What is memory management in java And how it is handled?

* Memory management in Java is primarily handled through **automatic garbage collection**, which helps in reclaiming the memory by destroying unused objects.  
  **Heap**: Stores objects and class instances. This is the main area managed by the garbage collector.
* **Stack**: Stores method call frames, local variables, and references to objects in the heap.
* **Method Area (MetaSpace in Java 8+)**: Stores class metadata, static variables, and method definitions.
* **Program Counter (PC) Register**: Keeps track of the current instruction being executed.
* **Native Method Stack**: Used for native (non-Java) method calls.  
    
    
  Q. Explain each best practice to avoid memory leak in java with example of code now.  
    
  **1. Avoid Unintentional Object References (especially in Collections)**
* **Problem**: Objects stored in collections (like List, Map) are not garbage collected if not removed.
* **Best Practice**: Remove unused objects from collections.

-------------------------------------------------------------------------

Map<string, object> cache = new HashMap<>();

public void addToCache(String key, Object value) {

    cache.put(key, value);

}

public void clearCache(String key) {

    cache.remove(key); // Prevents memory leak

}

**[2]Use Weak References for Caches**

**Problem**: Strong references in caches prevent GC.

**Best Practice**: Use WeakHashMap or WeakReference  
  
**What is a Strong Reference?**

In Java, **strong references** are the default type of reference. If an object is strongly referenced, the **Garbage Collector (GC)** will **never** remove it from memory as long as the reference exists.

**🔍 What is a Weak Reference?**

A **weak reference** allows the GC to collect the object **even if it is still referenced**, as long as it's only weakly referenced. This is useful for **caching**, where you don’t want the cache to prevent memory cleanup.

**WeakHashMap vs HashMap**

* HashMap uses **strong references** for keys and values.
* WeakHashMap uses **weak references** for **keys**. If a key is no longer used elsewhere, it can be garbage collected, and the entry is removed from the map.

import java.util.HashMap;

import java.util.WeakHashMap;

public class CacheExample {

    public static void main(String[] args) {

        // Strong reference example

        HashMap<object, string> strongMap = new HashMap<>();

        Object strongKey = new Object();

        strongMap.put(strongKey, "Strong Reference");

        // Weak reference example

        WeakHashMap<object, string> weakMap = new WeakHashMap<>();

        Object weakKey = new Object();

        weakMap.put(weakKey, "Weak Reference");

        // Remove strong references

        strongKey = null;

        weakKey = null;

        // Suggest GC

        System.gc();

        // Wait a bit for GC to run

        try { Thread.sleep(1000); } catch (InterruptedException e) {}

        System.out.println("StrongMap: " + strongMap);

        System.out.println("WeakMap: " + weakMap);

    }

}  
  
ouptput :  
StrongMap: {java.lang.Object@1b6d3586=Strong Reference}

WeakMap: {}

* The strongMap still holds the object.
* The weakMap entry is **gone** because the key was weakly referenced and GC collected it

**When to Use WeakHashMap or WeakReference?**

* **Caches**: You want to store data temporarily and allow GC to clean it up when memory is low.
* **Listeners or Callbacks**: Avoid memory leaks by not preventing GC of unused objects.

--------------------------------------------------------------------------------------------------------  
[3] **Close Resources Properly**

**Problem**: Open streams/sockets hold memory.

**Best Practice**: Use try-with-resources.

try (BufferedReader reader = new BufferedReader(new FileReader("file.txt"))) {

    String line = reader.readLine();

} // reader is auto-closed  
  
[4] **Avoid Static References to Large Objects**

**Problem**: Static fields live for the lifetime of the app.

**Best Practice**: Don’t store large objects statically.

// Bad

public static List bigList = new ArrayList<>();

// Better

public List getList() {

    return new ArrayList<>();

}  
  
[5] **Use Profiling Tools**

**Best Practice**: Use tools like **VisualVM**, **JProfiler**, or **Eclipse MAT** to detect memory leaks.

[6] **Be Careful with Inner Classes**

**Problem**: Non-static inner classes hold reference to outer class.

**Best Practice**: Use static inner classes when possible.

// Bad

class Outer {

    class Inner {

        // Holds reference to Outer

    }

}

// Good

class Outer {

    static class Inner {

        // No reference to Outer

    }

}

How do we implement security in spring boot application?

Implementing **security in a Spring Boot application** typically involves using **Spring Security**, a powerful and customizable authentication and access-control framework  
  
**Key Components of Spring Security**

1. **Authentication** – Verifying who the user is.
2. **Authorization** – Determining what the user is allowed to do.
3. **Password Encoding** – Storing passwords securely.
4. **Role-Based Access Control (RBAC)** – Restricting access based on user roles.
5. **JWT (JSON Web Token)** – For stateless authentication in REST APIs.  
     
   In Java (and specifically in Spring Boot), a **stateless REST API** means that the server does **not store any client session information** between requests. Each request from the client must contain **all the information** needed for the server to process it.

