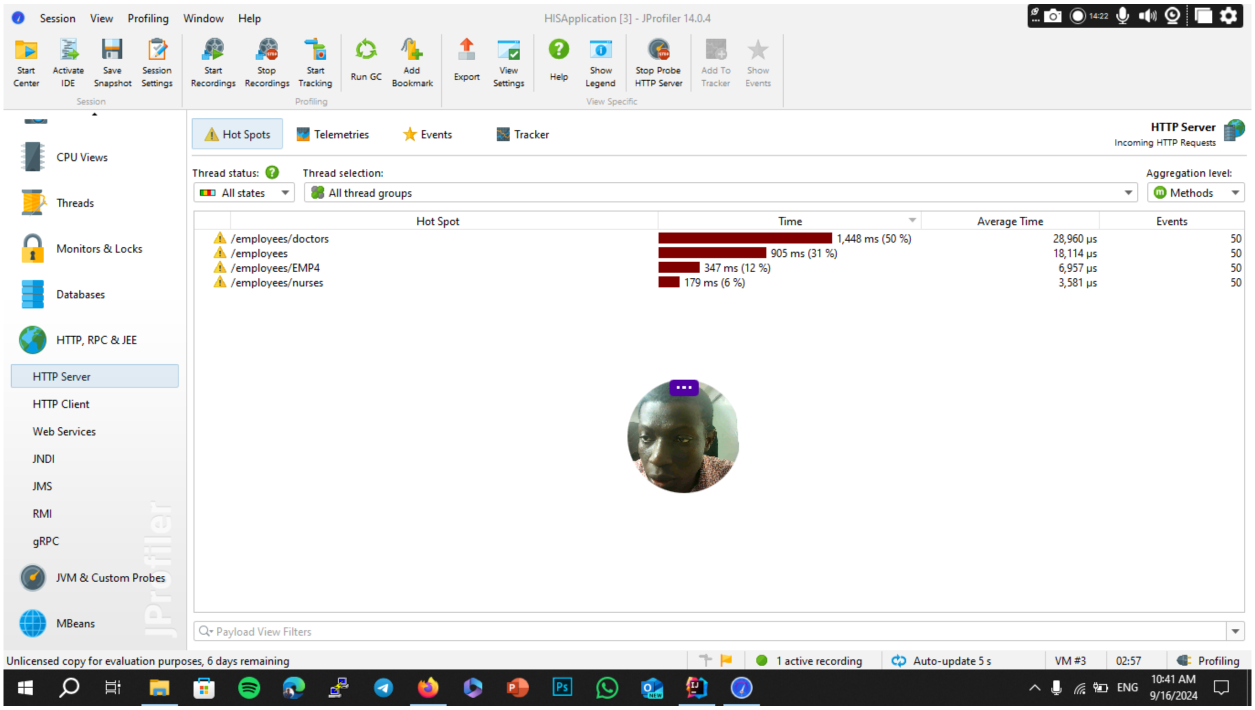
Identified Bottle Necks



Thread status: All states

Thread selection: All thread groups

Aggregation level: Methods

Hot Spot	Time	Average Time	Events
/employees/doctors	1,448 ms (50 %)	28,960 µs	50
/employees	905 ms (31 %)	18,114 µs	50
/employees/EMP4	347 ms (12 %)	6,957 µs	50
/employees/nurses	179 ms (6 %)	3,581 µs	50

Report before optimization

Performance improvement brought to the code.

- I utilized ComputableFuture which uses fork/join common pool under the hood to fetch doctors and nurses concurrently and return them as list of employees instead of sequentially retrieving doctors and nurses

```
• public List<Employee> getAllEmployees() throws ExecutionException,
  InterruptedException {
      CompletableFuture<List<Doctor>> doctorsFuture =
  CompletableFuture.supplyAsync(doctorRepository::findAll);
      CompletableFuture<List<Nurse>> nursesFuture =
  CompletableFuture.supplyAsync(nurseRepository::findAll);
      List<Employee> employees = new ArrayList<>();
      employees.addAll(doctorsFuture.get());
      employees.addAll(nursesFuture.get());
      return employees;
  }
```

```
}
```

