ADP Reflection

Group:
- Paul Kaufmann
- Viany Manuel
- Deniz Özcan
- Samuel Fiedorowicz

Design Choices

Proxy Pattern

We recognized that constantly querying the database for the same set of hotel room data was both inefficient and resource-intensive. This led us to the decision to implement a caching mechanism. The Proxy pattern offered a straightforward solution to intercept and manage these requests, caching the data and serving it from memory on subsequent requests. By caching data, the system minimizes redundant database calls, which reduces load on the database and speeds up the response times for the end-user.

This pattern is especially useful in scenarios where the data (hotel rooms in this case) remains consistent over time. As outdated data can be an issue with caching, in our integration of this pattern, the execution of multiple database calls when the data is not yet stale has to be considered. Fortunately, as hotel rooms are not changed at all (and in practice wouldn't be changed too frequently), thus this is just a theoretical issue for this app.

Chain of Responsibility

The Chain of Responsibility pattern was used to manage how booking requests are handled. In this project, it works like a line where each request goes from one handler to the next until one can deal with it. This setup makes it easier to add new steps or change how requests are handled without messing up the whole system. This way, the request keeps moving smoothly until it finds the right handler to complete the task.

This approach is helpful because it separates the different parts of handling a booking, like reserving a room, proceeding to payments. Plus, it's set up in a way that lets you easily add more handlers in the future. This makes the booking process not only flexible but also extendable for any changes or additions in the future.

Strategy Pattern

Employing the Strategy pattern for customer offers based on membership levels (Regular, Gold, Platinum) illustrates a keen understanding of object-oriented design principles. This pattern allows for the dynamic alteration of the algorithm used for offer calculation based on the customer's membership, promoting flexibility and scalability. It's a clear reflection of forward-thinking, preparing the system for easy extension with new membership types or offer calculation strategies without modifying existing code.

Integration of Resilience Patterns and Strategies

Caching

We used the Ehcache package for caching. It is used to cache rooms to speed up responses and reduce the workload. We used caching the following use cases

Getting All Rooms: When all rooms are returned, the gateway checks the cache. If they are cached (and not expired), the rooms are returned right away; if not, the rooms are fetched form the database and entered into the cache.

For One Room: Similarly, when someone wants details about a specific room, the gateway first looks in the cache. If the room is cached, it is returned; if not, it fetches the room and caches it

Benefits

- Speed up of requests
- Less load on the database
- Configurable

Configuration

```
@Component(value = "roomsCacheManagerService")
public class RoomsCacheManagerService {
    public final String ROOMS_CACHE_KEY = "rooms";
    public final String SINGLE_ROOM_CACHE_KEY = "room-id";

    public final CacheManager cacheManager = CacheManagerBuilder
            .newCacheManagerBuilder()
            .withCache(ROOMS_CACHE_KEY, CacheConfigurationBuilder
                    .newCacheConfigurationBuilder(String.class, RoomDto[].class,
                            ResourcePoolsBuilder.heap(10))
                    .withExpiry(ExpiryPolicyBuilder.timeToLiveExpiration(Duration.ofSeconds(5))))
            .withCache(SINGLE_ROOM_CACHE_KEY, CacheConfigurationBuilder
                    .newCacheConfigurationBuilder(Long.class, RoomDto.class,
                            ResourcePoolsBuilder.heap(10))
                    .withExpiry(ExpiryPolicyBuilder.timeToLiveExpiration(Duration.ofSeconds(5))))
            .build(true);

}
```

Circuit Breaker Pattern + Retries

The gateway handles the communication to the other services. It connects to the booking service (http//localhost:8081) and the room-service (http://localhost:8082). The main service reaches the other services via HTTP requests. As HTTP is synchronous and any exception that occurs when contacting the other services would result in an error for the users. Some of these errors are only temporal issues and can be solved by simply retrying an operation. Thus we use the circuit breaker pattern (by Resilience4j) for taking breaks between retries. In this setup, the circuit breaker helps deal with short-term problems like temporary server errors or slowdowns. It watches how often these problems happen and if they occur too much, it stops such requests for 50 seconds (configurable) before retrying. This break gives

things time to fix themselves without overloading the system or the server. After the break requests are performed again. If the requests are successful, the wait period is reset; if not, it takes another break. This way, the circuit breaker helps keep the system stable by not letting temporary issues cause bigger problems.

We configured it to handle three kinds of exceptions

- HTTP Exceptions with status code 5xx: org.springframework.web.client.HttpServerErrorException
- Timeouts: java.util.concurrent.TimeoutException
- IO-Exceptions: java.io.IOException

Benefits:

- Automatically gives a service time to self-recover
- Handling of transient errors

Bulkhead Pattern

We used the Resilience4j bulkhead pattern to limit how many concurrent calls (or tasks) can be executed at the same time for specific parts of your application. We use this pattern to reduce the overall load on the service. Our configuration is as follows:

- **maxConcurrentCalls**: 10: This setting means that only 10 calls can be handled at the same time. If your service is handling a request to fetch room details (singleRoom) or a list of rooms (rooms), and there are already 10 requests being processed, an 11th request will have to wait.
- **maxWaitDuration**: 10ms: This is how long a new request will wait for its turn to be processed if the maximum number of concurrent calls is reached. In this case, if there's no room to start handling a new request within 10 milliseconds, the request will be rejected or fail.

Benefits:

- Reduces load
- Resource distribution

Rate Limiting

We use Resilience4j rate limiting to protected the gateway service (and the whole app) from overloading. As this pattern is well known, here is the configuration we used:

- **limitForPeriod**: 10: This setting allows up to 10 requests to be processed within the specified period.
- **limitRefreshPeriod**: 1s: The specified period here is 1 second. This means after every second, the count of allowed requests resets to 10.
- timeoutDuration: 0: This indicates there is no waiting time for requests. If a request comes in and the rate limit has already been reached for that second, the request will be immediately rejected instead of being queued.

Benefits:

- Prevent overloads
- Manage traffic

Learnings/Insights

Implementing the proxy, chain-of-responsibility, and strategy patterns has significantly made the code more complex but more resilient to errors.. We think the services are more stable, but remain easy to change in the future. These patterns allow for greater extensibility, as they enable the addition of new functionality with minimal changes to existing code. By distributing responsibilities across different components and using abstraction, the logic is spread evenly across the codebase, making it easier to understand, maintain, and extend.

We learned that many exceptions are often temporary, such as temporary network failures or server overloads. The circuit breaker pattern plays a crucial role in managing these transient exceptions. By temporarily halting requests to a failing service, it gives the service time to recover. We liked how this pattern is very simple to add, but provides a lot of benefits for it.

It was good practice to implement rate limiting. There are many packages readily available and it is easy to integrate, offering a cost-effective solution to manage request rates without requiring a lot of boilerplate code.

In conclusion, the application of design patterns such as proxy, chain-of-responsibility, and strategy has streamlined the project's structure, making it more logical and easier to manage. Meanwhile, resilience patterns like circuit breakers and rate limiters are critical in handling transient exceptions and managing request flows, ensuring that the system remains robust and responsive under varying conditions. These strategies collectively contribute to a more resilient, maintainable, and scalable application.