**Basic Steps to Implement Security**

**1. Add Spring Security Dependency**

In pom.xml:  
  
<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>  
---------------------------------------------------------------------------  
2.**Create a Security Configuration Class**

@EnableWebSecurity

public class SecurityConfig extends WebSecurityConfigurerAdapter {

@Override

    protected void configure(HttpSecurity http) throws Exception {

        http

            .authorizeRequests()

            .antMatchers("/public/\*\*").permitAll() // No auth needed

            .antMatchers("/admin/\*\*").hasRole("ADMIN") // Role-based access

            .anyRequest().authenticated()

            .and()

            .httpBasic(); // or .formLogin() for web apps

    }  
@Override

    protected void configure(AuthenticationManagerBuilder auth) throws Exception {

        auth.inMemoryAuthentication()

            .withUser("user").password(passwordEncoder().encode("password")).roles("USER")

            .and()

            .withUser("admin").password(passwordEncoder().encode("admin")).roles("ADMIN");

    }

    @Bean

    public PasswordEncoder passwordEncoder() {

        return new BCryptPasswordEncoder();

    }

* }  
    
  Explanation of Code:  
  **[1]@EnableWebSecurity:** Enables Spring Security’s web security support.
* **WebSecurityConfigurerAdapter:** A base class that allows you to customize Spring Security by overriding methods like configure(HttpSecurity) and configure(AuthenticationManagerBuilder).  
    
  In Spring Security 6+, WebSecurityConfigurerAdapter is deprecated. You should use SecurityFilterChain beans instead.
* **http.authorizeRequests():** Starts defining access rules for HTTP endpoints.
* **.antMatchers("/public/\*\*").permitAll**(): Allows unrestricted access to any endpoint under /public/.
* **.antMatchers("/admin/\*\*").hasRole("ADMIN"):** Restricts access to /admin/\*\* endpoints to users with the ADMIN role.
* .**anyRequest().authenticated():** All other endpoints require authentication.
* **.httpBasic():** Enables basic authentication (username/password via HTTP headers). You could replace this with .formLogin() for form-based login in web apps.
* **auth.inMemoryAuthentication():** Defines users in memory (not from a database).
* **.withUser("user")...roles("USER"):** Adds a user with username "user", password "password" (encoded), and role "USER".
* **.withUser("admin")...roles("ADMIN"):** Adds an admin user with role "ADMIN".
* @Bean: Registers this method as a Spring bean.
* BCryptPasswordEncoder: A secure password hashing algorithm used to encode passwords.

Spring Security requires passwords to be encoded. This bean ensures that the passwords used in inMemoryAuthentication() are properly hashed.

**What Does inMemoryAuthentication() Mean?**

It tells Spring Security to **store user credentials in memory** (RAM) while the application is running. This is useful for:

* **Testing or prototyping** applications
* **Simple apps** that don’t need persistent user storage
* Avoiding the need for a database or external identity provide
* **How It Works**
* Creates a user named "user" with a password and role "USER"
* Stores this user in an internal memory structure (like a map or list)
* Uses this data to authenticate incoming requests

---------------------------------------------------------------------------------  
[3]**Use JWT for REST APIs (Optional but Recommended)**

* Generate JWT on login.
* Validate JWT on each request.
* Use filters to intercept and validate tokens.

**🛡️ Additional Security Features**

* **CSRF Protection** (enabled by default for web apps).
* **CORS Configuration** for cross-origin requests.
* **OAuth2/OpenID Connect** for third-party login (Google, GitHub, etc.).
* **Method-Level Security** using @PreAuthorize, @Secured.

Here’s a simple example of **method-level security** in a Spring Boot application using @PreAuthorize and @Secured.  
[1]Adding dependency in pom.xml

<dependency>

    <groupId>org.springframework.boot</groupId>

    <artifactId>spring-boot-starter-security</artifactId>

</dependency>

**Step 2: Enable Method Security**

In your main application class or a configuration class, add:

@EnableMethodSecurity  // For Spring Security 6+

@Configuration

public class SecurityConfig {

    // You can define custom security rules here if needed

}

[3] **step 3: Use Security Annotations on Methods**

You can now annotate service or controller methods with:

**🔹 @PreAuthorize**

Checks before method execution.

@PreAuthorize("hasRole('ADMIN')")

public void deleteUser(Long userId) {

    // Only ADMIN can delete users

}

**@PostAuthorize**

Checks after method execution.

@PostAuthorize("returnObject.owner == authentication.name")

public Document getDocument(Long id) {

    // Only the owner can access the document

}

[5]**@Secured**

Simpler role-based access.

@Secured("ROLE\_USER")

public void viewProfile() {

    // Only users with ROLE\_USER can view

}

**Step 4: Configure Authentication**

You’ll need to configure users and roles either in-memory, via a database, or using an external provider (like OAuth2, LDAP, etc.).

@Bean

public UserDetailsService userDetailsService() {

    UserDetails user = User.withDefaultPasswordEncoder()

        .username("admin")

        .password("password")

        .roles("ADMIN")

        .build();

    return new InMemoryUserDetailsManager(user);

}

**how to implement jwt token security in spring boot application?**

To implement JWT (JSON Web Token) security in a Spring Boot application, you typically follow these steps:  
  
**Step 1: Add Dependencies**

* In your pom.xml (for Maven):(these does not come under spring boot starter security)  
    
  **Purpose**: Adds the **JJWT** library, which helps in creating and parsing JWT tokens.

<dependency>

    <groupId>io.jsonwebtoken</groupId>

    <artifactId>jjwt</artifactId>

    <version>0.9.1</version>

</dependency>

<dependency>

    <groupId>org.springframework.boot</groupId>

    <artifactId>spring-boot-starter-security</artifactId>

</dependency>

<dependency>

    <groupId>org.springframework.boot</groupId>

    <artifactId>spring-boot-starter-web</artifactId>

</dependency>

**Step 2: Create JWT Utility Class**

This class handles token creation and validation.

public class JwtUtil {

    private final String SECRET\_KEY = "your\_secret\_key";

    public String generateToken(UserDetails userDetails) {

        return Jwts.builder()

