



**University of
East London**

CSE434 Aspect and Service-Oriented Software Systems

Lab 4 Assignment

Name	Mohamed Mostafa Bedair ElMaghraby
ID	20P7732

1- Models

```
5  @Entity 17 usages
6  @Table(name = "room")
7  public class Room {
8      @Id 2 usages
9      @GeneratedValue(strategy = GenerationType.IDENTITY)
10     private int id;
11     private String number; 3 usages
12     private int capacity; 3 usages
13     private double pricePerNight; 3 usages
14     private boolean isAvailable; 3 usages
15
16     public Room() { 1 usage
17     }
```

2- DTOs

```
5
6  public class CreateRoomDTO { 4 usages
7
8      @NotBlank(message = "Room number is required") 2 usages
9      @UniqueRoomNumber
10     private String number;
11
12     @NotNull(message = "Capacity is required") 2 usages
13     @Min(value = 1, message = "Capacity must be at least 1")
14     private Integer capacity;
15
16     @NotNull(message = "Price per night is required") 2 usages
17     @Positive(message = "Price must be positive")
18     private Double pricePerNight;
19
20     @NotNull(message = "Availability must be specified") 2 usages
21     private Boolean isAvailable;
22
23     // Getters & Setters
24     public String getNumber() { 1 usage
25         return number;
26     }
```

```
public class UpdateRoomDTO { 4 usages

    @UniqueRoomNumber 2 usages
    private String number;

    @Min(value = 1, message = "Capacity must be at least 1") 2 usages
    private Integer capacity; | 1

    @Positive(message = "Price must be positive") 2 usages
    private Double pricePerNight;

    private Boolean isAvailable; 2 usages
```

3- Controllers

```
4 @RequestMapping("/api/rooms")
5 public class RoomController {
6
7     @Autowired 5 usages
8     private RoomService roomService;
9
10    @GetMapping no usages
11    @RateLimit(limit = 5, duration = 60, keyPrefix = "getAllRooms")
12    public ResponseEntity<List<Room>> getAllRooms() { return ResponseEntity.ok(roomService.getAllRooms()); }
13
14    @GetMapping("/{id}") no usages
15    public ResponseEntity<Room> getRoomById(@PathVariable int id) {
16        return ResponseEntity.ok(roomService.getRoomById(id));
17    }
18
19    @PostMapping no usages
20    public ResponseEntity<Room> createRoom(@Valid @RequestBody CreateRoomDTO dto) {
21        return ResponseEntity.ok(roomService.createRoom(dto));
22    }
23
24    @PutMapping("/{id}") no usages
25    public ResponseEntity<Room> updateRoom(@PathVariable int id, @Valid @RequestBody UpdateRoomDTO dto) {
26        return ResponseEntity.ok(roomService.updateRoom(id, dto));
27    }
28
29    @DeleteMapping("/{id}") no usages
30    public ResponseEntity<Void> deleteRoom(@PathVariable int id) {
31        roomService.deleteRoom(id);
32        return ResponseEntity.noContent().build();
33    }
34}
```

