Q.1)Asynchronous Communication – 4 Key Approaches with Code

#	Method	Use Case	Importance
1️⃣	@Async	Send OTP, PAN verify	✅ Easy, already covered
2️⃣	Kafka (Message Broker)	Loan Sanction → Notification	⭐ Most common in microservices
3️⃣	Spring Events	Document Upload → Audit Trigger	✅ Lightweight in single service
4️⃣	WebFlux (Reactive)	Audit Logs Streaming, Credit Report API	⭐ For advanced async/rest apps

1️⃣ Real Use Case 1: Send OTP (non-blocking)

📦 1. Add Required Dependencies in pom.xml

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-web</artifactId>
</dependency>

<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter</artifactId>
</dependency>
```

✅ 2. Enable Async in Main Class

```
java
CopyEdit
@SpringBootApplication
@EnableAsync  // Important for enabling async behavior
public class LoanApplication {
   public static void main(String[] args) {
      SpringApplication.run(LoanApplication.class, args);
   }
}
```

⚙️ **3. Service Class with @Async**

```
import org.springframework.scheduling.annotation.Async;
import org.springframework.stereotype.Service;

@Service
public class OtpService {

    @Async
    public void sendOtp(String mobileNumber) {
        // simulate delay (e.g., API call to SMS service)
        try {
            System.out.println("⏳ Sending OTP to " + mobileNumber);
            Thread.sleep(3000); // Simulate delay
            System.out.println("✅ OTP Sent to " + mobileNumber);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

🎮 **4. Controller Triggering OTP in Background**

```
@RestController
@RequestMapping("/api/otp")
public class OtpController {

    @Autowired
    private OtpService otpService;

    @GetMapping("/send/{mobile}")
    public ResponseEntity<String> sendOtp(@PathVariable String mobile) {
        otpService.sendOtp(mobile);  // This runs asynchronously
        return ResponseEntity.ok("OTP request received. It will be sent shortly.");
```

```
    }
}
```

✅ **Output (Console Log)**

Request comes → API returns instantly:

"OTP request received"

In background:

⏳ Sending OTP to 9876543210

✅ OTP Sent to 9876543210

2️⃣ **Kafka – Microservice to Microservice Async**

✅ **Use Case:**

Loan is sanctioned → Push event to Kafka → Notification Service listens and sends Email/SMS

---

1. Add Kafka Dependencies (in pom.xml)

```
<dependency>
  <groupId>org.springframework.kafka</groupId>
  <artifactId>spring-kafka</artifactId>
</dependency>
```

---

2. Kafka Config (KafkaConfig.java)

```
@Configuration
public class KafkaConfig {

    @Bean
    public ProducerFactory<String, String> producerFactory() {
        return new DefaultKafkaProducerFactory<>(Map.of(
            ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092",
            ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class,
            ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class
```

```
        ));
    }

    @Bean
    public KafkaTemplate<String, String> kafkaTemplate() {
        return new KafkaTemplate<>(producerFactory());
    }
}
```

---

3. Kafka Producer (in Loan Service)

```
@Service
public class LoanService {

    @Autowired
    private KafkaTemplate<String, String> kafkaTemplate;

    public void sanctionLoan(String loanId) {
        // Your loan sanction logic...
        kafkaTemplate.send("loan-topic", "Loan Sanctioned with ID: " + loanId);
    }
}
```

---

4. Kafka Listener (in Notification Service)

```
@Service
public class NotificationListener {

    @KafkaListener(topics = "loan-topic", groupId = "loan-group")
    public void listen(String message) {
        System.out.println("📩 Notification Triggered: " + message);
        // send email or SMS logic
    }
```

}

Example 2 Document Communicate to Sanction Service

✅ DocumentService → Kafka topic → SanctionService

✅ **DocumentService → Kafka topic → SanctionService**

java

CopyEdit

// DocumentService - Producing Kafka message

@Autowired

private KafkaTemplate<String, DocumentEvent> kafkaTemplate;

public void uploadAndNotify(Document document) {

kafkaTemplate.send("document-verified-topic", new DocumentEvent(document.getCustomerId(), "VERIFIED"));

}

✅ **SanctionService - Kafka Listener**

java

CopyEdit

@KafkaListener(topics = "document-verified-topic", groupId = "loan-group")

public void listen(DocumentEvent event) {

if ("VERIFIED".equals(event.getStatus())) {

// proceed to sanction the loan

System.out.println("Document verified for customer: " + event.getCustomerId());