            .setSubject(userDetails.getUsername())

            .setIssuedAt(new Date())

            .setExpiration(new Date(System.currentTimeMillis() + 1000 \* 60 \* 60 \* 10)) // 10 hours

            .signWith(SignatureAlgorithm.HS256, SECRET\_KEY)

            .compact();

    }

    public String extractUsername(String token) {

        return Jwts.parser()

            .setSigningKey(SECRET\_KEY)

            .parseClaimsJws(token)

            .getBody()

            .getSubject();

    }

    public boolean validateToken(String token, UserDetails userDetails) {

        final String username = extractUsername(token);

        return (username.equals(userDetails.getUsername()) && !isTokenExpired(token));

    }

    private boolean isTokenExpired(String token) {

        Date expiration = Jwts.parser()

            .setSigningKey(SECRET\_KEY)

            .parseClaimsJws(token)

            .getBody()

            .getExpiration();

        return expiration.before(new Date());

    }

* }  
    
    
  Explanation of code :  
  [1] private final String SECRET\_KEY = "your\_secret\_key"; 🡪
* This is the **secret** used to sign the JWT. Keep this secure and never expose it publicly.  
  [2] generateToken() ->
* Jwts.builder(): Starts building a JWT.
* .setSubject(username): Sets the subject (usually the username).
* .setIssuedAt(new Date()): Sets the current time as the issue time.
* .setExpiration(...): Sets the token to expire in 10 hours.
* .signWith(...): Signs the token using HMAC SHA-256 and the secret key.
* .compact(): Finalizes and returns the token as a string.  
    
  [3] extractUsername() ->
* Parses the token using the secret key.
* Extracts the **subject** (username) from the token body.  
    
  [4] validateToken()->
* Checks if the username in the token matches the expected username.
* Also checks if the token is **not expired**.

[5] isTokenExpired

* Extracts the expiration date from the token.
* Compares it with the current date to check if it's expired.

✅ **Summary**: This utility class helps you:

* Create JWT tokens with a username and expiration.
* Extract the username from a token.
* Validate tokens by checking username and expiration.

**Step 3: Create JWT Filter**

Intercept requests and validate JWT.

public class JwtRequestFilter extends OncePerRequestFilter {

    @Autowired

    private JwtUtil jwtUtil;

    @Autowired

    private UserDetailsService userDetailsService;

    @Override

    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain chain)

        throws ServletException, IOException {

        final String authHeader = request.getHeader("Authorization");

        String username = null;

        String jwt = null;

        if (authHeader != null && authHeader.startsWith("Bearer ")) {

            jwt = authHeader.substring(7);

            username = jwtUtil.extractUsername(jwt);

        }

if (username != null && SecurityContextHolder.getContext().getAuthentication() == null) {

    var userDetails = userDetailsService.loadUserByUsername(username);

    if (jwtUtil.validateToken(jwt, userDetails.getUsername())) {

        var authToken = new UsernamePasswordAuthenticationToken(

                userDetails, null, userDetails.getAuthorities());

        authToken.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));

        SecurityContextHolder.getContext().setAuthentication(authToken);

    }

}

        chain.doFilter(request, response);

    }

* }  
    
  Explanation of code :
* OncePerRequestFilter: Ensures the filter runs **once per request**.
* HttpServletRequest and HttpServletResponse: Represent the incoming request and outgoing response.
* SecurityContextHolder: Holds the security context (authentication info).
* UsernamePasswordAuthenticationToken: Represents the authenticated user.
* WebAuthenticationDetailsSource: Adds request-specific details to the authentication object.
* @Component: Makes this filter a Spring-managed bean.
* Code Explanation:  
  [1]@Autowired private JwtUtil jwtUtil; @Autowired private MyUserDetailsService:
* Injects the JwtUtil class to handle token parsing and validation.
* Injects MyUserDetailsService to load user details from the username in the token.

[2]doFilterInternal() 🡪

* This method is called for every request.
* FilterChain lets the request continue to the next filter or controller.
* [3]authHeader ->
* Gets the Authorization header.
* Checks if it starts with "Bearer " (standard JWT format).
* Extracts the token and gets the username from it.  
    
  [4] **Validate and Set Authentication**
* Checks if username is present and no authentication is already set.
* Loads user details from the database or hardcoded source.
* Validates the token.
* Creates an authentication token and sets it in the security context.

### [5] **Continue Filter Chain: chain.doFilter(request, response);**

* Passes the request to the next filter or controller.  
    
   **Summary**: This filter:
* Runs on every request.
* Extracts and validates JWT from the header.
* Sets the authenticated user in the Spring Security context.

**Step 4: Configure Security**

@Configuration

@EnableMethodSecurity

public class SecurityConfig {

    @Autowired

    private JwtRequestFilter jwtRequestFilter;

    @Bean

    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

        http.csrf().disable()

            .authorizeHttpRequests(auth -> auth

                .requestMatchers("/authenticate").permitAll()

                .anyRequest().authenticated()

            )

            .sessionManagement(sess -> sess.sessionCreationPolicy(SessionCreationPolicy.STATELESS));

        http.addFilterBefore(jwtRequestFilter, UsernamePasswordAuthenticationFilter.class);

        return http.build();

    }

}  
  
Explanation of code :  
This step sets up Spring Security to:

* Allow unauthenticated access to /authenticate
* Require JWT authentication for all other endpoints
* Register the JWT filter in the security chain

[1] filterChain(HttpSecurity http) ->

* http.csrf().disable(): Disables CSRF protection (not needed for stateless REST APIs).
* .authorizeHttpRequests(): Starts defining access rules.
* .requestMatchers("/authenticate").permitAll(): Allows unauthenticated access to /authenticate.
* .anyRequest().authenticated(): Requires authentication for all other endpoints.
* .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS): Disables session creation; JWT is stateless.
* http.addFilterBefore(...): Adds our JWT filter **before** Spring’s built-in authentication filter.
* return http.build();: Builds and returns the security filter chain.

