Double Payment Handling using an Idempotent API

Scenario:

User taps "Pay" -> the UI calls your backend -> network times out -> user taps again.
Or your payment gateway retries automatically.

Without proper handling, this can cause:

- **Double payments**
- **Incorrect transaction states**
- **Poor user trust**

✅ SOLUTION: Idempotent API

What is an Idempotent API?

An API where **repeating the same request** (with the same `idempotency key`) **results in the same outcome** (no duplicate side-effects).

🧱 SYSTEM DESIGN OVERVIEW

Components:

1. **Client (Mobile/Web)**
2. **Payment API (Backend)**
3. **Idempotency Key Store (DB or Redis)**
4. **Payment Gateway (External Service)**
5. **Database (Transactions, Orders)**

🧭 FLOW

Step-by-step Payment Flow with Idempotency:

1. **Client generates an `Idempotency-Key`** (UUID or hash of request).
2. Send a **Payment Request** with this key.
3. Backend checks:
    - If key already exists:
        - Return stored result (status, payment ID, etc.)
    - Else:
        - Create payment.
        - Store result against the key (in DB/Redis).
        - Return response.

📦 IMPLEMENTATION HIGHLIGHT

1. API Request:

```
POST /pay
Headers:
  Idempotency-Key: abc123
Body:
  { amount: 100, userId: 42 }
```

2. Backend Logic:

```
String idempotencyKey = request.getHeader("Idempotency-Key");

if (idempotencyStore.exists(idempotencyKey)) {
    return idempotencyStore.get(idempotencyKey);
} else {
    PaymentResult result = processPayment(request);
    idempotencyStore.save(idempotencyKey, result);
    return result;
}
```

3. Idempotency Store Options:

- Relational DB: Table with `idempotency_key`, `response`, `timestamp`
- Redis: TTL-based fast lookup (recommended for high traffic)

⏱️ EXPONENTIAL BACKOFF

Why Needed?

- Network retries on failure (e.g. payment gateway slow)
- Without backoff → high load / duplicate attempts

Solution:

Use **Exponential Backoff + Jitter**:

```
// Pseudo-code
retryInterval = base * 2^attempt + random_jitter;
```

Example: 100ms, 200ms, 400ms ± jitter

Benefits:

- Prevents retry storms
- Reduces resource contention

⚠️ THUNDERING HERD PROBLEM

Scenario:

Multiple instances/clients retry the same request at once (e.g., server restart, burst retry).

Solution:

- **Distributed Lock** per idempotency key:
    - Only one thread processes the payment
    - Others wait for lock or return cached result

Redis-based Locking:

```
SET key "locked" NX PX 5000  // Acquires lock for 5 seconds
```

📊 ARCHITECTURE DIAGRAM

```
    +-------------+
          |   Client    |
          +-------------+
                 |
                 v
        [POST /pay with Idempotency-Key]
                 |
                 v
     +------------------------+
     |     API Gateway        |
     +------------------------+
                 |
                 v
       +------------------+
       | Payment Service  |
       +------------------+
         |           |
         |           v
         |     +------------------+
         |     | Idempotency Store|
         |     +------------------+
         |        (Redis/DB)
         |
         v
+-------------------------+
| External Payment Gateway|
+-------------------------+
         |
         v
+-------------------------+
|   Orders/Payments DB    |
+-------------------------+
```

✅ SUMMARY OF KEY DESIGN POINTS

Concern	Design
Double Payment	Use `Idempotency-Key` per request
Retry Management	Exponential Backoff with Jitter
Thundering Herd	Redis lock per key or message queue throttling
Storage	Store result of idempotent operations in DB or Redis
Thread Safety	Lock around idempotency key (optional but better under scale)

💡 BONUS: Tips for Production Readiness

- Set TTL on `Idempotency-Key` storage to auto-clean old entries.
- Log and monitor retries + failed attempts.
- Consider rate limiting clients abusing retry logic.
- Use strong uniqueness for idempotency key (UUID or hash of body).

