team size, and more. Let's delve into the details of both architectures:

Monolithic Architecture:

An entire application is built as a single, unified codebase in a monolithic architecture. All components, features, and functionalities are tightly integrated into a single executable or deployable unit. Here's a breakdown of its characteristics:

1. **Simplicity**: Monolithic architectures are relatively simpler to develop, test, and deploy since everything is within the same codebase.

2. **Single Deployment**: The entire application is deployed as a single unit, making deployment and version control straightforward.

3. **Communication Overhead**: Communication between different application parts happens within the same process, reducing overhead.

4. **Ease of Debugging**: Debugging and troubleshooting can be easier since all components are closely connected.

5. **Development Speed**: Initial development can be faster as there's no need to manage communication between different services.

6. **Scaling Challenges**: Scaling specific components can be challenging as the entire application needs to be scaled together.

7. **Technology Stack**: All application parts must use the same technology stack, limiting flexibility in technology choices.

Microservices Architecture:

Microservices architecture breaks down an application into a collection of loosely coupled, independently deployable services, each focusing on a specific business capability. Here are its key features:

1. **Modularity**: Microservices are modular, allowing different services to be developed, deployed, and maintained independently.

2. **Scalability**: Individual services can be scaled independently based on their specific resource demands, enabling better resource utilisation.

3. **Technology Diversity**: Different services can use different technologies and programming languages, making it possible to select the best tool for each task.

4. **Isolation and Fault Tolerance**: If one service fails, it doesn't necessarily bring down the entire application, enhancing fault tolerance and isolation.

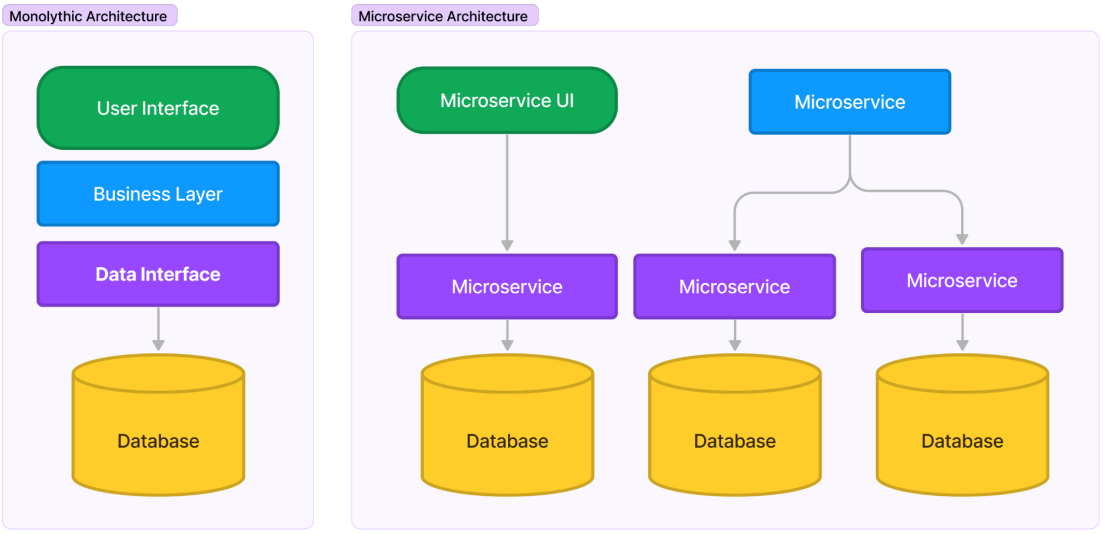
5. **Development Flexibility**: Development teams can work on different services concurrently, enabling faster feature development and innovation.

6. **Complexity**: Microservices introduce complexity, as inter-service communication must be managed effectively.

7. **Deployment Complexity**: Deploying and managing multiple services requires more sophisticated orchestration tools and practices.

8. **Operational Overhead**: Maintaining and monitoring multiple services might increase operational complexity.