4- Services

```
19  @Service 1 usage
20  public class RoomService {
21
22      @Autowired 7 usages
23      private RoomRepository roomRepository;
24
25      @Autowired 5 usages
26      private RedisClient redisClient;
27
28      @Autowired 2 usages
29      private ObjectMapper objectMapper;
30
31      private static final String ALL_ROOMS_CACHE_KEY = "rooms:all"; 5 usages
32      private static final Duration CACHE_TTL = Duration.ofMinutes(10); 1 usage
33      private static final String ROOM_LOCK_PREFIX = "room"; 1 usage
34
35      public List<Room> getAllRooms() { 1 usage
36          try {
37              String cachedRooms = redisClient.get(ALL_ROOMS_CACHE_KEY);
38              if (cachedRooms != null) {
39                  return objectMapper.readValue(cachedRooms, new TypeReference<List<Room>>() {});
40              }
41          } catch (Exception e) {
42              // fallback to DB
43          }
44
45          List<Room> rooms = roomRepository.findAll();
46          try {
47              String roomsJson = objectMapper.writeValueAsString(rooms);
48              redisClient.set(ALL_ROOMS_CACHE_KEY, roomsJson, CACHE_TTL);
49          } catch (Exception e) {
```

```

20     public class RoomService {
25         public List<Room> getAllRooms() { 1 usage
26             String roomsJson = roomRepository.findAll().stream()
27                 .map(room -> room.toJson())
28                 .collect(Collectors.joining(","));
29
30             redisClient.set(ALL_ROOMS_CACHE_KEY, roomsJson, CACHE_TTL);
31         } catch (Exception e) {
32             // cache store failed, no problem
33         }
34
35         return rooms;
36     }
37
38     public Room getRoomById(int id) { 1 usage
39         return roomRepository.findById(id).orElseThrow(() -> new RuntimeException("Room not found"));
40     }
41
42     @
43     public Room createRoom(CreateRoomDTO dto) { 1 usage
44         Room room = new Room();
45         room.setNumber(dto.getNumber());
46         room.setCapacity(dto.getCapacity());
47         room.setPricePerNight(dto.getPricePerNight());
48         room.setAvailable(dto.getIsAvailable());
49
50         Room saved = roomRepository.save(room);
51
52         // ✎ Invalidate cache after create
53         redisClient.delete(ALL_ROOMS_CACHE_KEY);
54
55         return saved;
56     }
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74

```

```

20     public class RoomService {
21
22         @DistributedLock(keyPrefix = ROOM_LOCK_PREFIX, keyIdentifierExpression = "#id", leaseTime = 120, timeUnit = TimeUnit.SECONDS)
23         public Room updateRoom(int id, UpdateRoomDTO dto) {
24             Room existingRoom = roomRepository.findById(id).orElseThrow(() -> new RuntimeException("Room not found"));
25
26             if (dto.getNumber() != null) existingRoom.setNumber(dto.getNumber());
27             if (dto.getCapacity() != null) existingRoom.setCapacity(dto.getCapacity());
28             if (dto.getPricePerNight() != null) existingRoom.setPricePerNight(dto.getPricePerNight());
29             if (dto.getIsAvailable() != null) existingRoom.setAvailable(dto.getIsAvailable());
30
31             Room updated = roomRepository.save(existingRoom);
32
33             // ✎ Invalidate cache after update
34             redisClient.delete(ALL_ROOMS_CACHE_KEY);
35
36             return updated;
37         }
38
39         public void deleteRoom(int id) { 1 usage
40             Room room = roomRepository.findById(id).orElseThrow(() -> new RuntimeException("Room not found"));
41             roomRepository.delete(room);
42
43             // ✎ Invalidate cache after delete
44             redisClient.delete(ALL_ROOMS_CACHE_KEY);
45         }
46
47
48
49
50
51
52
53
54
55
56
57
58
59
59
60
61
62
63
64
65
66
67
68
69
69
70
71
72
73
74

```

Note: caching is used in the getAllRooms endpoint, and a distributed lock is used with updateRoom endpoint.

