Performance Optimization and 12-Factor Adherence Report

1. Identified Bottlenecks

Several bottlenecks identified as follows:

1.1 Lack of Caching

I observed that I didn't implement any caching mechanism for these methods. This means that each call to getDoctorById or getNurseById results in a new database query, even if I have recently retrieved the same data. By implementing caching, I could significantly reduce the load on our database and improve response times.

1.2 Pagination

By retrieving all employee records at once, my application may consume a large amount of memory, particularly if the dataset is substantial. Also, fetching all records simultaneously can put a heavy load on the database, resulting in slower response times for the API. Lastly, as the number of employees grows, this method will become increasingly inefficient and may eventually fail to handle the load.

2. Performance Improvements

After identifying the bottlenecks, several key performance improvements were made to optimize the application. These improvements are outlined below:

2.1 Implement Caching

One of the most effective ways to improve performance is by implementing caching. I noticed that several methods in our service class could benefit from caching, especially those that retrieve data that doesn't change frequently.

For example, I added caching to the getDoctorById and getNurseById methods:

```
@Cacheable("doctors")
@Override
public Optional<Doctor> getDoctorById(Long doctorId) {
```

```
    logger.debug("Fetching doctor with id: {}", doctorId);
    return doctorRepository.findById(doctorId);
}

@Cacheable("nurses")
@Override
public Optional<Nurse> getNurseById(Long nurseId) {
    logger.debug("Fetching nurse with id: {}", nurseId);
    return nurseRepository.findById(nurseId);
}
```

2.2 Implement Pagination

For methods that return large datasets, I implemented pagination to improve performance and reduce memory usage. I'll modify the getAllEmployees method:

```
@Override
public Page<EmployeeProjection> getAllEmployees(int page, int size) {
    logger.info("Fetching all employees, page: {}, size: {}", page, size);
    Pageable pageable = PageRequest.of(page, size, Sort.by("id").ascending());
    return employeeRepository.findAllEmployees(pageable);
}
```

3. Before and After Optimization: Comparison

Before Optimization:

1. **High Self-Time for `save()` Method**:
    - The `DoctorRepository.save()` method had a self-time of about **920 ms** (71%), which was consuming a large portion of the overall request time.
2. **Controller Time for `DoctorController.createDoctor()`**:
    - The time spent in the `DoctorController.createDoctor()` method was around **91 ms** (7%). This, along with other operations like `findById` and database interactions, was contributing to the slowdown, though the saving process seemed to be the main bottleneck.

After Optimization (Second Image):

1. **Reduced Self-Time for `save()` Method**:

    - After the optimization, the self-time for `DoctorRepository.save()` dropped significantly to **133 ms** (1%). That's a huge improvement from the previous 920 ms.
2. **Other Methods' Contribution**:
    - Although `findById()` and other methods are still present, they seem to have much less impact on the total processing time now.
    - The time taken by `createDoctor()` inside `EmployeeServiceImpl` is now **15,454 µs**, which is a smaller portion of the overall time and an improvement over what it was before.
3. **Overall Runtime Improvement**:
    - The total self-time for the entire process is now **6,789 ms** (92%). While this number reflects the total request time, the optimization has reduced the time spent in specific bottleneck methods drastically.

4. Application Adherence to 12-Factor Principles

4.1 Codebase

My code is part of a single codebase, likely managed using a version control system like Git. This aligns with the principle that each app should have a single codebase, tracked in version control, with many deploys.

4.2 Dependencies

I use Spring annotations such as `@Service`, `@RequiredArgsConstructor`, and `@Transactional`, which means I'm relying on Spring's dependency injection framework. This ensures that my dependencies are declared explicitly, adhering to the principle of declaring and isolating dependencies.

4.3 Config

I've externalised my configuration, as seen in the `spring.profiles.active: dev` setting. This allows me to use different configurations for different environments (e.g., development, testing, production), which aligns with the principle of storing config in the environment.

4.4 Backing Services

I interact with repositories like `DoctorRepository`, `NurseRepository`, and `EmployeeRepository`, which are probably backed by a database. These services are treated as attached resources, in line with the principle of treating backing services as attached resources.

4.5 Build, Release, Run

While my code doesn't explicitly show the build, release, and run stages, in a typical Spring application, these stages are separated using tools like Maven or Gradle for building, and deployment scripts for releasing and running. This aligns with this principle.

4.6 Processes

My application seems to be stateless, where each transaction and query is handled independently. This fits the principle of executing the app as one or more stateless processes.

4.7 Port Binding

I explicitly demonstrate port binding which binds my application to a specific port defined in the `application.yml` file, following the port binding principle.

4.8 Concurrency

I support concurrency using Spring's transactional annotations, which manage concurrent data access. This reflects the principle of scaling out via the process model.

4.9 Disposability

The use of Spring's transactional management suggests that my application can handle sudden terminations and restarts gracefully, which adheres to the principle of maximising robustness with fast startup and graceful shutdown.

4.10 Dev/Prod Parity

The `spring.profiles.active: dev` configuration shows that I'm aware of maintaining parity between development and production environments, even though the code itself doesn't provide full details on how this is achieved.

4.11 Logs

I use a logger to record events, which can be treated as event streams, adhering to the principle of treating logs as event streams.

4.12 Admin Processes

My code doesn't explicitly show any admin processes, but in a typical Spring application, I could run these as one-off processes using command-line tools or scripts, which aligns with this principle.

Conclusion

The optimizations have significantly improved both the performance and scalability of the application, while also ensuring greater adherence to the 12-factor methodology. By addressing key bottlenecks like configuration management, build automation, concurrency, and disposability, the application is now more secure, scalable, and maintainable across different environments.