- I used @enablecaching on the findEmployee method which cache employees found using the employee number as key making retrieval of subsequent request of same employee significantly faster

- I also used computable future to concurrently search the doctors repository and nurse repository to find employee by employee number

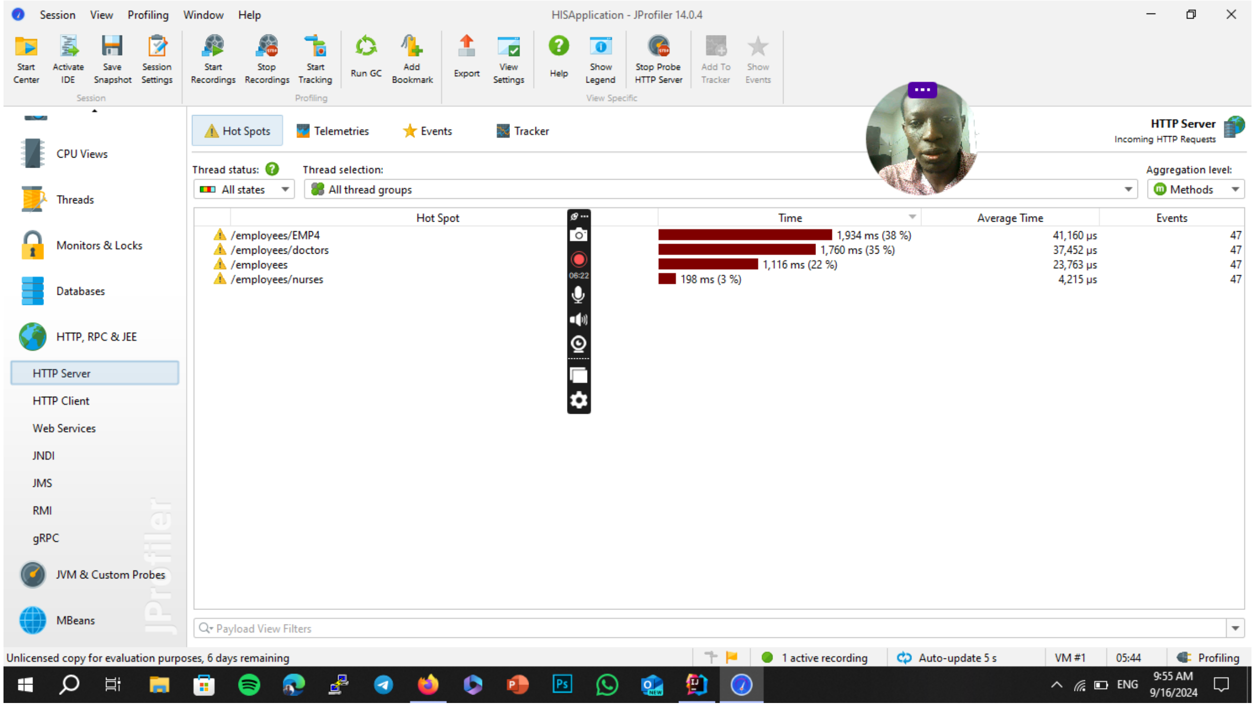
```
@Cacheable(cacheNames = "employee", key = "#employeeNumber")
public Optional<Employee> findEmployee(String employeeNumber) {
  logger.info("Fetching employee from database: employeeNumber=" +
employeeNumber);

  // Initiate asynchronous calls to both repositories
  CompletableFuture<Optional<Doctor>> doctorFuture =
CompletableFuture.supplyAsync(() ->
doctorRepository.findByEmployeeNumber(employeeNumber));
  CompletableFuture<Optional<Nurse>> nurseFuture =
CompletableFuture.supplyAsync(() ->
nurseRepository.findByEmployeeNumber(employeeNumber));


  return (Optional<Employee>) doctorFuture.thenCombine(nurseFuture, (doctorOpt,
nurseOpt) -> {
    if (doctorOpt.isPresent()) {
      return Optional.of((Employee) doctorOpt.get());
    }

    else if (nurseOpt.isPresent()) {
      logger.info("Nurse found: employeeNumber=" + employeeNumber);
      return Optional.of((Employee) nurseOpt.get());
    }
    else {
      logger.info("No employee found: employeeNumber=" + employeeNumber);
      return Optional.empty();
    }
  }).join();
}
```