[2] authenticationManager(AuthenticationConfiguration config)->

* Exposes the AuthenticationManager as a Spring bean.
* Required for authenticating users in the login controller.

[3] **Summary**: This configuration:

* Disables sessions and CSRF
* Allows login via /authenticate
* Secures all other endpoints

Registers the JWT filter  
  
  
**Step 5: Create Authentication Endpoint  
code :**@RestController

public class AuthController {

@Autowired

private AuthenticationManager authenticationManager;

@Autowired

private JwtUtil jwtUtil;

@PostMapping("/authenticate")

public String createToken(@RequestBody AuthRequest authRequest) throws Exception {

    authenticationManager.authenticate(

            new UsernamePasswordAuthenticationToken(authRequest.getUsername(), authRequest.getPassword())

    );

    return jwtUtil.generateToken(authRequest.getUsername());

}

**🔍 Line-by-line:**

* @PostMapping("/authenticate"): Maps this method to POST requests at /authenticate.
* @RequestBody AuthRequest authRequest: Accepts a JSON body with username and password.
* authenticationManager.authenticate(...): Authenticates the user using Spring Security.
* If successful, jwtUtil.generateToken(...) creates a JWT token for the user.

The token is returned as a plain string (you can wrap it in a JSON object if preferred).  
  
**AuthRequest.java (DTO )**public class AuthRequest {

    private String username;

    private String password;

    // Getters and setters

}

**🔍 Explanation:**

* A simple POJO (Plain Old Java Object) to hold login credentials.
* Used to map the incoming JSON request body.

✅ **Summary**: This controller:

* Accepts login credentials via /authenticate
* Authenticates the user
* Returns a JWT token if credentials are valid

* Microservices Architecture :  
  A screenshot of a computer program

  AI-generated content may be incorrect.  
    
  A close-up of a diagram

  AI-generated content may be incorrect.  
    
  DESIGN PATTEN IN MICROSERVICES:  
    
  Refer link :   
  [Microservices\_Eventdriven\_Architecture.pptx](https://capgemini-my.sharepoint.com/:p:/p/tamilselvi_j/EZxAt27RHI1LvJ4-UxMBBWwBKHbsNauFZoFL-n-AjU7eEw?wdOrigin=TEAMS-MAGLEV.undefined_ns.rwc&wdExp=TEAMS-TREATMENT&wdhostclicktime=1756972780027&web=1)  
    
  GATEWAY API DESIGN PATTERN :   
    
  A **Gateway API** (often called an API Gateway) acts as a single entry point for all client requests to a microservices-based system.  
    
  1. Core Components :  
  **API Gateway Service**: A standalone service that receives all client requests.
* **Routing Logic**: Determines which microservice(s) should handle the request.
* **Authentication & Authorization**: Validates tokens or credentials before forwarding requests.
* **Rate Limiting & Throttling**: Controls traffic to prevent overload.
* **Caching**: Stores frequent responses to reduce load.
* **Request Aggregation**: Combines responses from multiple microservices into one.
* **Protocol Translation**: Converts between protocols (e.g., HTTP to gRPC).  
    
  **Use Cases of Gateway API Pattern**
* **✅ 1. Simplified Client Communication**
* Clients interact with a single endpoint rather than multiple microservices. This reduces complexity and improves maintainability.
* **✅ 2. Security Enforcement**
* Centralized authentication and authorization logic ensures consistent security across services.
* **✅ 3. Request Aggregation**
* Useful when a client needs data from multiple microservices. The gateway can fetch and combine the data before sending it back.
* **✅ 4. Protocol Bridging**
* Allows clients to use HTTP while backend services use gRPC, WebSockets, etc.
* **✅ 5. Load Balancing & Failover**
* Distributes requests across instances and handles retries or fallbacks.
* **✅ 6. Canary Releases & A/B Testing**
* Routes a percentage of traffic to new versions of services for testing.

**Example Scenario**

Imagine an **e-commerce platform** with microservices like:

* Product Service
* User Service
* Order Service
* Review Service

Instead of the mobile app calling each service directly, it sends requests to the **API Gateway**, which:

* Authenticates the user
* Routes the request to the correct service
* Aggregates product + review data
* Returns a unified response

Here's a simple example of how to implement a **Spring Cloud Gateway** in a microservices architecture.  
  
**Project Structure**

Assume you have:

* gateway-service (Spring Cloud Gateway)
* user-service (Microservice)
* order-service (Microservice)
* **1. Add Dependencies (Maven)**
* <dependencies>
* <dependency>
* <groupId>org.springframework.cloud</groupId>
* <artifactId>spring-cloud-starter-gateway</artifactId>
* </dependency>
* <dependency>
* <groupId>org.springframework.cloud</groupId>
* <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
* </dependency>
* </dependencies>

2. Configure application.yml  
server:

  port: 8080

spring:

  application:

    name: gateway-service

  cloud:

    gateway:

      routes:

        - id: user-service

          uri: lb://user-service

          predicates:

            - Path=/users/\*\*

        - id: order-service

          uri: lb://order-service

          predicates:

* - Path=/orders/\*\*  
    
  Explanation of Code :  
  **Defines routes** for the Gateway to forward requests.
* id: A unique identifier for each route.
* uri: lb://user-service means it uses **load-balanced routing** via a service registry like **Eureka**.
* predicates: Conditions to match incoming requests. Here, it matches paths like /users/\*\* and /orders/\*\*.

