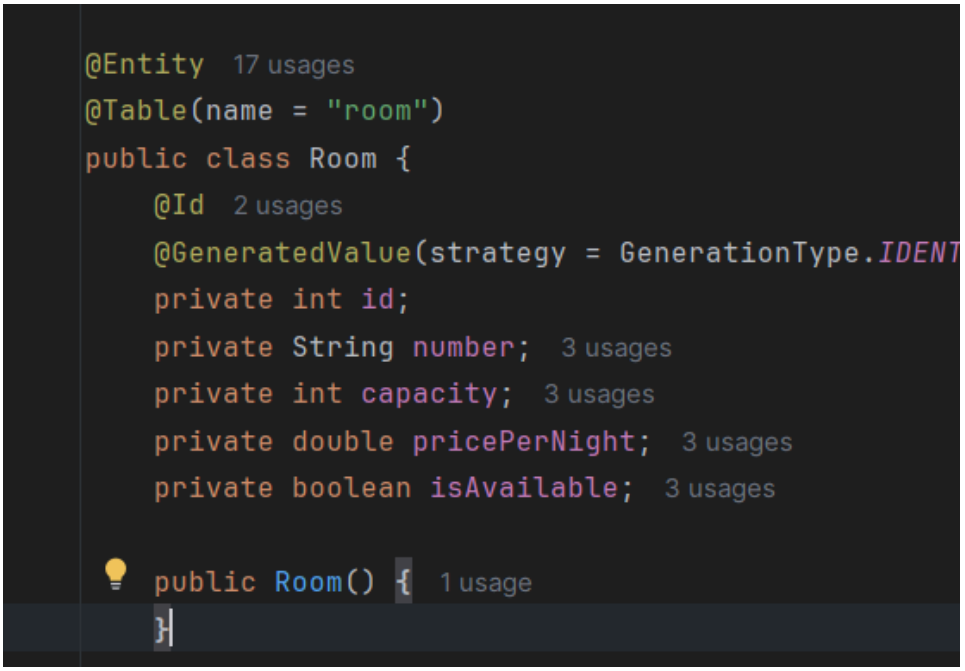
University of East London

CSE434 Aspect and Service-Oriented Software Systems

Lab 4 Assignment

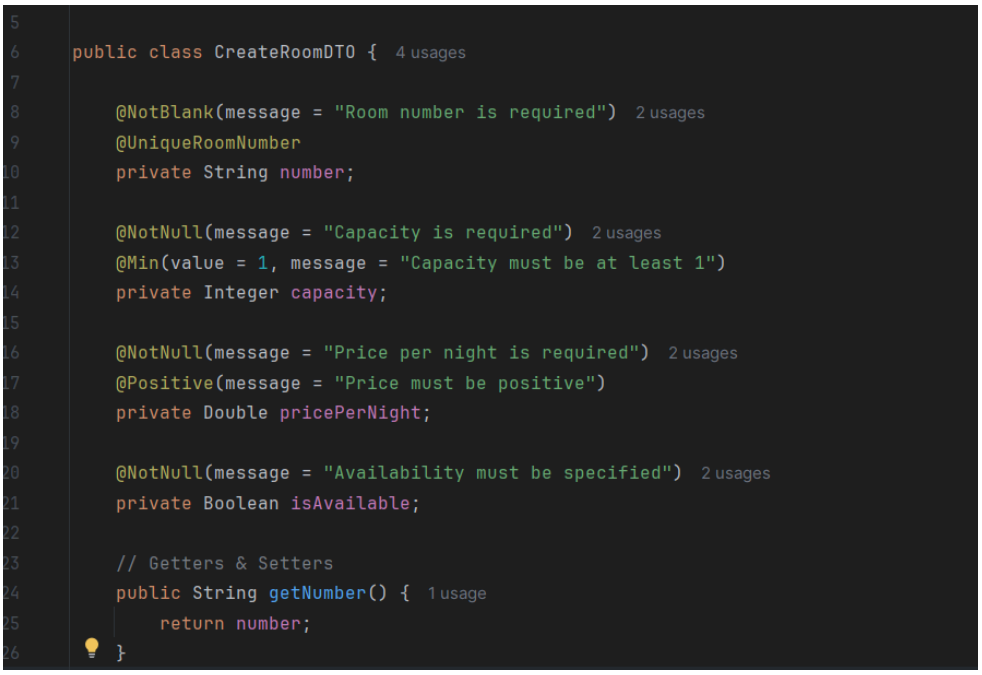
Name	Mohamed Mostafa Bedair ElMaghraby
ID	20P7732

1- Models



```java
@Entity  17 usages
@Table(name = "room")
public class Room {
    @Id  2 usages
    @GeneratedValue(strategy = GenerationType.IDENT
    private int id;
    private String number;  3 usages
    private int capacity;  3 usages
    private double pricePerNight;  3 usages
    private boolean isAvailable;  3 usages

    public Room() {  1 usage
    }
```

