Caching & Advanced Exception Handling
Amo Samuel

In this project, I've explored and implemented caching mechanisms to optimize application performance, reduce database load, and enhance the overall user experience. Additionally, I've employed a structured exception handling approach to make error reporting more efficient and user-friendly. Below, I will explain the caching strategies I used and how I implemented exception handling.

Caching Strategy 1: Caffeine (In-Memory Cache)

For rapid data retrieval within a single instance of the application, I chose **Caffeine** as the in-memory caching mechanism. It's lightweight, fast, and allows fine-tuned control over cache entries.

```
cache:
    caffeine:
      spec: maximumSize=1000,expireAfterWrite=1m,recordStats
```

In my DoctorService, I've implemented caching to retrieve the list of all doctors. This prevents repetitive database queries for the same data:

```
@Cacheable(value = "doctors", key = "'all_doctors'")  1 usage   samuelamo001
public List<DoctorDTO> finAllDoctors(){
    List<Doctor> doctors = doctorRepository.findAll();

    logger.info("Doctors have been retrieved from the database successfully");
    return doctors
            .stream() Stream<Doctor>
            .map(doctorMapper::convertToDTO) Stream<DoctorDTO>
            .collect(Collectors.toList());
}
```

Caching Strategy 2: Redis (Distributed Cache)

For a more scalable approach, I incorporated **Redis** as a distributed cache. Redis allows me to store cache data across different instances of the application, which ensures data consistency and availability in distributed environments.

Redis Cache Manager Setup:

I've configured the Redis cache manager with a time-to-live (TTL) of 1 minute for each entry:

```
@Bean  samuelamo001
CacheManager redisCacheManager() {
    RedisCacheConfiguration redisCacheConfiguration = RedisCacheConfiguration.defaultCacheConfig()
            .entryTtl(Duration.ofMinutes(1))
            .disableCachingNullValues()
            .serializeValuesWith(RedisSerializationContext.SerializationPair.fromSerializer(new GenericJa

    RedisCacheWriter redisCacheWriter = RedisCacheWriter.lockingRedisCacheWriter(jedisConnectionFactory()

    return RedisCacheManager.builder(redisCacheWriter)
            .cacheDefaults(redisCacheConfiguration)
            .transactionAware()
            .build();
}
```

Composite Caching Strategy

By using a **CompositeCacheManager**, I'm combining both Caffeine and Redis caching strategies. This gives me the flexibility to use in-memory caching for quick, localized queries, while Redis handles more complex, distributed scenarios.

This approach allows me to use both Caffeine and Redis simultaneously, taking advantage of the strengths of each caching strategy.

```java
@Bean  samuelamo001 *
CacheManager caffeineCacheManager(){
    CaffeineCacheManager caffeineCacheManager = new CaffeineCacheManager();
    caffeineCacheManager.setCaffeine(Caffeine.newBuilder()
            .maximumSize(1000));
    return caffeineCacheManager;
}

@Bean  samuelamo001
@Primary
CacheManager cacheManager(){
    return new CompositeCacheManager(caffeineCacheManager(), redisCacheManager());
}
```

```
CaffeineCacheManager caffeineCacheManager = new CaffeineCacheManager()
springdata
```

Exception Handling

I've created custom exceptions to handle specific errors related to doctors, such as when a doctor is not found or when there's a failure during creation. By using custom exceptions, I can provide clear error messages and manage them appropriately in the global handler.

This custom exception is thrown when a doctor with a given ID is not found in the system.

```java
package springdata.week1.springdata.Execptions.doctor;

public class DoctorCreationException extends RuntimeException {  4 usages  samuelamo001
    public DoctorCreationException(String message) {  no usages  samuelamo001
        super(message);
    }
}
```

Global Exception Handler

To avoid writing repetitive error-handling code throughout the application, I've centralized all exception handling in the GlobalExceptionHandler class. This class uses the @ControllerAdvice annotation to intercept exceptions globally and return custom error responses.

```java
public class GlobalExceptionHandler {

    private static final Logger logger = LoggerFactory.getLogger(GlobalExceptionHandler.class);  3

    @ExceptionHandler(DoctorNotFoundException.class)  samuelamo001
    public ResponseEntity<Object> handleDoctorNotFoundException(DoctorNotFoundException exception)
        logger.error("Doctor not found: {}", exception.getMessage());
        DoctorErrorResponse errorDetails = new DoctorErrorResponse(LocalDateTime.now(), exception.g
        return new ResponseEntity<>(errorDetails, HttpStatus.NOT_FOUND);
    }

    @ExceptionHandler(DoctorCreationException.class)  samuelamo001
    public ResponseEntity<Object> handleDoctorCreationException(DoctorCreationException exception)
        logger.error("Doctor creation failed: {}", exception.getMessage());
        DoctorErrorResponse errorDetails = new DoctorErrorResponse(LocalDateTime.now(), exception.g
        return new ResponseEntity<>(errorDetails, HttpStatus.BAD_REQUEST);
    }
```

Error Response Structure

I've structured my error responses using an ErrorResponse class that contains a timestamp and a message. This gives clients clear, consistent feedback about when the error occurred and what went wrong.

By keeping error messages structured and consistent, it's easier to debug issues, and the client gets meaningful, actionable information.

```java
@AllArgsConstructor  7 usages  samuelamo001
@NoArgsConstructor
@Getter
@Setter
public class DoctorErrorResponse {

    private LocalDateTime timestamp;
    private String message;
}
```

Conclusion

By combining Caffeine and Redis caching strategies, I’ve created a flexible and powerful caching system that optimizes performance and scalability. Caffeine is ideal for quick, localized access to data, while Redis provides the reliability and distribution needed for larger-scale applications.

On the exception handling side, I’ve ensured that the application responds gracefully to errors, providing useful information to both developers and clients. This approach not only improves the overall robustness of the system but also enhances the user experience by making errors easier to understand and address.