3. **Enable Eureka Client**

In GatewayApplication.java:

@SpringBootApplication

@EnableEurekaClient

public class GatewayApplication {

    public static void main(String[] args) {

        SpringApplication.run(GatewayApplication.class, args);

    }

}

4. **4. Sample Routing**

* Request to http://localhost:8080/users/123 → routed to user-service
* Request to http://localhost:8080/orders/456 → routed to order-service

**What is Eureka Service Registry?**

**Eureka** is a **service discovery** tool from Netflix, commonly used in Spring Cloud microservices architecture. It helps microservices **register themselves** and **discover each other** without hardcoding URLs.

**🔄 Key Roles of Eureka:**

1. **Service Registration**: Each microservice registers itself with Eureka when it starts.
2. **Service Discovery**: Other services (or the API Gateway) can query Eureka to find the location (IP/port) of a registered service.
3. **Load Balancing**: Eureka works with Ribbon or Spring Cloud LoadBalancer to distribute requests across multiple instances.
4. **Health Checks**: Eureka periodically checks if services are alive and removes unhealthy ones.

**Why Use @EnableEurekaClient in the Gateway or Microservice?**

The annotation @EnableEurekaClient tells Spring Boot that this application should:

* **Register itself** with the Eureka server
* **Fetch service registry** to discover other services

**📌 In the Gateway:**

* It needs to discover backend services like user-service, order-service, etc.
* So it must be a Eureka client to use lb://service-name URIs in application.yml.

**📌 In Microservices:**

* They register themselves with Eureka so the gateway (or other services) can find them.

**Example Flow**

1. user-service starts → registers with Eureka
2. order-service starts → registers with Eureka
3. gateway-service starts → registers with Eureka and fetches registry
4. Client sends request to gateway-service
5. Gateway uses Eureka to route request to user-service or order-service

CQRS DESIGN PATTERN :   
The **CQRS (Command Query Responsibility Segregation)** design pattern is a powerful architectural approach used in microservices to separate **read** and **write** operations.  
  
CQRS separates the **command side** (write operations like create, update, delete) from the **query side** (read operations like fetch or search). This allows each side to be optimized independently.  
  
A screenshot of a computer

AI-generated content may be incorrect.  
  
**CQRS in Product Order Service**

**🧩 Microservices Involved**

* **Product Service**
* **Order Service**
* **Query Service** (for read operations)
* **Command Service** (for write operations)
* **Event Store / Message Broker** (e.g., Kafka, RabbitMQ)

**Implementation Flow**

**✅ 1. Command Side (Write)**

* Client sends a **CreateOrderCommand** to the **Command Service**.
* Command Service validates and processes the request.
* It emits an **OrderCreatedEvent** to a message broker.
* The **Order Service** listens to this event and updates its database.

**🔍 2. Query Side (Read)**

* Client sends a **GetOrderQuery** to the **Query Service**.
* Query Service reads from a **read-optimized database** (e.g., denormalized view).
* Returns the result quickly without affecting write performance.
* **Example: Place Order**
* Client → Command API → CreateOrderCommand → Kafka → Order Service → Write DB
* **Example: Get Order Details**
* Client → Query API → GetOrderQuery → Read DB → Response  
    
  **Benefits of CQRS**
* **Scalability**: Read and write workloads scale independently.
* **Performance**: Read models can be optimized for fast queries.
* **Flexibility**: Different data models for read/write.
* **Event Sourcing Compatibility**: Works well with event-driven systems

**What is Event Sourcing in CQRS?**

**Event Sourcing** is a pattern where **state changes are stored as a sequence of events**, rather than just storing the current state in a database.

A screenshot of a white screen

AI-generated content may be incorrect.  
  
**Example in Product Order Service**

**🧱 Events:**

* ProductCreatedEvent
* OrderPlacedEvent
* OrderCancelledEvent

**🧱 Commands:**

* CreateProductCommand
* PlaceOrderCommand
* CancelOrderCommand

**🧱 State:**

* Rebuilt by replaying events from the **event store**.

**Spring Boot CQRS Code Structure (Simplified) Integrated with Kafka :  
Architecture Overview**

**🔁** Flow:

1. Command API receives a request to place an order.
2. It creates a PlaceOrderCommand and processes it.
3. An OrderPlacedEvent is generated and:
   * Stored in an Event Store
   * Published to Kafka
4. A Query Service listens to Kafka and updates a read model (e.g., MongoDB).
5. Clients query the Query API for order details.  
     