2- DTOs



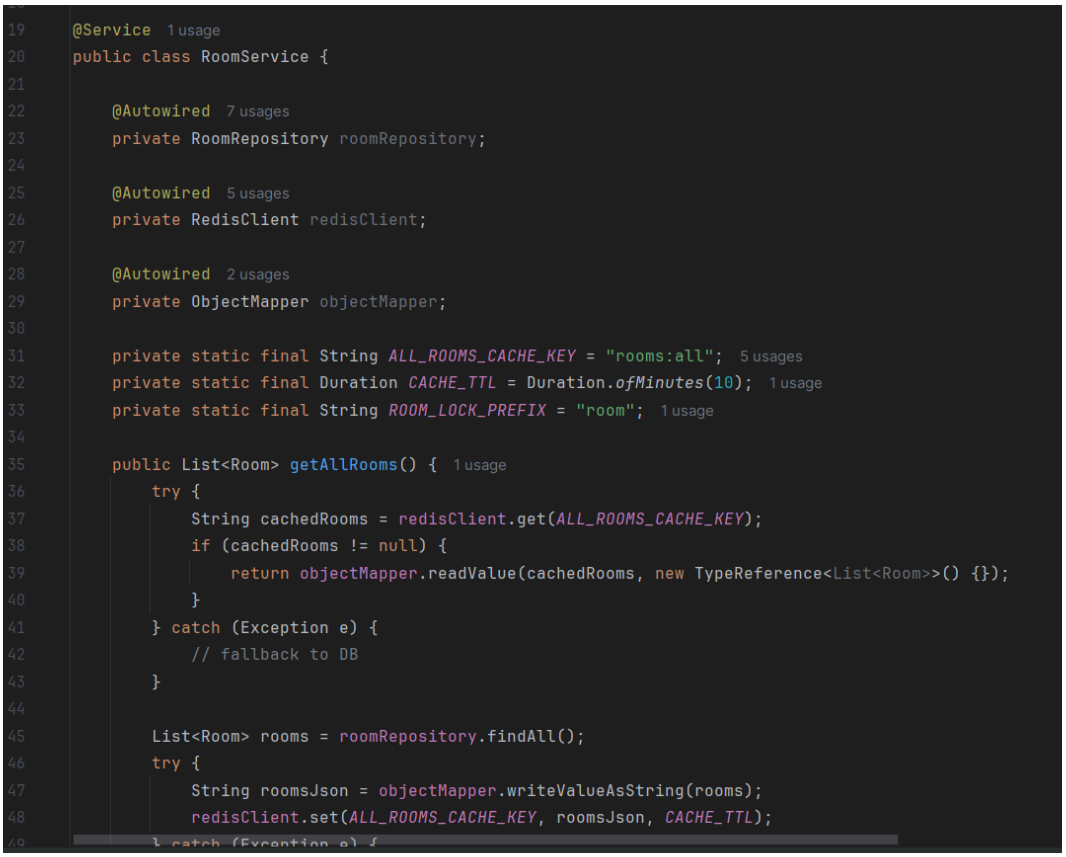
```java
public class CreateRoomDTO {  4 usages

    @NotBlank(message = "Room number is required")  2 usages
    @UniqueRoomNumber
    private String number;

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

    @NotNull(message = "Price per night is required")  2 usages
    @Positive(message = "Price must be positive")
    private Double pricePerNight;

    @NotNull(message = "Availability must be specified")  2 usages
    private Boolean isAvailable;

    // Getters & Setters
    public String getNumber() {  1 usage
        return number;
    }
```

```java
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;

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

    private Boolean isAvailable;  2 usages
```

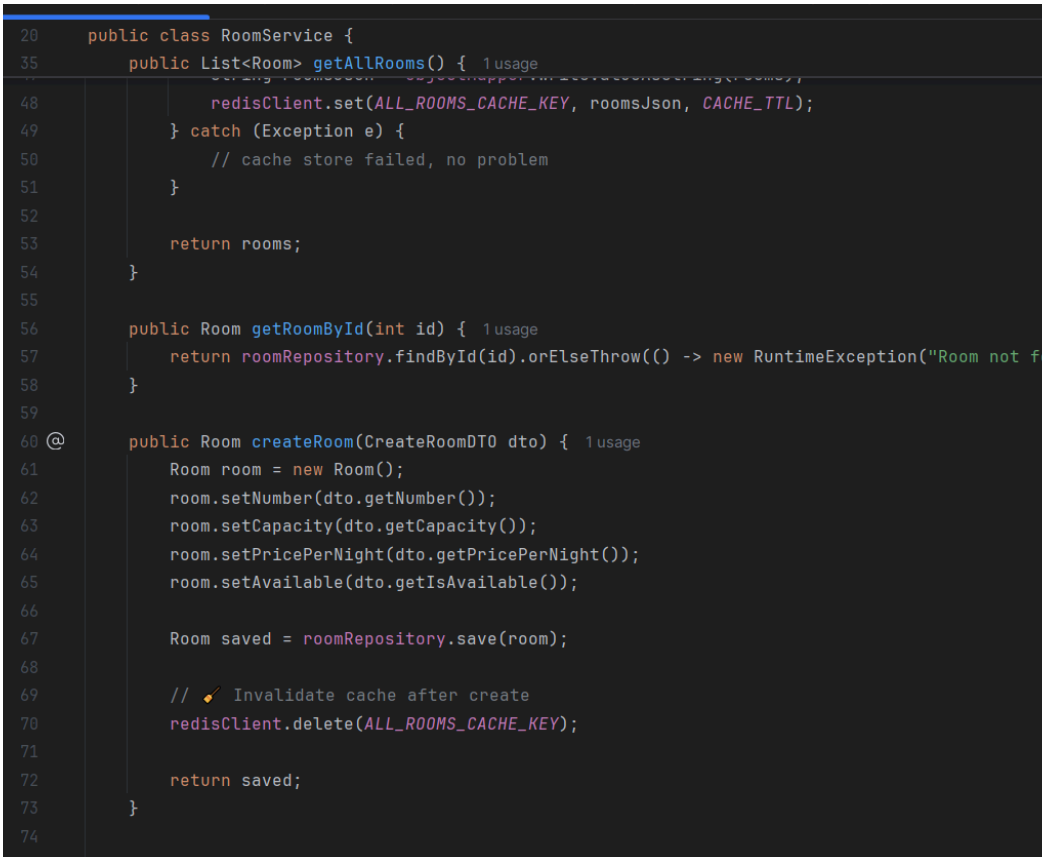
3- Controllers

```java
@RequestMapping("/api/rooms")
public class RoomController {

    @Autowired  5 usages
    private RoomService roomService;

    @GetMapping  no usages
    @RateLimit(limit = 5, duration = 60, keyPrefix = "getAllRooms")
    public ResponseEntity<List<Room>> getAllRooms() { return ResponseEntity.ok(roomService.getAllRooms()); }

    @GetMapping("/{id}")  no usages
    public ResponseEntity<Room> getRoomById(@PathVariable int id) {
        return ResponseEntity.ok(roomService.getRoomById(id));
    }

    @PostMapping  no usages
    public ResponseEntity<Room> createRoom(@Valid @RequestBody CreateRoomDTO dto) {
        return ResponseEntity.ok(roomService.createRoom(dto));
    }

    @PutMapping("/{id}")  no usages
    public ResponseEntity<Room> updateRoom(@PathVariable int id, @Valid @RequestBody UpdateRoomDTO dto) {
        return ResponseEntity.ok(roomService.updateRoom(id, dto));
    }

    @DeleteMapping("/{id}")  no usages
    public ResponseEntity<Void> deleteRoom(@PathVariable int id) {
        roomService.deleteRoom(id);
        return ResponseEntity.noContent().build();
    }
```