}

}

3.Spring Events – Internal Async Communication

Customer → Upload Document API → Upload to S3 → Publish DocumentUploadedEvent

↓

Listener handles it → Save Audit Log

✅ **1. Document Upload Controller**

```
@RestController
@RequestMapping("/documents")
public class DocumentController {

    @Autowired private DocumentService documentService;

    @PostMapping("/upload")
    public ResponseEntity<String> uploadDoc(@RequestParam("ssn") String ssn,
                                            @RequestParam("file") MultipartFile file) {
        documentService.uploadDocument(ssn, file);
        return ResponseEntity.ok("Document uploaded successfully");
    }
}
```

✅ **2. Document Service – Upload to S3 + Publish Event**

```
@Service
public class DocumentService {

    @Autowired private S3Service s3Service;
    @Autowired private ApplicationEventPublisher publisher;

    public void uploadDocument(String ssn, MultipartFile file) {
        String s3Url = s3Service.uploadToS3(file, ssn);

        // Publish Event After Upload
        publisher.publishEvent(new DocumentUploadedEvent(this, ssn, s3Url));
```

```
    }
}
```

✅ 3. S3Service – Upload File to AWS S3

```
@Service
public class S3Service {

    @Autowired private AmazonS3 amazonS3;

    private final String bucketName = "loan-documents-bucket";

    public String uploadToS3(MultipartFile file, String ssn) {
        String key = "documents/" + ssn + "/" + file.getOriginalFilename();
        try {
            amazonS3.putObject(new PutObjectRequest(bucketName, key, file.getInputStream(), null));
        } catch (Exception e) {
            throw new RuntimeException("S3 upload failed", e);
        }
        return amazonS3.getUrl(bucketName, key).toString();
    }
}
```

✅ 4. DocumentUploadedEvent Class

```
public class DocumentUploadedEvent extends ApplicationEvent {
    private String ssn;
    private String documentUrl;

    public DocumentUploadedEvent(Object source, String ssn, String documentUrl) {
        super(source);
        this.ssn = ssn;
        this.documentUrl = documentUrl;
```

```
    }

    public String getSsn() { return ssn; }
    public String getDocumentUrl() { return documentUrl; }
}
```

✅ **5. Audit Listener (Async)**

```
@Service
public class DocumentAuditListener {

    @Async
    @EventListener
    public void handleDocumentUploaded(DocumentUploadedEvent event) {
        System.out.println("📄 Audit Log: Document uploaded for SSN: " + event.getSsn());
        System.out.println("Document URL: " + event.getDocumentUrl());

        // You can save to DB or call another service here
    }
}
```

✅ **6. Enable Async Support**

```
@SpringBootApplication
@EnableAsync
public class LoanApplication {
    public static void main(String[] args) {
        SpringApplication.run(LoanApplication.class, args);
    }
}
```

✅ **Benefits of Spring Events in This Use Case:**

Feature	Benefit
✅ Decoupling	Upload logic and audit logic are separate
✅ Async	Doesn’t block upload flow
✅ Easy to Extend	Add more listeners without touching main code
✅ Real-time Audit	Track who uploaded what and when

🧠 **Final Interview Answer:**

Sir, in our Document Service, once a user uploads a document (like PAN), we store it in AWS S3 using Amazon SDK.
After successful upload, we **publish a Spring Event** called DocumentUploadedEvent with SSN and document URL.
This event is handled by an @Async @EventListener method, which creates an **audit log** entry in background.
This helps us keep services **modular**, **non-blocking**, and ready for future enhancements like sending alerts or emails.

4️⃣ **Spring WebFlux (Reactive + Non-blocking)**

✅ **Use Case:**

CIBIL Credit Score check or audit log streaming — response stream can be large or slow

---

1. Add Dependency

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

---

2. Reactive Controller

```
@RestController
@RequestMapping("/api/credit")
public class CreditController {

    @GetMapping(value = "/score", produces = MediaType.TEXT_EVENT_STREAM_VALUE)
```

```
    public Flux<String> getScore() {
        return Flux.just("Fetching CIBIL Score...")
                .concatWith(Mono.delay(Duration.ofSeconds(2)).flatMapMany(x -> Flux.just("Score: 758")))
                .concatWith(Flux.just("Done"));
    }
}
```

➞ This gives a **non-blocking response** streaming chunks like:

Fetching CIBIL Score...

(wait 2s)

Score: 758

Done

Why WebFlux was introduced when we had @Async?