   **Key Components**

**1. Command: PlaceOrderCommand.java**

Code :  
public class PlaceOrderCommand {

    private String orderId;

    private String productId;

    private int quantity;

    // Getters and setters

}

**2. Event: OrderPlacedEvent.java**public class OrderPlacedEvent {

    private String orderId;

    private String productId;

    private int quantity;

    // Constructor, getters, setters

}

3. **Event Store with Kafka Integration**

@Component

public class EventStore {

private final List<Object> events = new ArrayList<>();

@Autowired

private KafkaTemplate<String, Object> kafkaTemplate;

private final String topicName = "order-events";

public void save(Object event) {

events.add(event); // Store locally

kafkaTemplate.send(topicName, event); // Publish to Kafka

}

public List<Object> getEvents() {

return events;

}

}

* 1. **Command Handler**

@Service

public class OrderCommandHandler {

    @Autowired

    private EventStore eventStore;

    public void handle(PlaceOrderCommand command) {

        OrderPlacedEvent event = new OrderPlacedEvent(

            command.getOrderId(),

            command.getProductId(),

            command.getQuantity()

        );

        eventStore.save(event);

    }

}

* 1. **Kafka Consumer in Query Service**

@Service

public class OrderEventListener {

    @KafkaListener(topics = "order-events", groupId = "order-query-group")

    public void consume(OrderPlacedEvent event) {

        // Update read model (e.g., MongoDB)

        System.out.println("Received event: " + event.getOrderId());

        // Save to read DB

    }

}

* 1. **Kafka Configuration (application.yml)**

spring:

  kafka:

    bootstrap-servers: localhost:9092

    consumer:

      group-id: order-query-group

      key-deserializer: org.apache.kafka.common.serialization.StringDeserializer

      value-deserializer: org.springframework.kafka.support.serializer.JsonDeserializer

      properties:

        spring.json.trusted.packages: '\*'

    producer:

      key-serializer: org.apache.kafka.common.serialization.StringSerializer

      value-serializer: org.springframework.kafka.support.serializer.JsonSerializer

**Benefits of This Setup**

* **Scalable**: Read and write sides scale independently.
* **Auditable**: Full history of events.
* **Decoupled**: Services communicate via Kafka.
* **Flexible**: Read models can be optimized for queries.

**What is a Messaging Queue in Kafka?**

Kafka is not a traditional messaging queue like RabbitMQ or ActiveMQ. Instead, it's a **distributed event streaming platform** that behaves like a **log-based message broker**.

**🔁 Messaging Queue vs Kafka**

| **Feature** | **Traditional Queue (e.g., RabbitMQ)** | **Kafka** |
| --- | --- | --- |
| Message Model | Queue (FIFO) | Log (append-only) |
| Message Retention | Deleted after consumption | Retained for configured time |
| Consumer Model | Message removed after read | Consumers track their own offset |
| Scalability | Limited | Highly scalable and distributed |
| Replay Capability | No | Yes (replay from offset) |

**🔄 Kafka in CQRS + Event Sourcing**

In our example:

**✅ Kafka as a Message Queue**

* **Producer**: The EventStore publishes events like OrderPlacedEvent to a Kafka topic (order-events).
* **Topic**: Acts like a queue where events are stored in order.
* **Consumer**: The **Query Service** listens to the topic and updates the read model.

**✅ Kafka as an Event Log**

* Events are **not deleted** after consumption.
* Consumers can **replay events** by resetting their offset.
* This supports **event sourcing**, where the system state is rebuilt by replaying events.

**🧠 Why Kafka is Ideal for CQRS + Event Sourcing**

* **Durability**: Events are persisted.
* **Replayability**: You can rebuild read models anytime.
* **Decoupling**: Producers and consumers are independent.
* **Scalability**: Handles high throughput and distributed systems.

A diagram of a service

AI-generated content may be incorrect.

**SAGA DESIGN PATTERN :**The **Saga Design Pattern** is a crucial concept in **microservices architecture** for managing **distributed transactions**. Since traditional ACID transactions don't work well across microservices, Saga provides a way to ensure data consistency using a sequence of **local transactions** and **compensating actions**.

In the context of **microservices architecture**, **ACID transactions** refer to operations that follow the **ACID properties** — **Atomicity, Consistency, Isolation, and Durability** — to ensure reliable processing of data. However, implementing ACID transactions across microservices is **challenging** due to their **distributed nature**.

Let’s break this down:

**🔍 What Are ACID Properties?**

1. **Atomicity**: All steps in a transaction succeed or none do.
2. **Consistency**: The system moves from one valid state to another.
3. **Isolation**: Concurrent transactions do not interfere with each other.
4. **Durability**: Once a transaction is committed, it remains so even in case of failures.

**🧩 ACID in Microservices: Why It's Hard**

Microservices are typically:

* **Distributed across different databases**
* **Loosely coupled**
* **Independently deployable**

This makes **traditional ACID transactions** (like those in monolithic systems using a single relational database) difficult to implement across services.

**✅ How ACID Is Achieved in Microservices**

Instead of strict ACID, microservices often use **eventual consistency** and **patterns** like:

**1. Saga Pattern**

* Breaks a transaction into a series of **local transactions**.
* Each service performs its part and publishes an event.
* If something fails, **compensating transactions** are triggered to undo previous steps.

**Types of Saga Design Pattern**

There are two main types:

**1. Choreography-Based Saga**

* **No central coordinator**.
* Services communicate via **events**.
* Each service listens for events and performs its transaction, then publishes the next event.

**Use Case**: Lightweight, loosely coupled systems where services can react to events independently.

A close-up of a diagram

AI-generated content may be incorrect.  
  
2. **2. Orchestration-Based Saga**

* A **central orchestrator** controls the saga.
* It sends commands to services and listens for replies.
* Easier to manage and debug.

**Use Case**: Complex workflows where centralized control is preferred.  
 A diagram of a process

AI-generated content may be incorrect.

**Example: Spring Boot Implementation**

Let’s walk through both types with a simple **Order Service** scenario:

* **Order Service** creates an order.
* **Payment Service** processes payment.
* **Inventory Service** reserves items.