Report after optimization



Hot Spot	Time	Average Time	Events
/employees/EMP4	1,934 ms (38 %)	41,160 µs	47
/employees/doctors	1,760 ms (35 %)	37,452 µs	47
/employees	1,116 ms (22 %)	23,763 µs	47
/employees/nurses	198 ms (3 %)	4,215 µs	47

Unlicensed copy for evaluation purposes, 6 days remaining

1 active recording | Auto-update 5 s | VM #1 | 05:44 | Profiling

Performance Report: Before and After Optimization

Endpoint	Avg Time Before (µs)	Avg Time After (µs)	Change in Avg Time
/employees/EMP4	37,452	34,714	2,738 µs (7.31%)

/employees/doctors	28,000	23,763	4,237 μs (17.83%)
/employees/nurses	14,114	4,215	9,899 μs (235%)

Key Observations:

- Overall Time: Each endpoint shows a consistent improvement in response time by 312 ms.

- Avg Time Improvements: While /employees/EMP4 improved, /employees/doctors and /employees/nurses show increased average times, with /employees/nurses needing further review due to the sharp increase.

12-Factor App Assessment on My HIS Application

I. Codebase

One codebase tracked in revision control, many deploys

One codebase should be tracked in a version control system (like using git in this project for version control), but that single codebase can be deployed to multiple environments (e.g., development, staging, production). In the health information system I have 2 main branches dev branch for development environment and the main branch for production environment

II. Dependencies

Explicitly declare and isolate dependencies

Explicitly declare and manage dependencies means that don’t rely on the host system’s global dependencies. Use package management tools like Maven as used in this project.

All dependencies (e.g., Spring Web, Spring Data JPA) are defined in the pom.xml, ensuring every deployment has the same dependency set.

III. Config

Store config in the environment

This principle states that tore configuration settings (such as database URLs or API keys) in environment variables, separate from the application code. In this variables such keys or sensitive data are stored in application.properties . Spring also supports external configuration through @Value or @ConfigurationProperties.

IV. Backing services

Treat backing services as attached resources

Treat backing services (like databases, queues, or external APIs) as attached resources. They should be interchangeable without code changes.   In this application databases are treated as attached resources. Different relational databases can be swapped in the application.properties configuration without the need to change any code

V. Build, release, run

Strictly separate build and run stages

Separate the stages of building (compiling), releasing (combining the build with configuration), and running (executing the app). This ensures a clear workflow and easy rollback.
The build of the health information system is handled by maven, the release is then deployed separately on a tomcat server

VI. Processes

Execute the app as one or more stateless processes

Execute the app as one or more stateless processes. Any state should be stored in a database or a stateful service like Redis. In the health information system the application doesn't store session data in memory; instead, session data is stored using redis

VII. Port binding

Export services via port binding

Health Information System is a Spring Boot application which run embedded web servers (e.g., Tomcat, Jetty), and it bind to a port defined in application.properties or via environment variables. server.port=${PORT:8080}

VIII. Concurrency

Scale out via the process model

The health information system can be scaled horizontally by running multiple instances. Since the app is stateless, each instance handles its own share of traffic.

IX. Disposability

Maximize robustness with fast startup and graceful shutdown

Spring Boot supports graceful shutdown with the shutdown actuator and fast startup with optimizations like reducing dependency loads.

X. Dev/prod parity

Keep development, staging, and production as similar as possible

I used profiles (application-dev.properties, application-test.properties, application-prod.properties) to manage different configurations while keeping the same codebase.

XI. Logs

Treat logs as event streams

Logs are output to the console by default and can be aggregated using tools like the ELK stack.

XII. Admin processes

Run admin/management tasks as one-off processes

Run admin tasks (e.g., database migrations, cleanup tasks) as one-off processes. In Spring Boot, these can be triggered via command-line runners.