@Async = **Multi-threading**
But **each thread = memory** → slow under heavy load

⚠️ So if 1000 users come and each thread blocks waiting for a response → system crashes

✅ Reactive is non-blocking and event-loop based (like Node.js)

One thread = many users handled in async fashion

Q(2). 🔷 1. Synchronous Communication (REST API)

📌 Used when: Immediate response is needed (e.g., Login, ETB check, BRMS eligibility check)

🧠 How it works:

- Service A makes a direct HTTP/REST call to Service B.
- Service A waits for the response.
- If Service B is slow or down → timeout/failure.

🧪 Example: Loan Service calling BRMS Service synchronously

✅ LoanService → calls → BRMSService

```
@FeignClient(name = "brms-service")
public interface BrmsClient {
    @PostMapping("/brms/eligibility")
```

```
    BrmsResponse checkEligibility(@RequestBody LoanRequest request);
}
@Service
public class LoanService {

    @Autowired
    private BrmsClient brmsClient;

    public LoanResponse applyLoan(LoanRequest request) {
        BrmsResponse response = brmsClient.checkEligibility(request);
        if (response.isEligible()) {
            // proceed with loan creation
        }
        return new LoanResponse("Loan rejected", false);
    }
}
```

Q4.Aapke Vehicle/Personal **Loan Project ke APIs — Interview Ready List**

A. Internal Microservice APIs (aapke team/project ke andar)

Service	**Endpoint**	**Type**	**Use**
ETB/NTB Check	/api/etb/check?pan=ABCDE1234F	GET	Check if customer exists
Account Service	/api/account/create	POST	Create account for NTB
BRMS Rule Engine	/api/brms/rules	POST	Eligibility check
Loan Sanction	/api/loan/sanction	POST	Sanction logic
Repayment Plan	/api/repayment/calculate	POST	Monthly EMI calculation
Document Service	/api/document/upload	POST	Upload Aadhaar/PAN
Auth Service	/api/auth/login	POST	Login with username/password

Interview Line:

I worked on APIs like ETB/NTB check using PAN, customer account creation (for NTB), BRMS rule check to decide loan eligibility, and document upload logic using multi-part file upload.

B. Third-Party API Calls (integrated with outside system)

API	Description	Sync/Async	Use Case
PAN Verification (Govt/NSDL)	https://pan-api.com/verify	🔁 Async	Validate PAN is correct
SSN Verification (US)	https://gov-ssn.com/check	🔁 Async	Validate SSN before loan
CIBIL/Experian Score	https://cibil.com/score	🔁 Async	Get credit score
SMS Gateway (OTP)	https://sms-api.com/send	🔁 Async	Send OTP
Email API (Mailgun/SendGrid)	https://email-api.com/send	🔁 Async	Loan status mail
DMS (Document Storage)	https://dms-service/upload	🔁 Async	Upload Aadhaar/PAN

Interview Line:

I integrated third-party APIs like SSN verification, CIBIL credit check, Experian Check, and used external services like SMS and email notification APIs.
All of them were handled asynchronously with error handling and fallback logic.

Q3.What is the difference between Redis Cache and Normal Cache?

Feature	Normal Cache (In-Memory)	Redis Cache (Distributed)
📍 Location	Inside JVM (Heap memory)	External In-memory DB
🌐 Scope	Only within one microservice	Shared across microservices
🔄 Lifetime	Until app restarts or memory is cleared	Persistent (if needed), survives restarts
⚙️ Tool	@Cacheable, ConcurrentHashMap, Guava	Redis with @Cacheable, RedisTemplate, Lettuce
🧠 Used For	Small-scale, one service caching	Microservices, API results, session/token store
⚠️ Limits	Memory-bound, no TTL control	Scalable, TTL, eviction, pub-sub
🔐 Secure	Inside app	Needs Redis config & port security

Use Case in Your Loan Project

Use Case	Cache Type	Why
BRMS Rules – static master config	✅ Normal Cache	No DB hit, fixed data
PAN Verification result (for 10 min)	✅ Redis	External call avoid
CIBIL Score for 15 mins	✅ Redis	Costly API, reduce calls
Aadhaar OTP session	✅ Redis	Shared across services
Rate of Interest table (daily)	✅ Normal	Rarely changes

What Problem Are They Solving?