6- Rate limiting annotation, aspect, and exception:

```
public class RateLimitingAspect {
    public Object rateLimit(ProceedingJoinPoint joinPoint, RateLimit rateLimit) throws Throwable {
        ...
        if (currentCount == null) {
            // Should ideally not happen with increment, but handle defensively
            log.error("Failed to increment rate limit counter for key: {}", rateLimitKey);
            // Decide whether to proceed or block based on policy. Let's proceed for now.
            return joinPoint.proceed();
        }

        // 2. If it's the first request in this window, set the expiration
        if (currentCount == 1) {
            redisClient.expire(rateLimitKey, windowDuration);
            log.debug("Set expiration for rate limit key '{}' to {}", rateLimitKey, windowDuration);
        }

        // 3. Check if the limit is exceeded
        if (currentCount > limit) {
            log.warn("Rate limit exceeded for key '{}'. Count: {}, Limit: {}", rateLimitKey, currentCount, limit)
            throw new RateLimitExceedededException(
                "Rate limit exceeded. Limit: " + limit + " requests per " +
                rateLimit.duration() + " " + rateLimit.timeUnit().name().toLowerCase() + "."
            );
        }

        log.debug("Rate limit check passed for key '{}'. Count: {}, Limit: {}", rateLimitKey, currentCount, limit)

        // 4. Proceed with the original method execution if limit is not exceeded
        return joinPoint.proceed();
    }
}
```

```
@Aspect 1 usage
@Component
public class RateLimitingAspect {

    private static final Logger log = LoggerFactory.getLogger(RateLimitingAspect.class); 5 usages

    @Autowired 2 usages
    private RedisClient redisClient;

    @Around("@annotation(rateLimit)") // Advice runs around methods annotated with @RateLimit  no usages
    public Object rateLimit(ProceedingJoinPoint joinPoint, RateLimit rateLimit) throws Throwable {

        HttpServletRequest request = getCurrentHttpRequest();
        if (request == null) {
            log.warn("Rate limiting skipped: Could not obtain HttpServletRequest.");
            return joinPoint.proceed(); // Proceed without limiting if request context is unavailable
        }

        String clientIp = getClientIp(request);
        String methodKey = getMethodKey(joinPoint);
        String rateLimitKey = buildRateLimitKey(rateLimit.keyPrefix(), methodKey, clientIp);

        long limit = rateLimit.limit();
        Duration windowDuration = Duration.of(rateLimit.duration(), rateLimit.timeUnit().toChronoUnit());

        // 1. Increment the counter for the key
        Long currentCount = redisClient.increment(rateLimitKey);
```

```
public class RateLimitingAspect {
    public Object rateLimit(ProceedingJoinPoint joinPoint, RateLimit rateLimit) throws Throwable {
    }

    private HttpServletRequest getCurrentHttpRequest() { 1 usage
        ServletRequestAttributes attributes = (ServletRequestAttributes) RequestContextHolder.getRequestAttributes();
        return (attributes != null) ? attributes.getRequest() : null;
    }

    private String getClientIp(HttpServletRequest request) { 1 usage
        String xForwardedForHeader = request.getHeader(name: "X-Forwarded-For");
        if (xForwardedForHeader == null || xForwardedForHeader.isEmpty()) {
            return request.getRemoteAddr();
        }
        // X-Forwarded-For can contain a comma-separated list (client, proxy1, proxy2)
        return xForwardedForHeader.split(regex: "[, ]")[0].trim();
    }

    private String getMethodKey(ProceedingJoinPoint joinPoint) { 1 usage
        MethodSignature signature = (MethodSignature) joinPoint.getSignature();
        Method method = signature.getMethod();
        return method.getDeclaringClass().getName() + "#" + method.getName();
    }

    private String buildRateLimitKey(String prefix, String methodKey, String clientIp) { 1 usage
        if (prefix != null && !prefix.trim().isEmpty()) {
            return "rate_limit:" + prefix.trim() + ":" + clientIp;
        } else {
            return "rate_limit:" + methodKey + ":" + clientIp;
        }
    }
}
```

```
3
4     @Target(ElementType.METHOD) // Apply to methods 4 usages
5     @Retention(RetentionPolicy.RUNTIME) // Keep annotation info at
6     public @interface RateLimit {
7         long limit(); // * The maximum number of requests allowed w
8         long duration(); // The duration of the time window. 3 usages
9         TimeUnit timeUnit() default TimeUnit.SECONDS; // The time u
10        String keyPrefix() default ""; 2 usages
11    }
```

```
1 @ResponseStatus(value = HttpStatus.TOO_MANY_REQUESTS) 2 usages
2 public class RateLimitExceededException extends RuntimeException {
3     public RateLimitExceededException(String message) { 1 usage
4         super(message);
5     }
6 }
```