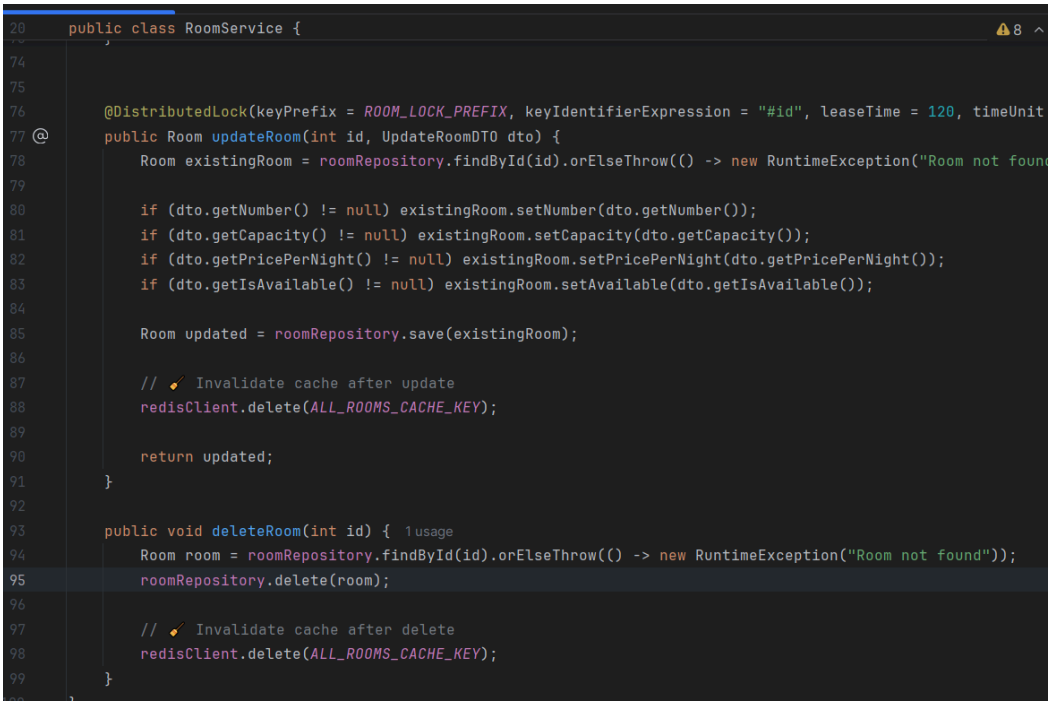
4- Services



```java
@Service  1 usage
public class RoomService {

    @Autowired  7 usages
    private RoomRepository roomRepository;

    @Autowired  5 usages
    private RedisClient redisClient;

    @Autowired  2 usages
    private ObjectMapper objectMapper;

    private static final String ALL_ROOMS_CACHE_KEY = "rooms:all";  5 usages
    private static final Duration CACHE_TTL = Duration.ofMinutes(10);  1 usage
    private static final String ROOM_LOCK_PREFIX = "room";  1 usage

    public List<Room> getAllRooms() {  1 usage
        try {
            String cachedRooms = redisClient.get(ALL_ROOMS_CACHE_KEY);
            if (cachedRooms != null) {
                return objectMapper.readValue(cachedRooms, new TypeReference<List<Room>>() {});
            }
        } catch (Exception e) {
            // fallback to DB
        }

        List<Room> rooms = roomRepository.findAll();
        try {
            String roomsJson = objectMapper.writeValueAsString(rooms);
            redisClient.set(ALL_ROOMS_CACHE_KEY, roomsJson, CACHE_TTL);
        } catch (Exception e) {
```