✅ **Problem**: Repeated DB/API calls for same data
✅ **Solution**: Cache stores recent/frequent data → reduces load + latency

In our loan project, we used two types of caching:

- **In-Memory Cache** (@Cacheable) for static data like BRMS rules and interest rates
- **Redis Cache** for PAN verification and CIBIL score – because these are heavy external APIs and don't change frequently

Redis was chosen because it supports TTL, cross-service sharing, and is highly performant for read-heavy data.

For example, once a PAN is verified, we store the result in Redis for 10 minutes using @Cacheable, so that next time we don't hit the external PAN verification API again.

✅ **Redis Cache – With Spring Boot**

🔷 **1. Add Redis Dependency**

xml

CopyEdit

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

🔷 **2. Redis Config in application.properties**

```
spring.cache.type=redis
spring.redis.host=localhost
```

spring.redis.port=6379

Redis must be running locally on port 6379
You can use Docker or Redis installer

🔷 **3. Enable Redis + TTL**

```
@Configuration
@EnableCaching
public class RedisConfig {

    @Bean
    public RedisCacheConfiguration cacheConfig() {
        return RedisCacheConfiguration.defaultCacheConfig()
            .entryTtl(Duration.ofMinutes(10))  // Default TTL
            .disableCachingNullValues();
    }
}
```

🔷 **4. Use @Cacheable with Redis**

```
@Service
public class CibilService {

    @Cacheable(value = "cibilScores", key = "#panNumber")
    public Integer getCibilScore(String panNumber) {
        // slow API call
        return 758;  // simulate score
    }
}
```

Using Redis Cache in Microservices:

- Redis is used to cache frequently accessed data to improve performance and reduce DB load.
- Example: ETB/NTB Check Service caches the customer existence status for a short TTL.

```
@Configuration
@EnableCaching
public class RedisConfig {
   @Bean
   public RedisConnectionFactory connectionFactory() {
      return new LettuceConnectionFactory();
   }

   @Bean
   public RedisTemplate<String, Object> redisTemplate() {
      RedisTemplate<String, Object> template = new RedisTemplate<>();
      template.setConnectionFactory(connectionFactory());
      return template;
   }
}

@Service
public class EtbService {

   @Cacheable(value = "customerCheck", key = "#pan")
   public boolean isExistingCustomer(String pan) {
      return repository.existsByPan(pan);
   }
}
```

Redis Interview Questions Based on This Project

Q: Where did you use Redis in your project and why?
A: We used Redis in the ETB/NTB Check Service and BRMS Service. Customer existence and rules were cached for short TTL to reduce DB calls, improve latency, and handle traffic spikes.

Q: What are the benefits of caching in microservices?
A:

- Reduces latency for frequent reads
- Minimizes database load
- Improves scalability
- Helps avoid rate-limiting on external systems

Q: How do you invalidate cache in Spring Boot?
A: Using @CacheEvict, either on update or scheduled jobs.

```
@CacheEvict(value = "customerCheck", key = "#pan")
public void updateCustomerStatus(String pan) {
   // update DB
}
@Bean
public CacheManager cacheManager() {
   RedisCacheConfiguration config = RedisCacheConfiguration
                .defaultCacheConfig()
               .entryTtl(Duration.ofMinutes(5));
   return RedisCacheManager.builder(connectionFactory()).cacheDefaults(config).build();
}
```

Q: What happens if Redis is down?
A:

- Application should handle fallback logic (query DB directly)
- Application still works → it just fetches data from DB instead of Redis.
- Redis is optional for functionality but critical for performance

Q. Can Redis be used for sharing data between services?

Not recommended. Redis should be used for **caching** not **communication**.

For inter-service communication, use:

- ✅ Kafka (asynchronous)
- ✅ REST API (synchronous)

Note→We used Redis in the ETB/NTB Check Service to cache PAN validation results for a short TTL (e.g., 5 minutes). This reduced DB hits during peak hours, improved user experience by responding faster, and ensured fresh data by auto-expiring cache entries. Redis helped us achieve both **performance and accuracy**.

Q.A third-party API is slow. How do you handle it?