**Choreography-Based Saga (Spring Boot + Kafka)**

**[1]Order Service publishes OrderCreatedEvent:**public void createOrder(Order order) {

    orderRepository.save(order);

    kafkaTemplate.send("order-events", new OrderCreatedEvent(order.getId()));

}

[2] **Payment Service listens and publishes PaymentCompletedEvent**

@KafkaListener(topics = "order-events")

public void handleOrderCreated(OrderCreatedEvent event) {

    // process payment

    kafkaTemplate.send("payment-events", new PaymentCompletedEvent(event.getOrderId()));

}  
  
**Inventory Service listens and reserves items**

@KafkaListener(topics = "payment-events")

public void handlePaymentCompleted(PaymentCompletedEvent event) {

    // reserve inventory

    kafkaTemplate.send("inventory-events", new InventoryReservedEvent(event.getOrderId()));

}

**Orchestration-Based Saga (Spring Boot + Camunda or Custom Orchestrator)**

**1. Saga Orchestrator initiates the saga**

public void startSaga(Order order) {

    orderService.createOrder(order);

    paymentService.processPayment(order.getId());

}  
  
**2. Orchestrator handles responses and calls next service**public void handlePaymentResponse(PaymentResponse response) {

    if (response.isSuccess()) {

        inventoryService.reserveItems(response.getOrderId());

    } else {

        orderService.cancelOrder(response.getOrderId());

}

}

**Compensation Logic**

Each service should implement a rollback method:

**Java**

public void cancelOrder(Long orderId) {

Order order = orderRepository.findById(orderId).orElseThrow();

order.setStatus("CANCELLED");

orderRepository.save(order);

}

| **Type** | **Communication** | **Complexity** | **Use Case** |
| --- | --- | --- | --- |
| Choreography | Event-driven | Low | Simple workflows |
| Orchestration | Central control | High | Complex workflows |

----------------------------------------------------------------------------------------------  
**Service Registry Pattern :**In a microservices architecture, services often need to discover and communicate with each other dynamically. A **Service Registry** acts as a **central directory** where services register themselves and discover other services.  
  
**Benefits:**

* **Dynamic discovery**: Services can find each other without hardcoding URLs.
* **Load balancing**: Registry can integrate with load balancers.
* **Fault tolerance**: Helps reroute traffic if a service goes down.
* **Scalability**: Easily add/remove services.

**Common Tools for Service Registry**

* **Eureka** (from Netflix) – most popular with Spring Boot.
* **Consul** – by HashiCorp.
* **Zookeeper** – often used with Kafka.
* **Implementing Service Registry in Spring Boot using Eureka**
* **1. Set up Eureka Server**

Add dependencies in pom.xml :  
<dependency>

    <groupId>org.springframework.cloud</groupId>

    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>

</dependency>

2. Enable Eureka Server in your main class:  
  
@SpringBootApplication

@EnableEurekaServer

public class EurekaServerApplication {

    public static void main(String[] args) {

        SpringApplication.run(EurekaServerApplication.class, args);

    }

}

3. In application.yml or application.properties:  
server:

  port: 8761

eureka:

  client:

    register-with-eureka: false

    fetch-registry: false

**register-with-eureka: false**

This tells the Eureka Server **not to register itself** with another Eureka instance.

* ✅ **Why?** Because it's the **main registry**—it doesn't need to register with another server.
* 🔄 If you had a **cluster of Eureka servers**, you might set this to true so they can replicate data between each other.

**🔍 fetch-registry: false**

This tells the Eureka Server **not to fetch the registry** from another Eureka instance.

* ✅ **Why?** Again, it's the **source of truth** for service registrations.
* 🔄 In a **peer-aware setup** (multiple Eureka servers), you might set this to true to enable syncing between them.
* **2. Create a Microservice and Register with Eureka**
* Add Eureka client dependency:

<dependency>

    <groupId>org.springframework.cloud</groupId>

    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

Enable Eureka Client:  
  
@SpringBootApplication

@EnableEurekaClient

public class ProductServiceApplication {

    public static void main(String[] args) {

        SpringApplication.run(ProductServiceApplication.class, args);

    }

}  
  
In application.yml:  
  
server:

  port: 8081

spring:

  application:

    name: product-service

eureka:

  client:

    service-url:

      defaultZone: http://localhost:8761/eureka/  
  
**3. Service Discovery Using RestTemplate or Feign**

**Using RestTemplate:**

@Autowired

private RestTemplate restTemplate;

public String getOrderDetails() {

    return restTemplate.getForObject("http://order-service/orders", String.class);

}

**Make sure to annotate RestTemplate with @LoadBalanced:**@Bean

@LoadBalanced

public RestTemplate restTemplate() {

    return new RestTemplate();

}  
  
**How Eureka Service Discovery Works**

1. Service Registration

Each microservice (e.g., order-service, product-service) registers itself with the Eureka Server when it starts up.

In its application.yml:

spring:

  application:

    name: order-service

eureka:

  client:

    service-url:

      defaultZone: http://localhost:8761/eureka/  
  
This means:

* The service will **send a heartbeat** to http://localhost:8761/eureka/ to register itself.
* It uses the spring.application.name (order-service) as its **service ID**.
* **2. Service Discovery**
* When another service (e.g., product-service) wants to call order-service, it doesn't need the actual IP or port. Instead, it uses the **logical name**:

restTemplate.getForObject("http://order-service/orders", String.class);

This works because:

* RestTemplate is annotated with @LoadBalanced, which enables **client-side load balancing** using **Ribbon** (or Spring Cloud LoadBalancer).
* Eureka provides a list of instances for order-service.
* Ribbon picks one instance and resolves the actual IP and port.