9. **Communication Between Services**: Effective communication methods (e.g., APIs, messaging systems) must be implemented between services.



Choosing Between Monolithic and Microservices

The decision to use monolithic or microservices architecture depends on the project's scope, expected growth, team expertise, development speed, and operational complexity. Smaller projects with limited scalability needs might find a monolithic architecture suitable due to its simplicity. On the other hand, larger and more complex projects that demand scalability, technology diversity, and agility might benefit from a microservices architecture.

In practice, some projects also opt for hybrid architectures, starting with a monolithic approach and then gradually refactoring parts of the application into microservices as needed. It's important to note that both approaches have their trade-offs, and there's no one-size-fits-all solution. The choice should be made considering the specific requirements and goals of the project.

Principles of Microservices Design

Designing microservices involves adhering to several fundamental principles to ensure the success and effectiveness of the architecture. These principles contribute to creating a scalable, maintainable, and resilient system. Here are the fundamental principles of microservice design in detail:

1. **Single Responsibility Principle (SRP)**: Each microservice should have a single, well-defined responsibility or business capability. This ensures that each service is focused on a specific task and remains cohesive. Separating concerns makes developing, testing, and maintaining individual services easier.

2. **Decentralised Data Management**: Each microservice should have its own database or data store specific to its needs. This avoids tightly coupling services through a shared database, allowing services to evolve independently without impacting others.

3. **API-First Design:** Define clear and consistent APIs for each microservice. APIs serve as contracts for communication between services, making it easier for different teams to collaborate and ensuring service compatibility.

4. **Loose Coupling**: Services should interact with each other in a loosely coupled manner, meaning they should be able to evolve independently without causing widespread changes. This is achieved through well-defined APIs and messaging systems.

5. **Autonomous Development and Deployment**: Microservices should be independently deployable and upgradeable. Teams can develop, test, and deploy services without coordinating with other teams, enabling faster development cycles and reducing bottlenecks.

6. **Technology Diversity**: Each microservice can use the best-suited technology stack for its specific task. This allows teams to leverage the strengths of different technologies and programming languages, promoting innovation and efficiency.

7. **Resilience and Fault Isolation**: Design services with resilience in mind. A failure in one microservice should not cascade into a system-wide failure. Implement mechanisms like circuit breakers and retries to handle and isolate failures.

8. **Event-Driven Architecture**: Utilises events and asynchronous communication patterns to enable services to react to changes and events in the system. This supports real-time updates, scalability, and responsiveness.

9. **Scalability and Load Distribution**: Microservices can be scaled independently based on their workload. This allows efficient resource utilisation and ensures that only the necessary components are scaled, avoiding over-provisioning.

10. **Monitoring and Observability**: Implement robust tracking, logging, and tracing across services to gain insights into system health, performance, and potential issues. This helps in quickly identifying and resolving problems.

11. **Continuous Integration and Continuous Deployment (CI/CD)**: Automation is crucial. Implement CI/CD pipelines to automate testing, deployment, and updates, ensuring a consistent and reliable release process for each microservice.

12. **Autonomy and Decentralization**: Microservices should operate with autonomy. Each service team should have control over its service's development, deployment, and maintenance. This empowers teams to make decisions that align with their service's requirements and business goals, fostering innovation and ownership.

13. **Polyglot Persistence**: Choose the most suitable data storage technology for each microservice's needs. This principle acknowledges that different services may have different data storage requirements, and using the right tool for the job can optimise performance and data management.

Including these principles further enriches the comprehensive set of guidelines for designing microservices, helping to address various aspects of development, operations, and architecture.

InterService Communication

RestTemplate is a class in the Spring Framework that provides a convenient way to interact with RESTful web services by making HTTP requests. It's commonly used for inter-service communication in microservices architectures, where one microservice communicates with another over HTTP using RESTful principles. Here's how RestTemplate works and is used for interservice communication:

1. **Dependency and Configuration**: To use RestTemplate, you must include the necessary Spring dependencies in your project, usually via Maven or Gradle. Once the dependencies are set up, you typically create a bean of type RestTemplate in your Spring configuration. This bean can be customised with options like timeouts, message converters, and authentication settings.