Here's a complete Spring Boot implementation of an Idempotent Payment API in Java, with:

- Idempotency key validation
- Redis-based cache store
- Distributed lock to avoid thundering herd
- Simulated payment processing
- Response caching

🔧 Project Structure

```
src/main/java/com/example/payment
├── controller
│   └── PaymentController.java
├── service
│   ├── PaymentService.java
│   └── IdempotencyService.java
├── model
│   └── PaymentRequest.java
├── config
│   └── RedisConfig.java
```

🔧 Redis Configuration (`RedisConfig.java`)

```
@Configuration
```

```
public class RedisConfig {
    @Bean
    public RedisTemplate<String, String> redisTemplate(RedisConnectionFactory
factory) {
        RedisTemplate<String, String> template = new RedisTemplate<>();
        template.setConnectionFactory(factory);
        return template;
    }
}
```

📦 Model (`PaymentRequest.java`)

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class PaymentRequest {
    private String userId;
    private double amount;
}
```

🔐 Idempotency Service (`IdempotencyService.java`)

```
@Service
public class IdempotencyService {
    @Autowired
    private RedisTemplate<String, String> redisTemplate;

    private static final String IDEMPOTENT_KEY_PREFIX = "idem:key:";
    private static final String LOCK_KEY_PREFIX = "lock:key:";
    private static final long TTL = 600; // seconds

    public boolean isDuplicate(String key) {
        return redisTemplate.hasKey(IDEMPOTENT_KEY_PREFIX + key);
    }

    public void storeResponse(String key, String responseJson) {
        redisTemplate.opsForValue().set(IDEMPOTENT_KEY_PREFIX + key,
responseJson, TTL, TimeUnit.SECONDS);
    }

    public String getResponse(String key) {
        return redisTemplate.opsForValue().get(IDEMPOTENT_KEY_PREFIX + key);
    }
    public boolean acquireLock(String key) {
        String lockKey = LOCK_KEY_PREFIX + key;
        Boolean success = redisTemplate.opsForValue().setIfAbsent(lockKey,
"locked", Duration.ofSeconds(5));
        return Boolean.TRUE.equals(success);
    }

    public void releaseLock(String key) {
        redisTemplate.delete(LOCK_KEY_PREFIX + key);
    }
}
```

💰 Payment Service (`PaymentService.java`)

```
public class PaymentService {

 @Retryable(
     value = {RuntimeException.class},
     maxAttempts = 3,
     backoff = @Backoff(delay = 1000, multiplier = 2)
 )
 public String process(PaymentRequest request) {
    throw new RuntimeException("Simulated transient error");
 }

 @Recover
 public String recover(RuntimeException ex, PaymentRequest request) {
   System.out.println("Recovered from: " + ex.getMessage());
   // fallback if all retries exhausted
   return "Payment successful for user " + request.getUserId() + " of amount
₹" + request.getAmount();
 }
```

🎯 Controller (`PaymentController.java`)

```
@RestController
@RequestMapping("/payment")
public class PaymentController {

    @Autowired
    private PaymentService paymentService;

    @Autowired
    private IdempotencyService idempotencyService;
    @PostMapping
    public ResponseEntity<String> pay(@RequestHeader("Idempotency-Key") String idempotencyKey,
                                      @RequestBody PaymentRequest request) {
        if (idempotencyService.isDuplicate(idempotencyKey)) {
            String cached = idempotencyService.getResponse(idempotencyKey);
            return ResponseEntity.ok("✅ Duplicate Request - Returning Cached Result: " + cached);
        }

        if (!idempotencyService.acquireLock(idempotencyKey)) {
            return ResponseEntity.status(HttpStatus.CONFLICT).body("⚠️ Another request in progress");
        }

        try {
            String result = paymentService.process(request);
            idempotencyService.storeResponse(idempotencyKey, result);
            return ResponseEntity.ok(result);
        } finally {
            idempotencyService.releaseLock(idempotencyKey);
        }
    }
}
```

🧠 Diagram Recap

```
Client ---> [POST /payment + Idempotency-Key]
            |
            v
      +-------------------------+
      |  PaymentController      |
      +-------------------------+
        |        |         |
        |     Check Key    |
        |        |         |
        v        v         v
  IdempotencyService ----> Redis Cache
        |
        v
  PaymentService ---> External Gateway
        |
        v
  Store Response in Redis (with TTL)
```

✅ Testing

Curl Example:

```
curl -X POST http://localhost:8080/payment \
  -H "Content-Type: application/json" \
  -H "Idempotency-Key: test1234" \
  -d '{"userId":"shivang","amount":99.99}'
```

Re-run same curl → it returns cached response.

Here is the code sample on github.
https://github.com/shivangbtech/IdempotentPaymentAPI