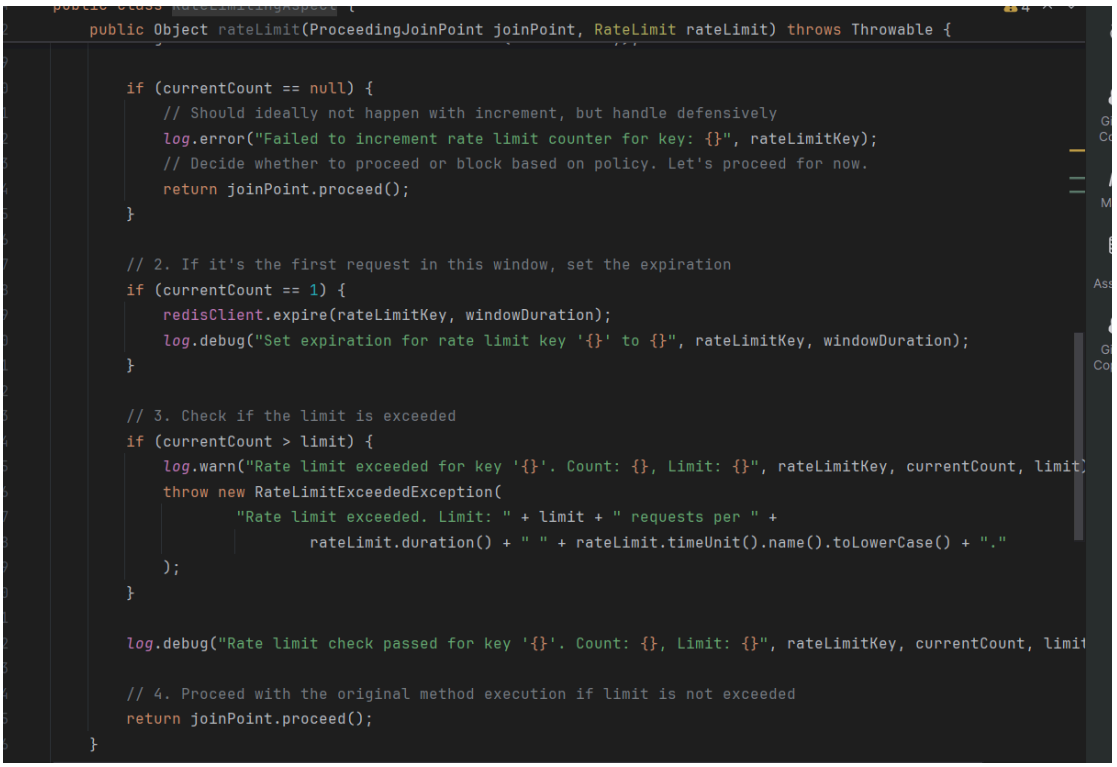
```
public class RoomService {
    public List<Room> getAllRooms() {  1 usage
            redisClient.set(ALL_ROOMS_CACHE_KEY, roomsJson, CACHE_TTL);
        } catch (Exception e) {
            // cache store failed, no problem
        }

        return rooms;
    }

    public Room getRoomById(int id) {  1 usage
        return roomRepository.findById(id).orElseThrow(() -> new RuntimeException("Room not f
    }

    public Room createRoom(CreateRoomDTO dto) {  1 usage
        Room room = new Room();
        room.setNumber(dto.getNumber());
        room.setCapacity(dto.getCapacity());
        room.setPricePerNight(dto.getPricePerNight());
        room.setAvailable(dto.getIsAvailable());

        Room saved = roomRepository.save(room);

        // 🧹 Invalidate cache after create
        redisClient.delete(ALL_ROOMS_CACHE_KEY);

        return saved;
    }
```



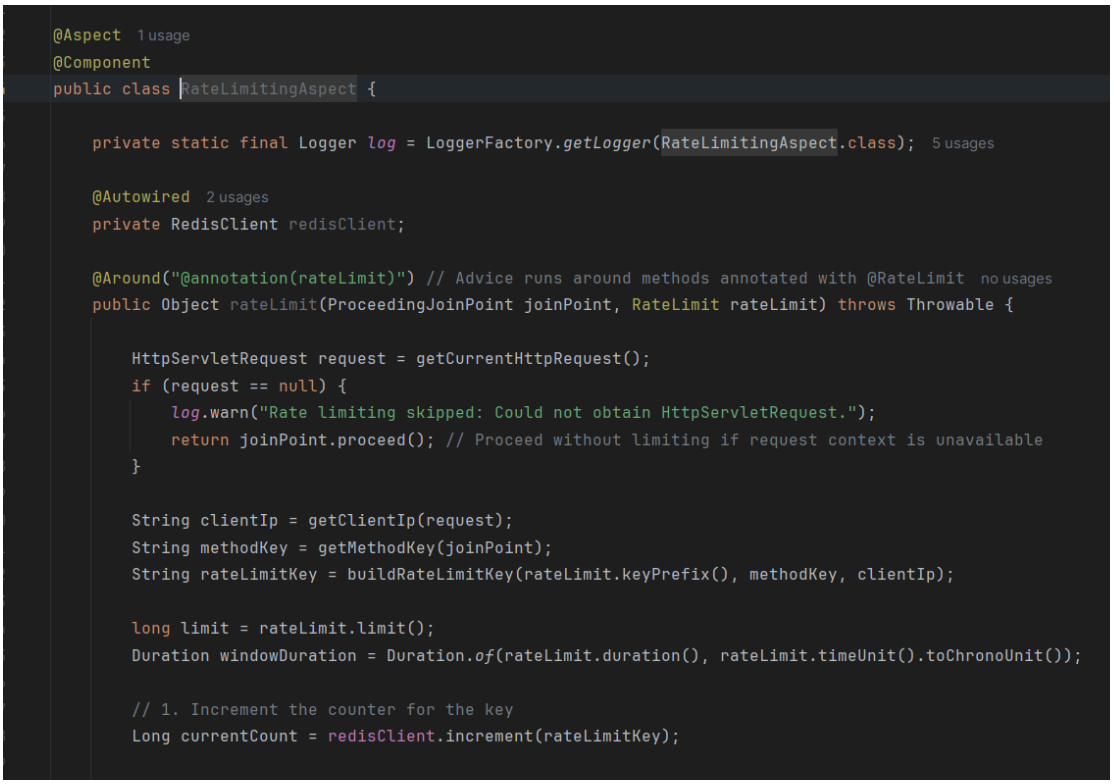
```
public class RoomService {  8


    @DistributedLock(keyPrefix = ROOM_LOCK_PREFIX, keyIdentifierExpression = "#id", leaseTime = 120, timeUnit
    public Room updateRoom(int id, UpdateRoomDTO dto) {
        Room existingRoom = roomRepository.findById(id).orElseThrow(() -> new RuntimeException("Room not found

        if (dto.getNumber() != null) existingRoom.setNumber(dto.getNumber());
        if (dto.getCapacity() != null) existingRoom.setCapacity(dto.getCapacity());
        if (dto.getPricePerNight() != null) existingRoom.setPricePerNight(dto.getPricePerNight());
        if (dto.getIsAvailable() != null) existingRoom.setAvailable(dto.getIsAvailable());

        Room updated = roomRepository.save(existingRoom);

        // 🧹 Invalidate cache after update
        redisClient.delete(ALL_ROOMS_CACHE_KEY);

        return updated;
    }

    public void deleteRoom(int id) {  1 usage
        Room room = roomRepository.findById(id).orElseThrow(() -> new RuntimeException("Room not found"));
        roomRepository.delete(room);

        // 🧹 Invalidate cache after delete
        redisClient.delete(ALL_ROOMS_CACHE_KEY);
    }
```