2. **Making HTTP Requests**: RestTemplate provides methods for making various HTTP requests, such as GET, POST, PUT, and DELETE. These methods correspond to the HTTP methods used in RESTful APIs. If applicable, you pass in the target URL, request parameters, headers, and, optionally, the request body.

3. **Handling Responses**: RestTemplate returns the response from the HTTP request as a Java object, typically deserialised from the response body. The deserialisation is based on the configured message converters. You can handle different responses by specifying the expected response type when requesting.

4. **Error Handling**: RestTemplate also handles HTTP errors (e.g., 4xx and 5xx responses) by default. You can customise error handling behaviour, such as manually throwing exceptions for specific status codes or handling errors.

For example, in the **CN Kart** application, we make an API call to Inventory Service to check if a particular item is in inventory.

```
public String placeOrder(OrderRequest orderRequest) {
        Order order = new Order();
        order.setOrderNumber(UUID.randomUUID().toString());
        order.setPrice(orderRequest.getPrice());
        order.setQuantity(orderRequest.getQuantity());
        order.setSkuCode(orderRequest.getSkuCode());

        // Call Inventory Service, and place order if product is in stock
        String resourceUrl =
"http://localhost:8083/api/inventory?skuCode={skuCode}&qty={qty}";
        boolean isInStock = Boolean.TRUE.equals(restTemplate.
                getForEntity(resourceUrl, Boolean.class,
                        orderRequest.getSkuCode(),
                        orderRequest.getQuantity()).getBody());
        if(isInStock) {
            orderRepository.save(order);
            return "Order Placed";
        }
        else {
            return "Item is not in stock, please try again later";
            }
    }
```

Discovery and Registration

Eureka is a service discovery and registration tool provided by Netflix as part of their open-source microservices infrastructure components. It enables dynamic service registration and discovery within a microservices architecture. In this setup, services register themselves with the Eureka server, and other services can discover and communicate with registered services without hard-coding their network locations. Here's how the discovery and registration process works using Eureka:

1. **Eureka Server Setup**: Set up an Eureka server, the central registry for all services. It maintains a list of registered services, metadata, and health status. The Eureka server can be configured using Spring Cloud or directly with the Eureka libraries.

2. **Service Registration**: When a microservice starts, it registers itself with the Eureka server. It provides metadata such as its service name, instance ID, host, port, and health status. This registration process usually includes the appropriate Spring Cloud Eureka annotations and configuration in the microservice's codebase.

3. **Heartbeats and Health Checks**: After registering, a microservice sends periodic heartbeats to the Eureka server to indicate that it's still alive. The Eureka server monitors these heartbeats and maintains an up-to-date list of available services. Health checks can be customised, allowing the Eureka server to determine whether a service instance is healthy.

4. **Service Discovery**: When a microservice wants to communicate with another service, it queries the Eureka server to discover the target service's network location (host and port). This dynamic discovery eliminates the need for hardcoded URLs, making the architecture more resilient to changes in service locations.

5. **Load Balancing**: Eureka can also be integrated with load balancers, such as Netflix Ribbon, to distribute requests across multiple instances of a service. Load balancers use the registry information from the Eureka server to route requests effectively.

