Performance Optimization Report

Table of Contents

Identification of Bottlenecks and Optimization of Codes

1. Identified bottlenecks
   a. Inefficient Algorithms
      Unnecessary introduction of a new ArrayList and usage of for loop to append bed DTOs

```
public synchronized List<BedDto> getAllBeds() throws Exception {
    List<Bed> beds = bedRepository.findAll();
    List<BedDto> bedDtos = new ArrayList<>();
    for (Bed bed : beds) {
        bedDtos.add(convertToDto(bed));
    }
    System.out.println(GcStatsUtil.getGcStats());
    return bedDtos;
}
```

   b. Excessive Object Creation and Garbage Collection
      The introduction of ArrayList, when there is already a List created leads to the duplicate list creation and also duplicate objects for each list member.
   c. Synchronization and Concurrency Issues
      Unnecessary introduction and wrong usage of synchronized in the various functions including addbed function and update bed function.
      The locking of the bed to be updated could have been done on the specific bed, after retrieving the specific bed rather than locking the entire function as subsequent updates may not be on that particular bed.
2. Performance Improvements
   a. Eliminated inefficient algorithms
   b. Eliminated unnecessary object creation and this reduced the tendency of heap being occupied hence reducing high rate of Garbage collection
   c. Introduce Caching to improve performance

Comparison between before and after optimization

Code Optimization Performance Summary

Reduction of Object Creation in addBed function and endpoint

Inefficient Code: The initial inefficient function created a new bed object before passing it to the convertToDto for it to be returned as a DTO to the controller. This can be seen in the code snippet below.

```
public synchronized BedDto addBed(BedDto bedDto) throws Exception {
    Bed newBed = bedRepository.save(convertToEntity(bedDto));
    return convertToDto(newBed);
}
```

Optmized Code: In the optimized code, the creation of a newBed object is eliminated. This is because it makes it redundant, meaning it can be returned directly as a DTO without having to first assign the saved bed to an object before returning it. Below is the optimized code. The optimized code can be seen in the code snippet below.

```
public synchronized BedDto addBed(BedDto bedDto) throws Exception {

    return convertToDto(bedRepository.save(convertToEntity(bedDto)));
}
```

Elimination of redundant object creation in the getBedbyId function

Inefficient code: The inefficient algorithm in the getBedById function fetches the list of all beds, that is populating a list object from java collections which contains multiple bed objects and then subsequently loop through the list to find the bed that matches the id of the bed being searched for. This leads to unnecessary object creation and higher time complexity. Below is the code snippet of the getBedById function.

```
public BedDto getBedById(Long id) throws Exception {
    List<BedDto> allBeds = getAllBeds();
    for (BedDto bedDto : allBeds) {
        if (bedDto.getId().equals(id)) {
            return bedDto;
        }
    }
    throw new RuntimeException("Bed not found");
}
```

Optimized Code: The Optimized Code capitalizes on the findbyId JPA repository function which is efficient in performance and has a better time complexity. This ensures that the code's performance is increased and is efficient in not creating unnecessary objects. Below is snippet of the code.

```
public BedDto getBedById(Long id) {
    return convertToDto(bedRepository.findById(id).orElseThrow());
}
```

Correcting wrong implementation of synchronization in updateBed function

Inefficient code: The inefficient code of the updateBed implemented synchronization wrongly. For a single threaded instance, this won't be an issue but in a multithreaded instance, this will affect performance. Assuming more than one thread, each thread looking to modify a specific bed detail. Since a thread when accessing the updateBed function locks the entire function, the subsequent threads which may be targeting different bed resource will have to wait for the first thread to finish its task which makes the code highly inefficient. Below is the code snippet

```
public synchronized BedDto updateBed(BedDto bedDto) throws Exception {
    BedDto editedBedDto = new BedDto();
    if (bedDto.getId() != null) {
        Optional<Bed> editBed = bedRepository.findById(bedDto.getId());
        if (editBed.isPresent()) {
            Bed bed = editBed.get();
            bed.setBedNumber(bedDto.getBedNumber());
            bed.setWard(bedDto.getWard());
            bedRepository.save(bed);
            editedBedDto.setBedNumber(bedDto.getBedNumber());
            editedBedDto.setWard(bedDto.getWard());
        }
    }
    System.out.println(GcStatsUtil.getGcStats());
    return editedBedDto;
}
```

Optimized Code: The optimized code rather applies synchronization of code block instead of the entire function. This works by allowing a thread to rather lock a specific bed resource it's looking to modify. This allows other threads to also at the same time access other bed resource to modify without interfering with the work of the other thread. This is possible by locking the specific bed resource rather than locking the entire update bed function.

```
public BedDto updateBed(BedDto bedDto) throws Exception {
    BedDto editedBedDto = new BedDto();
    if (bedDto.getId() != null) {
        Optional<Bed> editBed = bedRepository.findById(bedDto.getId());
        if (editBed.isPresent()) {
            Bed bed = editBed.get();
            synchronized (bed) {
                bed.setBedNumber(bedDto.getBedNumber());
                bed.setWard(bedDto.getWard());
                bedRepository.save(bed);
            }
            editedBedDto.setBedNumber(bedDto.getBedNumber());
            editedBedDto.setWard(bedDto.getWard());
        }
    }
```

```
    System.out.println(GcStatsUtil.getGcStats());
    return editedBedDto;
}
```

Implementation of Caching

Inefficient Code: The various read functions didn't implement caching. This leads to high latency whenever a bed detail or bed list has to be fetched for the client.

Optimized Code: The optimized code implemented caching using Ecache. This helps reduce latency when frequently accessed data is being requested by a client.

Adherence to 12 factor principles

Principle	Adherence
Code Based	Adhered
Dependencies	Adhered
Config	Adhered
Backing Service	Adhered
Build, release, run	Not adhered
Processes	Adhered
PortBinding	Adhered
Concurrency	Not adhered
Disposability	Not adhered
Development and production	Not adhered
Logs	Not Adhered
Admin processes	Not adhered