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 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 RateLimitExceededException(
                    "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);
```

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

    private HttpServletRequest getCurrentHttpRequest() {  1 usage
        ServletRequestAttributes attributes = (ServletRequestAttributes) RequestContextHolder.getReq
        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;
        }
    }
```

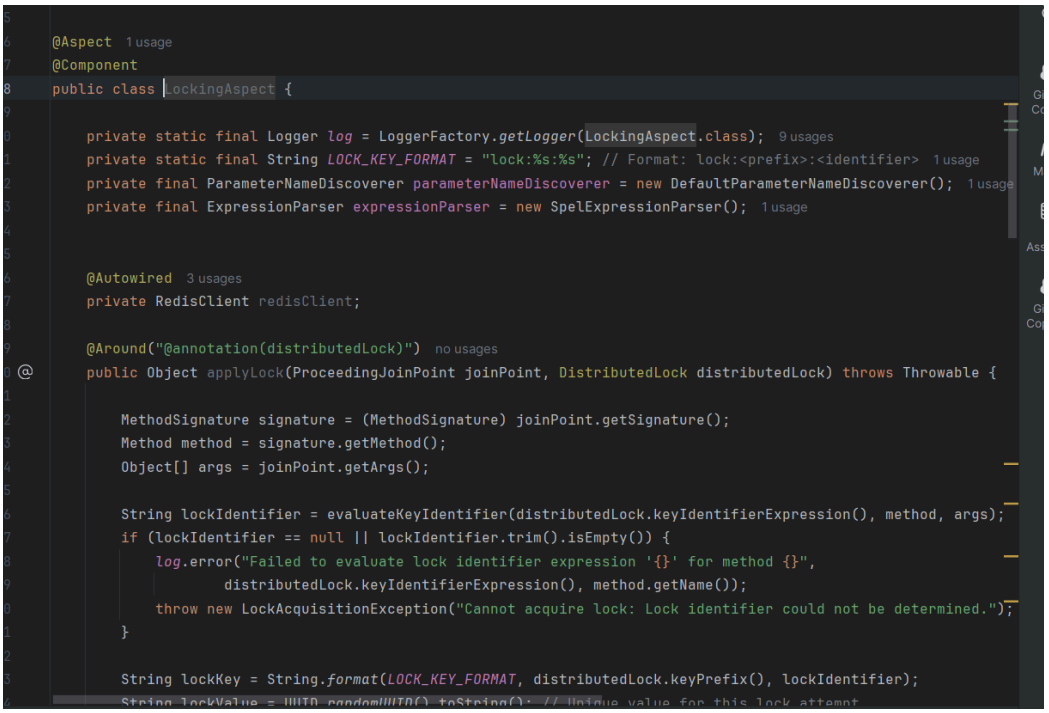
```java
@Target(ElementType.METHOD) // Apply to methods  4 usages
@Retention(RetentionPolicy.RUNTIME) // Keep annotation info at
public @interface RateLimit {
    long limit(); // * The maximum number of requests allowed w
    long duration(); // The duration of the time window.  3 usage
    TimeUnit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

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

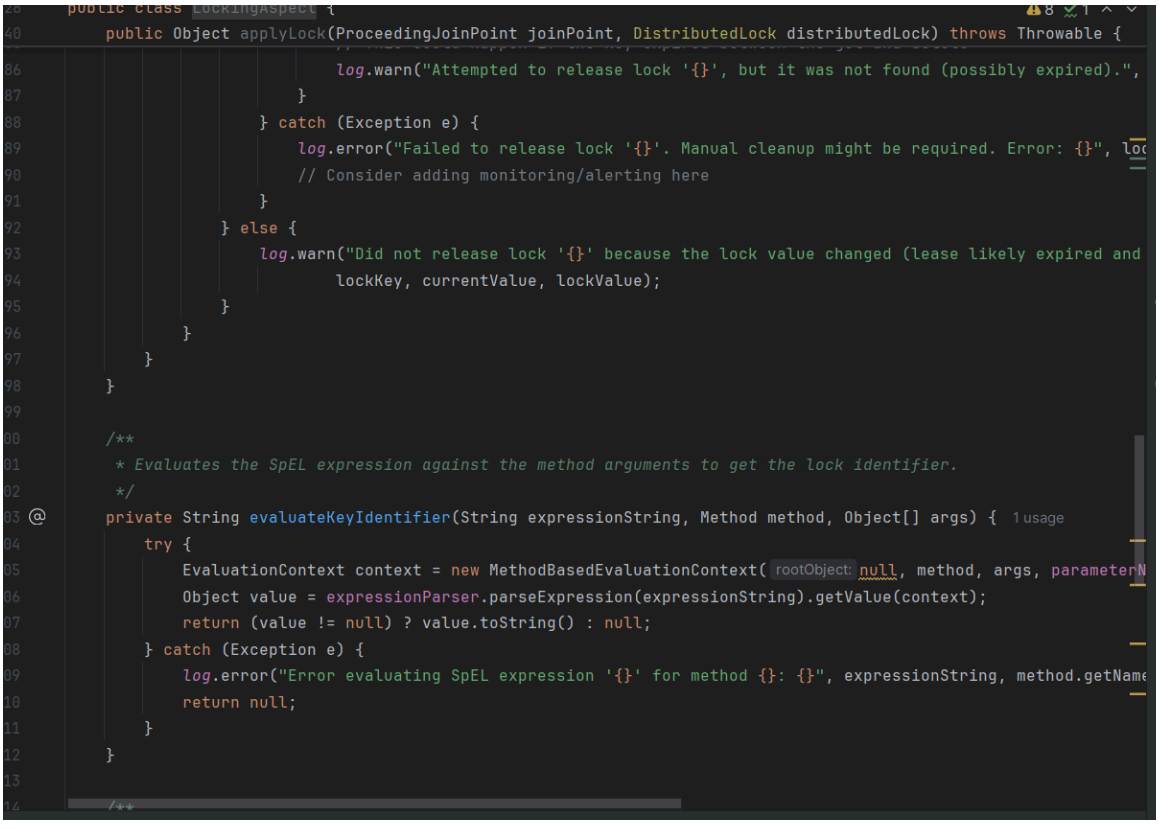
7- Locking annotation, aspect, and exception:

```java
@Target(ElementType.METHOD)  4 usages
@Retention(RetentionPolicy.RUNTIME)
public @interface DistributedLock {

    /**
     * The base key prefix for the lock in Redis.
     * A unique identifier (like an entity ID) will usually be appended.
     */
    String keyPrefix();  2 usages

    /**
     * SpEL expression to extract the unique identifier part of the lock key
     * from the method arguments. Example: "#id" or "#roomDTO.roomId".
     * The argument names must match the parameter names in the method signature.
     */
    String keyIdentifierExpression();  3 usages

    /**
     * Lock lease duration. How long the lock is held before automatically expiring
     * if the holder crashes. Prevents deadlocks.
     */
    long leaseTime() default 30; // Default 30 seconds  3 usages

    /**
     * Time unit for the lease time.
     */
    TimeUnit timeUnit() default TimeUnit.SECONDS;  3 usages
}
```

```java
public class LockingAspect {
    public Object applyLock(ProceedingJoinPoint joinPoint, DistributedLock distributedLock) throws Throwable {
        String lockKey = String.format(LOCK_KEY_FORMAT, distributedLock.keyPrefix(), lockIdentifier);
        String lockValue = UUID.randomUUID().toString(); // Unique value for this lock attempt
        Duration leaseDuration = Duration.of(distributedLock.leaseTime(), distributedLock.timeUnit().toChronoUnit

        boolean lockAcquired = false;
        try {
            log.debug("Attempting to acquire lock '{}' with value '{}' for {} {}",
                    lockKey, lockValue, distributedLock.leaseTime(), distributedLock.timeUnit().name());

            // Try to acquire the lock using setIfAbsent with expiration
            lockAcquired = redisClient.setIfAbsent(lockKey, lockValue, leaseDuration);

            if (lockAcquired) {
                log.info("Lock acquired successfully: '{}'", lockKey);
                // Proceed with the original method execution
                return joinPoint.proceed();
            } else {
                log.warn("Failed to acquire lock '{}', it is held by another process.", lockKey);
                throw new LockAcquisitionException("Resource is currently locked by another operation. Please try
            }
        } finally {
            // Release the lock ONLY if this specific attempt acquired it
            if (lockAcquired) {
                // Check if the lock value still matches ours before deleting
                // This prevents deleting a lock acquired by another process if our lease expired
                String currentValue = redisClient.get(lockKey);
                if (lockValue.equals(currentValue)) {
                    try {
                        Boolean deleted = redisClient.delete(lockKey);
```



```java
@Aspect  1 usage
@Component
public class LockingAspect {

    private static final Logger log = LoggerFactory.getLogger(LockingAspect.class);  9 usages
    private static final String LOCK_KEY_FORMAT = "lock:%s:%s"; // Format: lock:<prefix>:<identifier>  1 usage
    private final ParameterNameDiscoverer parameterNameDiscoverer = new DefaultParameterNameDiscoverer();  1 usage
    private final ExpressionParser expressionParser = new SpelExpressionParser();  1 usage


    @Autowired  3 usages
    private RedisClient redisClient;

    @Around("@annotation(distributedLock)")  no usages
    public Object applyLock(ProceedingJoinPoint joinPoint, DistributedLock distributedLock) throws Throwable {

        MethodSignature signature = (MethodSignature) joinPoint.getSignature();
        Method method = signature.getMethod();
        Object[] args = joinPoint.getArgs();

        String lockIdentifier = evaluateKeyIdentifier(distributedLock.keyIdentifierExpression(), method, args);
        if (lockIdentifier == null || lockIdentifier.trim().isEmpty()) {
            log.error("Failed to evaluate lock identifier expression '{}' for method {}",
                    distributedLock.keyIdentifierExpression(), method.getName());
            throw new LockAcquisitionException("Cannot acquire lock: Lock identifier could not be determined.");
        }