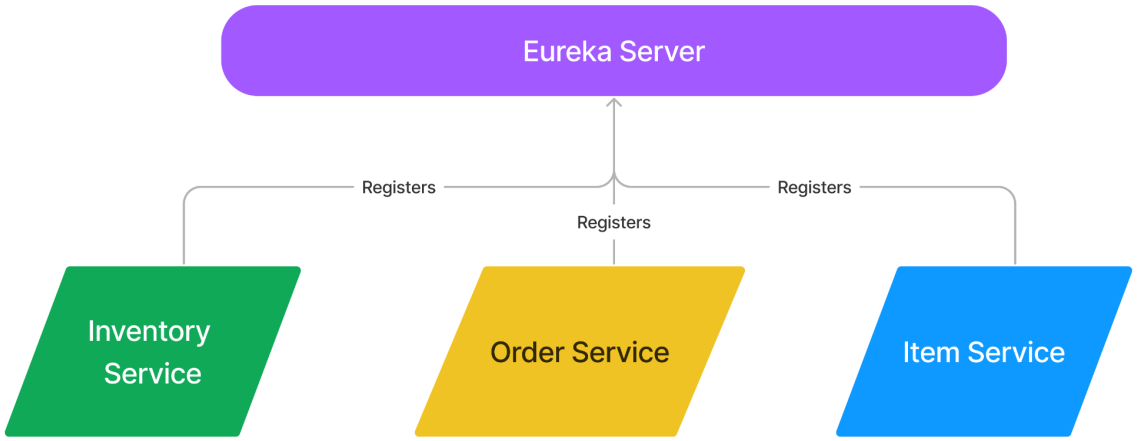
6. **Configuration**: To set up Eureka, you must configure the Eureka server and client library in each microservice. This configuration involves service names, registration details, and Eureka server URLs.

7. **High Availability**: In a production environment, you might deploy multiple instances of the Eureka server for high availability and fault tolerance. This way, if one Eureka server goes down, the remaining ones can still handle service registration and discovery.

8. **Example Usage**: CN Kart Application

```
// Discovery Server-
@SpringBootApplication
@EnableEurekaServer
public class DiscoveryServerApplication {
	public static void main(String[] args) {
		SpringApplication.run(DiscoveryServerApplication.class,args);
	}
}
// Application.yml
eureka:
  instance:
    hostname: localhost
  client:
    register-with-eureka: false
    fetch-registry: false
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
server:
  port: 8761

// Discovery Client
@SpringBootApplication
@EnableEurekaClient
public class InventoryApplication {
	public static void main(String[] args) {
		SpringApplication.run(InventoryApplication.class, args);
	}
}
```



Using Eureka simplifies the management of service discovery and registration in a dynamic microservices environment. It promotes flexibility and scalability by allowing services to be added, removed, and scaled with ease. Keep in mind that while Eureka is a widely used solution, other service discovery tools are available in the microservices ecosystem, such as Consul and ZooKeeper, which offer similar functionalities.

Circuit Breaker - Hystrix

Hystrix is a widely used open-source library developed by Netflix that provides fault tolerance and latency tolerance for distributed systems. It's beneficial in microservices architectures where services can fail or become slow for various reasons. Hystrix helps prevent cascading failures and provides mechanisms to handle and recover from failures gracefully. It's often called a "***Circuit Breaker***" due to its core functionality of preventing further requests to a failing service, similar to an electrical circuit breaker.

Here's how Hystrix works and its key features in detail:

1. **Circuit Breaker Pattern**: Hystrix implements the Circuit Breaker pattern. When a microservice repeatedly fails or experiences high latency, Hystrix trips the circuit breaker, stopping any requests from being sent to that service. This prevents overloading the failing service and allows it time to recover.

2. **Core Features**

    a. **Fallback Mechanism**: If a service call fails or exceeds defined latency thresholds, Hystrix can provide a fallback response or execute a fallback method, ensuring the user experience is not severely impacted.

    b. **Metrics Collection**: Hystrix collects various metrics related to service calls, including success rate, response times, and error rates. These metrics help in monitoring and making informed decisions about service health.

c. **Dynamic Thresholds**: Hystrix adjusts its thresholds dynamically based on the real-time metrics it collects. This ensures the circuit breaker adapts to changing conditions and avoids false positives or negatives.

d. **Request Caching**: Hystrix can cache responses from successful requests, allowing subsequent identical requests to be served from the cache, thus reducing the load on the service.

e. **Request Collapsing**: Hystrix can combine multiple requests into a single batch request, reducing the number of requests sent to a service and improving efficiency.

f. **Concurrency Control**: Hystrix can limit the number of concurrent requests to a specific service, preventing overload situations.