Technique	**Purpose**
✅ Timeout setting	Stop waiting after N seconds
✅ Retry	Try again 2–3 times if it fails
✅ Circuit breaker	Stop calling a broken API temporarily
✅ Fallback	Give a default response if API is too slow
✅ Async Calls	Don't block the user, handle later (Kafka)

```
<dependency>
   <groupId>io.github.resilience4j</groupId>
   <artifactId>resilience4j-spring-boot2</artifactId>
</dependency>
```

```
@Service
public class CibilService {

   @CircuitBreaker(name = "cibilApi", fallbackMethod = "fallbackCibil")
   @Retry(name = "cibilApi", maxAttempts = 3)
   @TimeLimiter(name = "cibilApi")
   public CompletableFuture<String> fetchCibilScore(String pan) {
      return CompletableFuture.supplyAsync(() -> {
         // Call to slow 3rd-party API
         return restTemplate.getForObject("https://thirdparty.com/cibil?pan=" + pan, String.class);
```

```
        });
    }


    // If API fails or is too slow
    public CompletableFuture<String> fallbackCibil(String pan, Throwable t) {
        return CompletableFuture.completedFuture("CIBIL API down. Please try again later.");
    }
}
```

Q. How do you handle timeouts, retries, and circuit breakers?

Timeouts: We configure connection and read timeouts in our RestTemplate or WebClient to ensure slow APIs don't block the system.

```
@Bean
public RestTemplate restTemplate() {
    HttpComponentsClientHttpRequestFactory factory = new
HttpComponentsClientHttpRequestFactory();
    factory.setConnectTimeout(2000);  // 2 seconds
    factory.setReadTimeout(3000);     // 3 seconds
    return new RestTemplate(factory);
}
```

Retries: We use Spring Retry for retrying failed calls due to temporary network issues.

```
@Retryable(value = Exception.class, maxAttempts = 3, backoff = @Backoff(delay = 2000))
public String callExternalApi() {
    // API call logic
}
```

Circuit Breaker: We use Resilience4j to stop calling a service that’s continuously failing. A fallback method is provided to return a safe response.

```
@CircuitBreaker(name = "loanService", fallbackMethod = "fallbackMethod")
public String callLoanService() {
    // API call logic
```

```
}

public String fallbackMethod(Throwable t) {
   return "Service is temporarily unavailable. Please try later.";
}
```

Q. One microservice fails. What happens?

In our microservices architecture, if one service like Document or Repayment fails, we handle it using circuit breakers and fallback logic. We also use Kafka for retry queues to make the system resilient and avoid user impact.

➡️ **Before Failure:**

- User completes form → Authentication → ETB check → Loan service → Now at Document Upload step.

➡️ **Failure Case:**

- Document Service is down or unresponsive.
- Since it's a REST call, the **Loan Portal UI** shows an error message like "Service is currently unavailable. Try again later."

➡️ **Behind the scenes:**

- The API Gateway tried forwarding the request to Document Service, but it failed.
- **Circuit Breaker** (Resilience4j) prevents retry spam and returns a fallback response.

```
@CircuitBreaker(name = "documentService", fallbackMethod = "fallbackDocUpload")
public String uploadDocs(MultipartFile file) {
   // actual API call to Document Service
}

public String fallbackDocUpload(Throwable t) {
   return "Document service is temporarily down.";
}
```

➡️ **Impact:**

- Because our system is built with fault tolerance, the failure doesn’t crash the entire process.
- We show user-friendly errors and continue from that step when the service is back.

Q.How do you ensure rollback or zero downtime deployment?

We follow blue-green or rolling deployment using Jenkins and AWS. We run health checks and use feature flags and versioned APIs to ensure safe rollback and zero downtime

1. **Blue-Green Deployment:**
    - **We maintain two environments: Blue (active) and Green (idle).**
    - **Deploy new version to Green → Health Check → Switch traffic to Green.**
    - **If anything breaks, switch back to Blue immediately.**
2. **Rolling Deployment (in Kubernetes):**
    - **Services are updated one pod at a time.**
    - **Traffic is routed only to healthy pods.**
    - **Prevents full system crash during deployment.**
3. **Health Checks & Readiness Probes:**
    - **Configured in EC2/ALB or Kubernetes.**
    - **Only healthy containers receive traffic.**
4. **Feature Flags (Toggles):**
    - **New features can be enabled/disabled dynamically without redeployment.**
    - **Helps rollback partially if a specific feature causes issues.**
5. **Versioned APIs:**
    - **Older clients use /v1/api/... while newer ones move to /v2/api/...**
    - **Backward compatibility avoids runtime failure.**
6. **Git + Jenkins CI/CD Pipeline:**
    - **Code push triggers Jenkins pipeline → builds → runs tests → deploys to staging → then to prod with approval.**

Q. How do you validate a user token in downstream services?

In our microservices architecture, token validation is done in all downstream services using JWT validation logic. Here's how:

1. User logs in:
    - Auth Service issues a JWT token signed using HS256 and a secret key.
2. User calls any downstream service (e.g., Loan Service):
    - Token is sent in request header: Authorization: Bearer <token>

3. Loan Service (or any other) extracts and validates the token:

```
public Claims validateToken(String token) {
   return Jwts.parser()
      .setSigningKey(secret)
      .parseClaimsJws(token.replace("Bearer ", ""))
      .getBody();
}
```

4. **If token is valid:**
    - Proceed with request and extract user info from claims.
5. **If invalid or expired:**
    - Return 401 Unauthorized.
6. **We also use a JwtFilter to apply validation globally in downstream services:**

```
@Component
public class JwtFilter extends OncePerRequestFilter {


   @Override
   protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,
FilterChain filterChain)
         throws ServletException, IOException {


      String token = request.getHeader("Authorization");
      if (token != null && token.startsWith("Bearer ")) {
         try {
            Claims claims = validateToken(token);
            request.setAttribute("user", claims.getSubject());
         } catch (Exception e) {
            response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
            return;
         }
      }
      filterChain.doFilter(request, response);
```

```
    }
}
```

Note-→All our services validate the JWT token before processing any request. We use a global filter that extracts the token, validates it with our shared secret key, and proceeds only if the token is valid.

Q. How to send token in header using code (RestTemplate example):

```
HttpHeaders headers = new HttpHeaders();

headers.set("Authorization", "Bearer " + token);

HttpEntity<String> entity = new HttpEntity<>(headers);


ResponseEntity<String> response = restTemplate.exchange(
   "http://loan-service/api/loan/details",
   HttpMethod.GET,
   entity,
   String.class
);
```

Explanation:

- We create an HttpHeaders object and set the JWT token in the Authorization header.
- Then we wrap it inside an HttpEntity.
- This entity is passed to RestTemplate.exchange() to make an HTTP call with headers.

Q. Three Microservices is communicated each other and one services is not working which design pattern is used in thar case?

If one of three communicating services fails, we use Circuit Breaker and fallback to isolate the failure. If a service is down, we show the user a friendly message like 'Service is temporarily unavailable, please try again later', or show cached/last known data with a disclaimer. This ensures the system doesn't crash and keeps the user experience smooth until the service recovers.". We return default responses or redirect to Kafka queues to keep the system responsive and resilient.

1. **Circuit Breaker Pattern** – To prevent cascading failure.
2. **Retry Pattern** – Retry a few times before failing.
3. **Fallback Pattern** – Return default/cached response.
4. **Bulkhead Pattern** (optional) – Isolate resources per service/thread.

5. **Message Queue (Kafka)** – For async retry and decoupling if needed.

💥 **Scenario:**

Suppose three services are working together:

- **CustomerService → LoanService → RepaymentService**

If **LoanService** fails during a transaction:

- **CustomerService** will get an error or timeout.
- The entire request chain can break if not handled.

⚠️ **Problems Without Protection:**

- User sees error or timeout.
- Incomplete or inconsistent data.
- All services waiting for each other (cascading failure).

❌ **What If We Don't Use Circuit Breaker?**

If we don’t use Circuit Breaker and **LoanService** goes down:

- **CustomerService → LoanService → RepaymentService**
- CustomerService will keep trying to call LoanService on every request.
- These failed requests will **block threads**, cause delays, and increase resource usage.
- If multiple services behave this way, **it can crash the system** (cascading failure).
- Any downstream services depending on LoanService (like RepaymentService) **won’t get the required data**, and may also fail.

🔁 **Real Impact:**

- Users will keep getting timeouts or 500 errors.
- Other services waiting for responses will also slow down.
- In production, this can bring the system to a halt.

That’s why **Circuit Breaker is critical** — it cuts off failing service calls and gives fallback responses or queues them instead.

🛡️ **Solution: Use Circuit Breaker + Fallback Pattern**

We implement **Resilience4j** (or Hystrix) for **Circuit Breaker** to protect from total system failure.

🧱 **What Circuit Breaker Does:**

- Monitors failure rate.
- If a service fails repeatedly, it “opens the circuit”.

- Redirects traffic to fallback logic.
- Prevents overloading the failed service.

Here's how it works in real projects:

✅ **Case 1 – Fallback with a message (temporary failure)**
You return:

json

CopyEdit

{

"status": "Loan service temporarily unavailable. Please try again later."

}

✅ User knows there's a problem but UI doesn't break.