        String lockKey = String.format(LOCK_KEY_FORMAT, distributedLock.keyPrefix(), lockIdentifier);
        String lockValue = UUID.randomUUID().toString(); // Unique value for this lock attempt
```

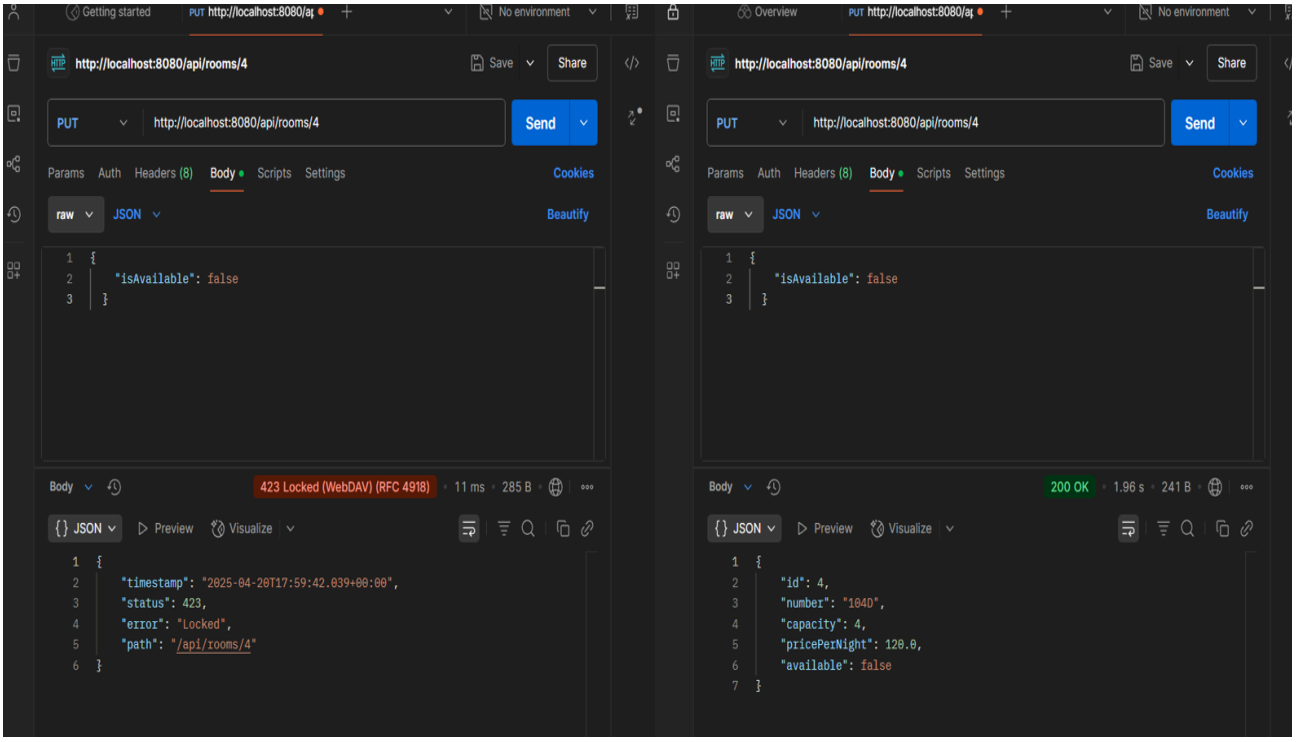


```java
public class LockingAspect {
    public Object applyLock(ProceedingJoinPoint joinPoint, DistributedLock distributedLock) throws Throwable {
                            log.warn("Attempted to release lock '{}', but it was not found (possibly expired).",
                        }
                    } catch (Exception e) {
                        log.error("Failed to release lock '{}'. Manual cleanup might be required. Error: {}", loc
                        // Consider adding monitoring/alerting here
                    }
                } else {
                    log.warn("Did not release lock '{}' because the lock value changed (lease likely expired and
                            lockKey, currentValue, lockValue);
                }
            }
        }
    }

    /**
     * Evaluates the SpEL expression against the method arguments to get the lock identifier.
     */
    private String evaluateKeyIdentifier(String expressionString, Method method, Object[] args) {  1 usage
        try {
            EvaluationContext context = new MethodBasedEvaluationContext( rootObject: null, method, args, parameterN
            Object value = expressionParser.parseExpression(expressionString).getValue(context);
            return (value != null) ? value.toString() : null;
        } catch (Exception e) {
            log.error("Error evaluating SpEL expression '{}' for method {}: {}", expressionString, method.getName
            return null;
        }
    }

    /**
```

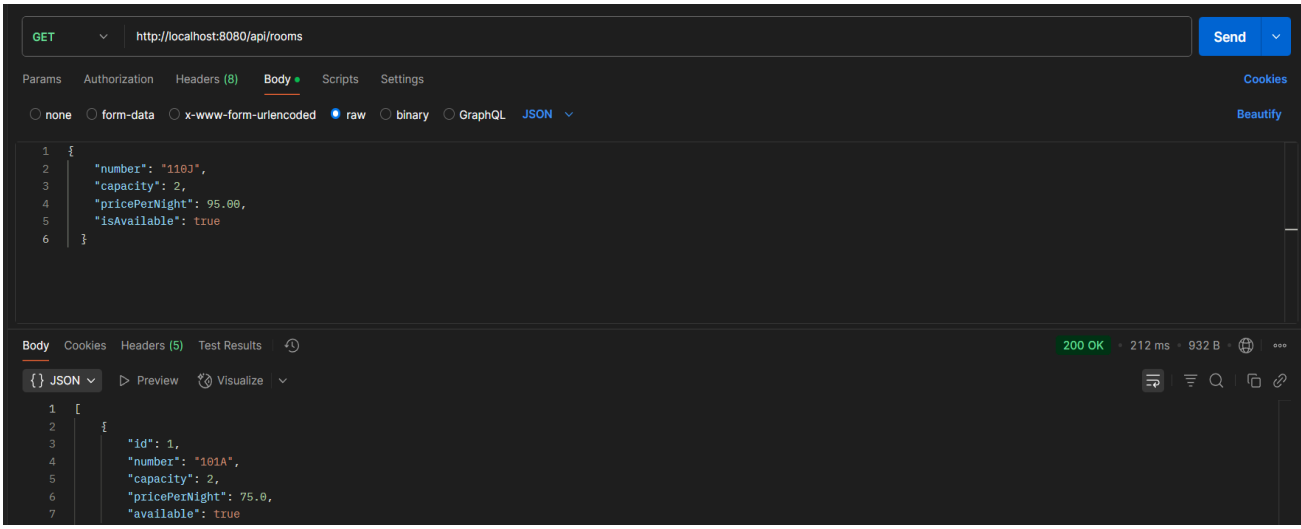
```java
package com.example.lab4.Exceptions;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ResponseStatus;

