Caching Strategies and Exception Handling in Spring Boot

I focused on implementing caching strategies and handling exceptions effectively in a Spring Boot application. Below is a detailed walkthrough of my progress:

1. Caching in Java with Caffeine

I used **Caffeine** to implement in-memory caching for queries related to finding doctors by their specialties. This allowed me to avoid repeated database hits and improve response times when the same queries were executed multiple times.

```
@Cacheable(value = "doctorsBySpeciality", key = "#speciality", unless = "#result.isEmpty()",
cacheManager = "caffeineCacheManager")
@Transactional(readOnly = true)
@Override
public List<Doctor> findDoctorsBySpeciality(String speciality) {
    logger.info("Finding doctors by speciality: {}", speciality);
    List<Doctor> doctors = doctorRepository.findBySpeciality(speciality);
    logger.info("Found {} doctors with speciality: {}", doctors.size(), speciality);
    return doctors;
}
```

In the above method, the `@Cacheable` annotation caches the result of the `findDoctorsBySpeciality` method using **Caffeine**. The caching logic ensures that an empty result set isn't cached, saving cache memory for relevant results.

2. Redis Integration

To implement a distributed cache, I integrated **Redis** as the cache provider for retrieving department information. Redis allows for better scalability and persistence, unlike in-memory caches like Caffeine, which are limited to individual JVM instances.

```
@Override
@Cacheable(value = "departments", key = "#id", cacheManager = "redisCacheManager")
public Department getDepartmentById(Long id) {
    return departmentRepository.findById(id).orElseThrow(() -> new
DepartmentNotFoundException("Department not found with id: " + id));
}
```

This method leverages **Redis** for caching, enabling faster lookups of department details, while ensuring that cache invalidation happens when data changes.

3. Caching Strategies

I implemented both **write-through** and **write-behind** caching strategies. These strategies help ensure consistency between the cache and the underlying database.

- **Write-through** ensures that any updates to the database are immediately written to the cache.
- **Write-behind** can be implemented by delaying updates to the cache, prioritizing performance at the cost of immediate consistency.

I used Caffeine and Redis to demonstrate these caching strategies. Caffeine's in-memory cache supports a fast lookup, while Redis ensures data durability in a distributed environment.

4. Advanced Exception Handling

Handling exceptions effectively is critical for application stability. I implemented custom exception handlers using `@ControllerAdvice` and `@ExceptionHandler` annotations. This global exception handler ensures a consistent response structure for different error conditions.

```
@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(NotFoundException.class)
    @ResponseStatus(HttpStatus.NOT_FOUND)
    public ResponseEntity<String> handleNotFoundException(NotFoundException ex) {
        return new ResponseEntity<>(ex.getMessage(), HttpStatus.NOT_FOUND);
    }

    @ExceptionHandler(Exception.class)
    @ResponseStatus(HttpStatus.INTERNAL_SERVER_ERROR)
    public ResponseEntity<String> handleGenericException(Exception ex) {
        return new ResponseEntity<>("An unexpected error occurred: " + ex.getMessage(),
HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

The **`GlobalExceptionHandler`** class centralised exception handling, improving code readability and maintainability. I created a custom `NotFoundException` to handle cases where resources were not found, such as departments or doctors. This exception results in a `404 Not Found` response.

5. Configuration

In terms of configuration, I set up both Redis and Caffeine cache managers in my Spring Boot application:

- **Caffeine Cache Manager**:

```
@Configuration
@EnableCaching
public class CacheConfig {

    @Bean("caffeineCacheManager")
    public CacheManager caffeineCacheManager() {
        CaffeineCacheManager cacheManager = new CaffeineCacheManager("doctorsBySpeciality");
        cacheManager.setCaffeine(caffeineCacheBuilder());
        return cacheManager;
    }

    Caffeine<Object, Object> caffeineCacheBuilder() {
        return Caffeine.newBuilder()
                .initialCapacity(100)
                .maximumSize(500)
                .expireAfterWrite(1, TimeUnit.HOURS)
                .recordStats();
    }
}
```

- **Redis Cache Manager**:

```
@Configuration
@EnableCaching
public class RedisConfig {

    @Primary
    @Bean("redisCacheManager")
    public CacheManager redisCacheManager(RedisConnectionFactory connectionFactory) {
        RedisCacheConfiguration cacheConfig = RedisCacheConfiguration.defaultCacheConfig()
                .entryTtl(Duration.ofMinutes(10))
                .disableCachingNullValues()
                .serializeValuesWith(RedisSerializationContext.SerializationPair.fromSerializer(RedisSerializer.json()));
        return RedisCacheManager.builder(connectionFactory)
                .cacheDefaults(cacheConfig)
                .build();
    }
}
```

In the `application.yml`, I configured Redis and Caffeine cache managers as well as database connections:

```
spring:
  application:
    name: hospitalmgmt
  datasource:
    url: jdbc:postgresql://localhost:5432/test?currentSchema=hospital_mgmt
    username:
    password:
  data:
    redis:
      host: 192.168.56.38
```

```
      port: 6379
    main:
    allow-bean-definition-overriding: true
server:
  port: 8083
logging:
  level:
    org:
      springframework:
        web: TRACE
```

Conclusion:

By utilising **Caffeine** and **Redis** for caching, I was able to significantly improve the performance of the hospital management system. The advanced exception handling mechanisms provided a robust and user-friendly error response. This lab enhanced my understanding of caching strategies, exception handling, and Spring Boot's integration capabilities.