**3. Behind the Scenes**

Here's what happens step-by-step:

1. product-service starts and registers with Eureka.
2. It also **fetches the registry** from Eureka (list of all services).
3. When RestTemplate calls http://order-service/orders, it:
   * Looks up order-service in the registry.
   * Gets the actual IP and port (e.g., http://localhost:8082).
   * Makes the HTTP call.

---------------------------------------------------------------------------------------------------------  
**Circuit breaker pattern its usage and how it is implemented in microservices**  
  
The **Circuit Breaker Pattern** is a crucial design pattern in **microservices architecture** that helps improve system resilience and fault tolerance. Here's a breakdown of its **usage**, **benefits**, and **implementation**:  
  
**What is the Circuit Breaker Pattern?**

It’s inspired by electrical circuit breakers. In software systems, it prevents a service from repeatedly trying to execute an operation that's likely to fail—especially when a downstream service is unavailable or slow.

**Why Use It in Microservices?**

Microservices often depend on other services. If one service fails or becomes slow, it can cause cascading failures. The circuit breaker pattern helps by:

* **Preventing resource exhaustion** (e.g., thread pools, memory)
* **Failing fast** instead of waiting for timeouts
* **Allowing recovery** by retrying after a cooldown period
* **Improving system stability and responsiveness**

**States of a Circuit Breaker**

1. **Closed**: Requests flow normally. If failures exceed a threshold, it transitions to **Open**.
2. **Open**: Requests are blocked immediately. After a timeout, it transitions to **Half-Open**.
3. **Half-Open**: A limited number of requests are allowed. If they succeed, it goes back to **Closed**; if they fail, it returns to **Open**.

**Example with Resilience4j (Spring Boot)**@Bean

public CircuitBreakerRegistry circuitBreakerRegistry() {

    return CircuitBreakerRegistry.ofDefaults();

}

@CircuitBreaker(name = "myService", fallbackMethod = "fallbackMethod")

public String callExternalService() {

    // Call to external microservice

}

public String fallbackMethod(Throwable t) {

    return "Fallback response";

#### } **3. Configuration Options**

* Failure rate threshold
* Wait duration in open state
* Permitted calls in half-open state
* Timeout duration

**Monitoring and Metrics**

Integrate with tools like:

* **Prometheus + Grafana**
* **Spring Boot Actuator**
* **ELK Stack**

------------------------------------------------------------------------------------------------------------  
 **complete example** of how to implement the **Circuit Breaker Pattern** in a **Spring Boot application** using **Resilience4j** and **Spring Boot Actuator** for monitoring.  
**Step-by-Step Implementation**

**1. Add Dependencies**

In your pom.xml:

<dependencies>

    <!-- Spring Boot Starter Web -->

    <dependency>

        <groupId>org.springframework.boot</groupId>

        <artifactId>spring-boot-starter-web</artifactId>

    </dependency>

    <!-- Resilience4j Circuit Breaker -->

    <dependency>

        <groupId>io.github.resilience4j</groupId>

        <artifactId>resilience4j-spring-boot2</artifactId>

    </dependency>

    <!-- Spring Boot Actuator -->

    <dependency>

        <groupId>org.springframework.boot</groupId>

        <artifactId>spring-boot-starter-actuator</artifactId>

    </dependency>

</dependencies>  
  
---------------------------------------------------------------------------------------------------------  
**2. Create a Service with Circuit Breaker**

@CircuitBreaker(name = "externalService", fallbackMethod = "fallbackResponse")

public String callExternalService() {

    // This might fail due to network issues, timeouts, etc.

    return restTemplate.getForObject("http://some-service/api", String.class);

}

public String fallbackResponse(Throwable t) {

    // This gets called when the above method fails or circuit is open

    return "Service is currently unavailable. Please try again later.";

}  
  
**Explanation:**

* @CircuitBreaker(name = "externalServiceCB"): Tells Resilience4j to apply circuit breaker logic.
* fallbackResponse(Throwable t): Called when the circuit is open or the method fails.

**3. Create a REST Controller**import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController

public class ApiController {

    private final ExternalService externalService;

    public ApiController(ExternalService externalService) {

        this.externalService = externalService;

    }

    @GetMapping("/test")

    public String testCircuitBreaker() {

        return externalService.callExternalApi();

    }

}  
  
**4. Configure Circuit Breaker in application.yml**resilience4j:

  circuitbreaker:

    instances:

      externalServiceCB:

        registerHealthIndicator: true

        slidingWindowSize: 5

        failureRateThreshold: 50

        waitDurationInOpenState: 10s

        permittedNumberOfCallsInHalfOpenState: 2

        minimumNumberOfCalls: 5

        automaticTransitionFromOpenToHalfOpenEnabled: true

management:

  endpoints:

    web:

      exposure:

        include: health,info,metrics

**Explanation of config:**

* failureRateThreshold: If 50% of calls fail, the circuit opens.
* waitDurationInOpenState: Time to wait before transitioning to half-open.
* permittedNumberOfCallsInHalfOpenState: Number of test calls allowed.
* registerHealthIndicator: Enables health check via Actuator.

**5. Monitor with Actuator**

Once running, you can access:

* **Health endpoint**: http://localhost:8080/actuator/health
* **Metrics endpoint**: http://localhost:8080/actuator/metrics/resilience4j.circuitbreaker.calls

These show the state of the circuit breaker and metrics like success/failure rates.  
  
**✅ Summary**

This setup:

* Protects your service from cascading failures.
* Automatically retries after a cooldown.
* Provides observability via Spring Boot Actuator.