✅ **Case 2 – Fallback with cached/last-known data (Redis or DB)**

java

CopyEdit

public String fallbackLoan(String loanId, Exception ex) {

return redisTemplate.opsForValue().get("loan-status:" + loanId);

}

✅ User sees *last available info* instead of an error.

✅ **Case 3 – Fallback with Retry → Kafka Queue**

java

CopyEdit

// Retry failed request asynchronously later

kafkaTemplate.send("loan-retry-topic", requestObject);

return "Your request is in queue, status will update shortly.";

✅ Data consistency is maintained later through retry.

1.Circuit Breaker Pattern
@CircuitBreaker(name = "serviceB", fallbackMethod = "fallbackB")

```
public String callServiceB() {
   return restTemplate.getForObject("http://SERVICE-B/api", String.class);
}

public String fallbackB(Exception e) {
   return "Service B is currently unavailable. Please try later.";
}
```

2. **Retry Pattern**

Library: Resilience4j Retry or Spring Retry

```
@Retry(name = "serviceB", fallbackMethod = "fallbackB")
public String callServiceB() {
   // Retry logic
}
```

3. Saga Pattern (for data consistency)

Situation	Pattern to Use	Purpose
Service fails permanently	**Circuit Breaker**	**Avoid repeated failures, degrade gracefully**
Service fails temporarily	**Retry Pattern**	**Retry after short delay**
Long-running transaction fails	**Saga Pattern**	**Maintain data consistency using rollback**
Want to isolate failure impact	**Bulkhead Pattern**	**Prevent one service failure from affecting others**

If three services are communicating and one fails, we use **Circuit Breaker** to prevent cascading failures, **Retry** for transient errors, and **Saga Pattern** for maintaining **data consistency** across services.

1. **OrderService** calls **InventoryService first** to check product availability.
2. If stock exists, proceed to **PaymentService** to charge the customer.
3. Then call **DeliveryService** to dispatch.

Important ye explain kr dena interview mein:

🛡 **How It's Handled:**

- ✅ **Retry Pattern**: OrderService tries 2–3 times to call InventoryService.
- ✅ **Circuit Breaker**: If still down, circuit opens to prevent overload.
- ✅ **Fallback Pattern**: OrderService returns a message: "Your order is received. Inventory check pending."
- ✅ **Kafka + Saga Pattern**: OrderService sends event to inventory-check-failure-topic, and background saga starts rollback if needed.
    - Rollback Payment via Kafka event to payment-cancel-topic
    - Notify user via email/SMS

🧠 **Interview Line:**

“In our order flow, if any downstream service like InventoryService fails, we retry a few times, fallback with a safe message, and use Kafka for eventual rollback using the Saga pattern. This way, we preserve user experience and system consistency.”

Q. How to handle exception logging and custom error response in your rest api in springboot?

```
public class ErrorResponse {
    private String timestamp;
    private int status;
    private String error;
    private String message;
    private String path;

    // Constructor
    public ErrorResponse(int status, String error, String message, String path) {
        this.timestamp = java.time.LocalDateTime.now().toString();
```

```
        this.status = status;
        this.error = error;
        this.message = message;
        this.path = path;
    }

    // Getters and setters
}



import org.springframework.http.*;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.context.request.WebRequest;
import org.slf4j.*;

@ControllerAdvice
public class GlobalExceptionHandler {

    private static final Logger logger = LoggerFactory.getLogger(GlobalExceptionHandler.class);

    // Handle all uncaught exceptions
    @ExceptionHandler(Exception.class)
    public ResponseEntity<ErrorResponse> handleAllExceptions(Exception ex, WebRequest request) {
        logger.error("Unhandled Exception occurred: ", ex);  // logs full stack trace

        ErrorResponse response = new ErrorResponse(
            HttpStatus.INTERNAL_SERVER_ERROR.value(),
            "Internal Server Error",
            ex.getMessage(),
            request.getDescription(false)
        );
```

```
        return new ResponseEntity<>(response, HttpStatus.INTERNAL_SERVER_ERROR);
    }

    // Handle specific exception (example: Resource Not Found)
    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<ErrorResponse> handleResourceNotFound(ResourceNotFoundException ex,
WebRequest request) {
        logger.warn("Resource not found: {}", ex.getMessage());

        ErrorResponse response = new ErrorResponse(
            HttpStatus.NOT_FOUND.value(),
            "Not Found",
            ex.getMessage(),
            request.getDescription(false)
        );

        return new ResponseEntity<>(response, HttpStatus.NOT_FOUND);
    }

    // You can add more custom handlers for BadRequest, Validation, etc.
}

public class ResourceNotFoundException extends RuntimeException {
    public ResourceNotFoundException(String message) {
        super(message);
    }
}
{
  "timestamp": "2025-07-05T14:45:30.123",
  "status": 404,
```