3. **Integration with Microservices**: Hystrix is often integrated into microservices architectures to provide resilience at the service level. Each microservice that makes external calls can wrap those calls in Hystrix commands, benefiting from circuit breaking and fallback mechanisms.

4. **Configuration**: Hystrix provides configuration options to set circuit breaker properties, such as error threshold percentage, request volume threshold, and sleep window. These properties define when the circuit breaker should open, close, or half-open (allowing a few requests to test service health).

5. **Example Usage**: CN Kart Application

```
public class OrderController {

    private final OrderService orderService;

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    @HystrixCommand(fallbackMethod = "fallbackPlaceOrder")
    public String placeOrder(@RequestBody OrderRequest orderRequest) {
        log.info("Placing Order");
        return orderService.placeOrder(orderRequest);
    }

    public String fallbackPlaceOrder(@RequestBody OrderRequest orderRequest) {
       return "Service is Not available";
    }
}
```

6. **Benefits**:

a. **Resilience**: Hystrix helps prevent cascading failures by isolating failures to individual services.

b. **Improved User Experience**: Fallback mechanisms ensure that even if a service is unavailable, users receive some response instead of errors.

c. **Operational Insights**: Hystrix metrics provide valuable insights into service health and performance, aiding in monitoring and troubleshooting.

Hystrix has been widely adopted in the microservices community for building more reliable and resilient systems. However, it's important to note that as of my last update in September 2021, Netflix has officially deprecated Hystrix in favour of other solutions like resilience4j or adopting circuit-breaking capabilities from frameworks like Spring Cloud. Always refer to the latest documentation and best practices for the most up-to-date information on resilience strategies and tools.

Feign Client

A Feign client is a declarative HTTP client library that simplifies making HTTP requests to other services or APIs in a microservices architecture. Developed by Netflix and now a part of the Spring Cloud ecosystem, Feign allows developers to define API communication using Java interfaces and annotations. This abstraction streamlines interaction with remote services, making microservice communication more efficient and manageable. Here's how the concept of a Feign client works and its significance in microservice communication:

Significance of Feign Client in Microservice Communication:

1. **Simplified Code**: Feign eliminates the need for writing boilerplate HTTP code, reducing code complexity and making the client-side codebase more concise and readable.

2. **Abstraction**: Feign abstracts away the details of HTTP communication and service discovery, allowing developers to focus on defining the API interactions rather than managing the low-level networking aspects.

3. **Standardisation**: Feign clients promote consistency in API consumption across different services, making adhering to established conventions and contracts easier.

4. **Integration with Microservices Ecosystem**: Feign seamlessly integrates with other Spring Cloud components like Eureka for service discovery and Ribbon for load balancing, enhancing the overall microservices ecosystem.

5. **Rapid Development**: Using Feign, developers can quickly create and maintain Feign client interfaces, accelerating development.

7. **Example Usage**: CN Kart Application

```
// Inventory Controller
@RestController
@RequestMapping("/api/inventory")
@RequiredArgsConstructor
@Slf4j
public class InventoryController {

    private final InventoryService inventoryService;

    @GetMapping
    @ResponseStatus(HttpStatus.OK)
   public boolean isInStock(@RequestParam Long skuCode,@RequestParam Integer qty){
        log.info("Received inventory check request for skuCode: {}", skuCode);
        return inventoryService.isInStock(skuCode,qty);
    }
}
```

```
// Feign implementation in Order Service
@FeignClient(name="INVENTORY-SERVICE")
public interface InventoryService {


@GetMapping("/api/inventory?skuCode={skuCode}&qty={qty}")
public boolean isInStock(@RequestParam Long skuCode,@RequestParam Integer qty);
}
```

In summary, a Feign client simplifies microservice communication by offering a declarative approach to defining and using APIs. It enhances code readability, reduces development effort, and integrates well with other microservices tools, contributing to the efficiency and effectiveness of microservice-based systems.