@ResponseStatus(value = HttpStatus.LOCKED)  3 usages
public class LockAcquisitionException extends RuntimeException {
    public LockAcquisitionException(String message) {  2 usages
        super(message);
    }

    public LockAcquisitionException(String message, Throwable cause) {  no usages
        super(message, cause);
    }
}
```

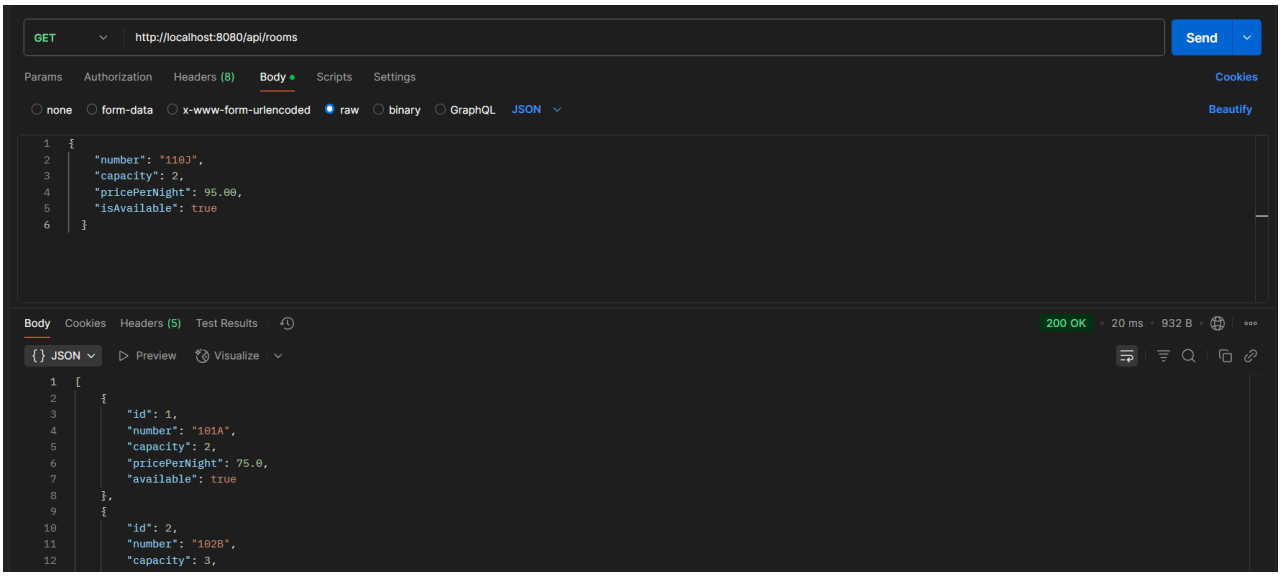
8- testing with Postman

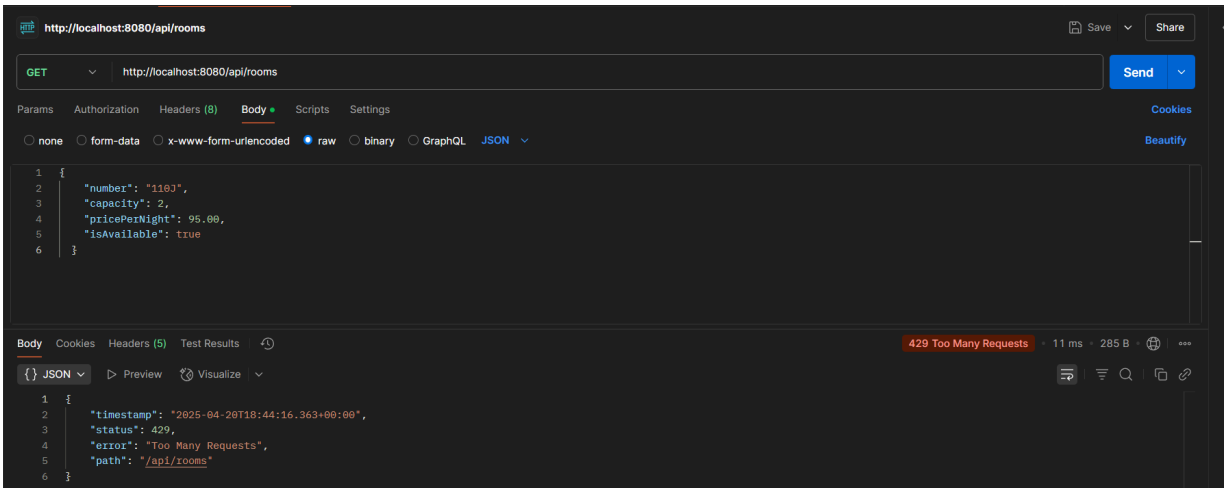
- Testing the updateRoom endpoint and trying to update the same room with two requests at the same time:



- Calling getAllRooms endpoint several times in a row:







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