7- Locking annotation, aspect, and exception:

```
3     @Target(ElementType.METHOD) 4 usages
4     @Retention(RetentionPolicy.RUNTIME)
5     public @interface DistributedLock {
6
7         /**
8          * The base key prefix for the lock in Redis.
9          * A unique identifier (like an entity ID) will usually be appended.
10         */
11        String keyPrefix(); 2 usages
12
13        /**
14         * SpEL expression to extract the unique identifier part of the lock key
15         * from the method arguments. Example: "#id" or "#roomDTO.roomId".
16         * The argument names must match the parameter names in the method signature.
17         */
18        String keyIdentifierExpression(); 3 usages
19
20        /**
21         * Lock lease duration. How long the lock is held before automatically expiring
22         * if the holder crashes. Prevents deadlocks.
23         */
24        long leaseTime() default 30; // Default 30 seconds 3 usages
25
26        /**
27         * Time unit for the lease time.
28         */
29        TimeUnit timeUnit() default TimeUnit.SECONDS; 3 usages
30    }
```

```
28     public class LockingAspect {
29         public Object applyLock(ProceedingJoinPoint joinPoint, DistributedLock distributedLock) throws Throwable {
30
31             String lockKey = String.format(LOCK_KEY_FORMAT, distributedLock.keyPrefix(), lockIdentifier);
32             String lockValue = UUID.randomUUID().toString(); // Unique value for this lock attempt
33             Duration leaseDuration = Duration.of(distributedLock.leaseTime(), distributedLock.timeUnit().toChronoUnit());
34
35             boolean lockAcquired = false;
36             try {
37                 log.debug("Attempting to acquire lock '{}' with value '{}' for {} {}", lockKey, lockValue, distributedLock.leaseTime(), distributedLock.timeUnit().name());
38
39                 // Try to acquire the lock using setIfAbsent with expiration
40                 lockAcquired = redisClient.setIfAbsent(lockKey, lockValue, leaseDuration);
41
42                 if (lockAcquired) {
43                     log.info("Lock acquired successfully: '{}'", lockKey);
44                     // Proceed with the original method execution
45                     return joinPoint.proceed();
46                 } else {
47                     log.warn("Failed to acquire lock '{}', it is held by another process.", lockKey);
48                     throw new LockAcquisitionException("Resource is currently locked by another operation. Please try again later.");
49                 }
50             } finally {
51                 // Release the lock ONLY if this specific attempt acquired it
52                 if (lockAcquired) {
53                     // Check if the lock value still matches ours before deleting
54                     // This prevents deleting a lock acquired by another process if our lease expired
55                     String currentValue = redisClient.get(lockKey);
56                     if (lockValue.equals(currentValue)) {
57                         try {
58                             Boolean deleted = redisClient.delete(lockKey);
59                         }
60                     }
61                 }
62             }
63         }
64     }
```

```
5
6     @Aspect 1 usage
7     @Component
8     public class LockingAspect {
9
10
11     private static final Logger log = LoggerFactory.getLogger(LockingAspect.class); 9 usages
12     private static final String LOCK_KEY_FORMAT = "lock:%s:%s"; // Format: lock:<prefix>:<identifier> 1 usage
13     private final ParameterNameDiscoverer parameterNameDiscoverer = new DefaultParameterNameDiscoverer(); 1 usage
14     private final ExpressionParser expressionParser = new SpelExpressionParser(); 1 usage
15
16
17     @Autowired 3 usages
18     private RedisClient redisClient;
19
20     @Around("@annotation(distributedLock)") no usages
21     @
22     public Object applyLock(ProceedingJoinPoint joinPoint, DistributedLock distributedLock) throws Throwable {
23
24         MethodSignature signature = (MethodSignature) joinPoint.getSignature();
25         Method method = signature.getMethod();
26         Object[] args = joinPoint.getArgs();
27
28         String lockIdentifier = evaluateKeyIdentifier(distributedLock.keyIdentifierExpression(), method, args);
29         if (lockIdentifier == null || lockIdentifier.trim().isEmpty()) {
30             log.error("Failed to evaluate lock identifier expression '{}' for method {}", distributedLock.keyIdentifierExpression(), method.getName());
31             throw new LockAcquisitionException("Cannot acquire lock: Lock identifier could not be determined.");
32         }
33
34         String lockKey = String.format(LOCK_KEY_FORMAT, distributedLock.keyPrefix(), lockIdentifier);
35         String lockValue = UUID.randomUUID().toString(); // Unique value for this lock attempt
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
```

```
55
56     public class LockingAspect {
57
58         public Object applyLock(ProceedingJoinPoint joinPoint, DistributedLock distributedLock) throws Throwable {
59
60             try {
61                 String lockKey = String.format(LOCK_KEY_FORMAT, distributedLock.keyPrefix(), lockIdentifier);
62                 String lockValue = UUID.randomUUID().toString(); // Unique value for this lock attempt
63
64                 log.info("Attempting to acquire lock '{}'.", lockKey);
65
66                 Object result = joinPoint.proceed();
67
68                 log.info("Lock acquired successfully for key '{}'.", lockKey);
69
70                 return result;
71             } catch (Exception e) {
72                 log.error("Failed to acquire lock '{}'. Error: {}", lockKey, e);
73
74                 if (distributedLock.isExclusive()) {
75                     log.warn("Attempted to release lock '{}', but it was not found (possibly expired).", lockKey);
76                 } else {
77                     log.warn("Did not release lock '{}' because the lock value changed (lease likely expired and lockKey, currentValue, lockValue);", lockKey);
78                 }
79             }
80         }
81
82         /**
83          * Evaluates the SpEL expression against the method arguments to get the lock identifier.
84          */
85         @
86         private String evaluateKeyIdentifier(String expressionString, Method method, Object[] args) { 1 usage
87             try {
88                 EvaluationContext context = new MethodBasedEvaluationContext( rootObject: null, method, args, parameterNameDiscoverer );
89                 Object value = expressionParser.parseExpression(expressionString).getValue(context);
90                 return (value != null) ? value.toString() : null;
91             } catch (Exception e) {
92                 log.error("Error evaluating SpEL expression '{}' for method {}: {}", expressionString, method.getName(), e);
93                 return null;
94             }
95         }
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
```

```

1 package com.example.lab4.Exceptions;
2
3 import org.springframework.http.HttpStatus;
4 import org.springframework.web.bind.annotation.ResponseStatus;
5
6 @ResponseStatus(value = HttpStatus.LOCKED) 3 usages
7 public class LockAcquisitionException extends RuntimeException {
8     public LockAcquisitionException(String message) { 2 usages
9         super(message);
10    }
11
12    public LockAcquisitionException(String message, Throwable cause) { no usages
13        super(message, cause);
14    }
15}