API Gateway

An API Gateway is a centralised server or service that serves as an entry point for clients to access various microservices within a distributed architecture. It handles communication complexities, routing, and security, acting as a traffic coordinator and simplifying client-to-service interactions. By abstracting away the intricacies of multiple services, API Gateways ensure a unified and controlled interface for clients.
They manage authentication, load balancing, response aggregation, and caching, enhancing performance, security, and scalability. In summary, API Gateways are pivotal in optimising microservices communication, enhancing user experience, and maintaining system reliability.

Here are a few additional points highlighting the significance of an API Gateway:

1. **Security Enforcement**: API Gateways centralise security measures, such as authentication, authorisation, and encryption. They provide a single point to enforce security policies and protect services from unauthorised access.

2. **Simplified Client Code**: API Gateways abstract the complexities of service interactions, resulting in cleaner and more straightforward client-side code. Clients only need to interact with the Gateway, which handles the intricacies of routing and communication with various services.

3. **Rate Limiting and Quotas**: API Gateways can enforce rate limits and quotas on incoming requests, preventing abuse and ensuring fair usage of services.

4. **Analytics and Monitoring**: API Gateways offer insights into usage patterns, performance metrics, and traffic trends. These analytics assist in making informed decisions, optimising resources, and identifying potential issues.

5. **Version Management**: API Gateways manage the versioning of APIs, enabling smooth transitions and maintaining backward compatibility for clients as services evolve.

6. **Microservices Independence**: API Gateways shield clients from the internal structure of microservices. This insulation allows services to be added, removed, or replaced without affecting clients.

7. **Response Transformation**: API Gateways can transform responses from services to a format expected by clients, ensuring consistent data presentation and reducing client-side processing.

8. **Centralised Logging and Tracing**: API Gateways can collect logs and trace information, aiding in troubleshooting and monitoring service interactions.

Incorporating an API Gateway addresses multiple challenges in microservices architectures, streamlining communication, security, monitoring, and maintenance. It is crucial for building robust, manageable, and efficient distributed systems.

Conclusion

Microservices is an architectural approach in software development where complex applications are broken down into smaller, independently deployable services. Each service focuses on a specific business capability, communicating with other services through well-defined APIs or messaging systems. This modular structure enables development teams to work concurrently on different services, promoting flexibility, scalability, and technology diversity. Microservices architecture offers several key benefits:

1. **Modular Scalability and Flexibility**: Microservices break applications into smaller, independently deployable services, allowing individual components to be scaled, developed, and updated independently. This modular approach promotes agility, efficient resource utilisation, and the ability to use diverse technology stacks.

2. **Resilient and Specialized Services**: Microservices enhance system resilience by containing failures within specific services, preventing cascading issues. Each service focuses on a distinct business capability, optimising performance, improving user experiences, and supporting specialised optimisations.

3. **Decentralised Collaboration and Autonomy**: Development teams have autonomy over their services, fostering innovation, accountability, and faster development cycles. Clear APIs facilitate service communication, while event-driven architecture enables real-time updates and responsiveness.

While offering benefits like adaptability and scalability, microservices also require managing challenges like communication complexity and operational overhead. The decision to adopt this architecture should consider project scope, team expertise, and potential trade-offs, necessitating effective practices and tools for successful implementation.

Instructor Codes

- CNKart Application

References

1. Official Documentation
2. What are Microservices?
3. Monolithic vs Microservice Architecture
4. Monolithic vs Microservice Architecture II
5. Principles of Microservice Design
6. Eureka and Spring Cloud
7. Spring Cloud
8. Hystrix
9. API Gateway
10. Feign Client
11. Advantages and Disadvantages of Microservices