"error": "Not Found",

"message": "Customer with ID 123 not found",

"path": "/api/customers/123"

}

Note→ I use @ControllerAdvice and @ExceptionHandler to handle exceptions globally and return structured, user-friendly error responses. I also log all errors using SLF4J for debugging without exposing sensitive stack trace info to the API clients

Q. JWT 3 Part Mein Hota Hai?

Part No.	Naam	Kya karta hai
1	Header	Batata hai konsa algorithm use hua
2	Payload	User ka data hota hai (email, role etc)
3	Signature	Prove karta hai token fake to nahi

🔍 1 **Header**

json

CopyEdit

{

"alg": "HS256",

"typ": "JWT"

}

Matlab: "Main HS256 algorithm se sign hua hoon"
👉 Isko encode karke ban jaata hai JWT ka pehla part.

👨💼 2 **Payload (Main Data)**

json

CopyEdit

{

"sub": "user@gmail.com",

"roles": ["ADMIN"],

"exp": 1719810600

}

🛡️ 3️⃣ **Signature (Sabse Important)**

Ye part token ko **secure banata hai**.

Server ye signature banata hai is formula se:

SIGNATURE = encode(Header + Payload + secret key)

👉 **Basic Answer:**

"Hum JWT tokens ke liye **HS256** algorithm ka use karte hain, jisme ek secret key hoti hai jo token ko sign karti hai. Ye signature validate karta hai ki token tamper nahi hua hai."

👉 **Advanced Answer (if needed):**

"HS256 ek symmetric algorithm hai. Agar future mein hume public-private key validation chahiye, to hum **RS256** jaise asymmetric algorithm use kar sakte hain."

JWT token ke **Header** + **Payload** ko mila ke ek **Signature** banta hai using HS256 algorithm.

🧪 **Signature Formula:**

text

CopyEdit

HMACSHA256(

base64UrlEncode(header) + "." + base64UrlEncode(payload),

secret_key

)

Q2. How do you decide the boundaries of a microservice?
A: Based on business capabilities and domain-driven design principles. Each service handles a specific domain like Customer, Account, Loan, or Repayment.

Q3. What is the role of API Gateway in your project?
A: It acts as a single entry point, handles routing, security (JWT verification), rate-limiting, CORS, and request aggregation.

Q4. How does your service handle service discovery?
A: Using Spring Cloud Eureka, each service registers with Eureka Server. API Gateway or clients discover them dynamically.

How Do You Validate a User Token in Downstream Services?

🔐 **Simple Explanation:**

1. **Authentication Service** generates a **JWT token** after successful login.
2. This token is sent in the **Authorization header** for all further requests.
3. Every **downstream microservice** (LoanService, AccountService, BRMS, etc.) contains a **JWT validation filter** (like JwtFilter).
4. The JwtFilter:
    - Extracts the token from the request
    - Verifies its signature and expiration
    - If valid, allows the request
    - If invalid, rejects with 401 Unauthorized

✅ **Where to Validate?**

✅ You place **JwtFilter in every microservice** (except Auth Service) using a **Spring Security filter or OncePerRequestFilter**.

🔧 **JwtFilter Example in Downstream Service (Loan Service)**

java

CopyEdit

```
@Component
public class JwtFilter extends OncePerRequestFilter {

    @Autowired
    private JwtUtil jwtUtil;

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,
FilterChain chain)
            throws ServletException, IOException {

        String authHeader = request.getHeader("Authorization");
        String token = null;
```

```
        String username = null;

        if (authHeader != null && authHeader.startsWith("Bearer ")) {
            token = authHeader.substring(7);
            username = jwtUtil.extractUsername(token);
        }

        if (username != null && SecurityContextHolder.getContext().getAuthentication() == null) {
            if (jwtUtil.validateToken(token)) {
                UsernamePasswordAuthenticationToken auth =
                        new UsernamePasswordAuthenticationToken(username, null, new ArrayList<>());
                SecurityContextHolder.getContext().setAuthentication(auth);
            }
        }

        chain.doFilter(request, response);
    }
}
```