```

8- testing with Postman

- Testing the updateRoom endpoint and trying to update the same room with two requests at the same time:

The screenshot shows two parallel POST requests in Postman. Both requests are directed to `http://localhost:8080/api/rooms/4`. The left request's body is:

```

1 {
2     "isAvailable": false
3 }

```

The right request's body is identical. Both requests return a `423 Locked` response. The left request's response body is:

```

1 {
2     "timestamp": "2025-04-26T17:59:42.039+00:00",
3     "status": 423,
4     "error": "Locked",
5     "path": "/api/rooms/4"
6 }

```

The right request's response body is:

```

1 {
2     "id": 4,
3     "number": "1040",
4     "capacity": 4,
5     "pricePerNight": 120.0,
6     "available": false
7 }

```

- Calling getAllRooms endpoint several times in a row:

The image consists of three vertically stacked screenshots of a Postman API client interface, demonstrating a sequence of requests to the `/api/rooms` endpoint.

Screenshot 1: The first request is sent to `http://localhost:8080/api/rooms`. The response status is `200 OK`, time taken is `212 ms`, and size is `932 B`. The response body is a JSON array containing one room object:

```

1 [
2   {
3     "id": 1,
4     "number": "101A",
5     "capacity": 2,
6     "pricePerNight": 95.00,
7     "available": true
8   }
9 ]

```

Screenshot 2: The second request is sent to the same endpoint. The response status is `200 OK`, time taken is `20 ms`, and size is `932 B`. The response body is identical to the first:

```

1 [
2   {
3     "id": 1,
4     "number": "101A",
5     "capacity": 2,
6     "pricePerNight": 95.00,
7     "available": true
8   }
9 ]

```

Screenshot 3: The third request is sent to the same endpoint. The response status is `429 Too Many Requests`, time taken is `11 ms`, and size is `285 B`. The response body is a JSON object indicating a rate limit error:

```

1 {
2   "timestamp": "2025-04-20T18:44:16.363+00:00",
3   "status": 429,
4   "error": "Too Many Requests",
5   "path": "/api/rooms"
6 }

```

Note: we can see in the first call we got the response in 212ms, and 20 ms in the second due to getting a cache hit, and in the third request we got “429 too many requests” because of the ratelimiter.

Github Repo link:

<https://github.com/MohamedMaghrabyyy/Aspect-and-Service-Oriented-Programming-Labs/tree/main/lab4>