System Performance
Submission 4 Specification

[404 Not Found]
SWEN90007 SM2 2023 Project

Members

1. Jun Xu (1074120) juxu1

Github username: JunXu-1

juxu1@student.unimelb.edu.au

2. Shuowen Yu (1174223) shuoweny

Github username: shuoweny

shuoweny@student.unimelb.edu.au

3. Yuchen Cao (1174014) yuccao

Github username: YuchenCAO01

yuccao@student.unimelb.edu.au

4. Zhaolong Meng (1118637) zhaolongm

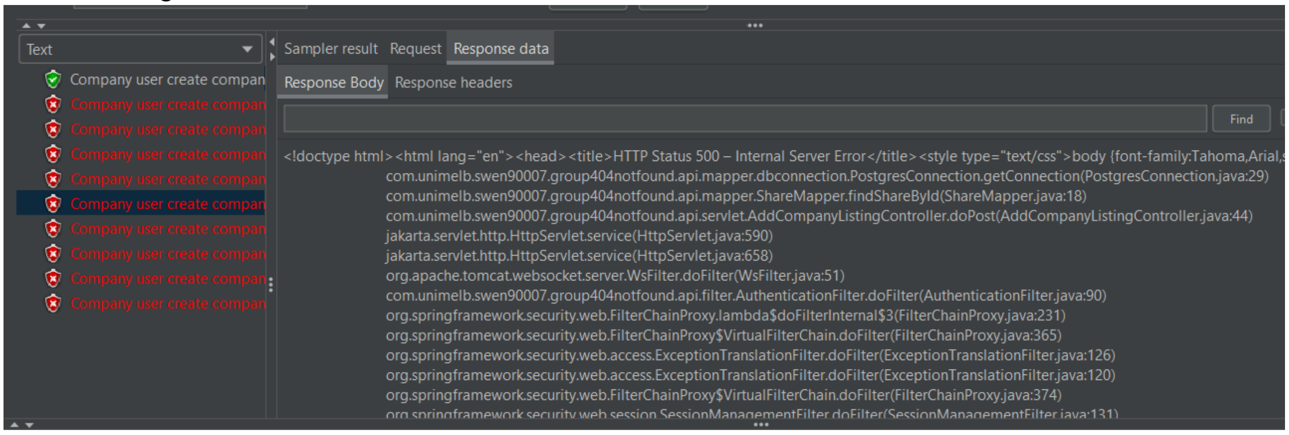
Github username: kdyxnm

zhaolongm@student.unimelb.edu.au

1. Areas to Improve

1.1 No Connection Pool

The below image shows an error when the number of connections has exceeded the limitation.


Text
Company user create compan
Company user create compan
Company user create compan
Company user create compan
Company user create compan
Company user create compan
Company user create compan
Company user create compan
Company user create compan
Company user create compan
Sampler result
Request
Response data
Response Body
Response headers
Find

```
<!doctype html><html lang="en"><head><title>HTTP Status 500 – Internal Server Error</title><style type="text/css">body {font-family:Tahoma,Arial,s
                com.unimelb.swen90007.group404notfound.api.mapper.dbconnection.PostgresConnection.getConnection(PostgresConnection.java:29)
                com.unimelb.swen90007.group404notfound.api.mapper.ShareMapper.findShareById(ShareMapper.java:18)
                com.unimelb.swen90007.group404notfound.api.servlet.AddCompanyListingController.doPost(AddCompanyListingController.java:44)
                jakarta.servlet.http.HttpServlet.service(HttpServlet.java:590)
                jakarta.servlet.http.HttpServlet.service(HttpServlet.java:658)
                org.apache.tomcat.websocket.server.WsFilter.doFilter(WsFilter.java:51)
                com.unimelb.swen90007.group404notfound.api.filter.AuthenticationFilter.doFilter(AuthenticationFilter.java:90)
                org.springframework.security.web.FilterChainProxy.lambda$doFilterInternal$3(FilterChainProxy.java:231)
                org.springframework.security.web.FilterChainProxy$VirtualFilterChain.doFilter(FilterChainProxy.java:365)
                org.springframework.security.web.access.ExceptionTranslationFilter.doFilter(ExceptionTranslationFilter.java:126)
                org.springframework.security.web.access.ExceptionTranslationFilter.doFilter(ExceptionTranslationFilter.java:120)
                org.springframework.security.web.FilterChainProxy$VirtualFilterChain.doFilter(FilterChainProxy.java:374)
                org.springframework.security.web.session.SessionManagementFilter.doFilter(SessionManagementFilter.java:131)
```

While we use Jmeter to do load testing, we find that the connections of the database have exceeded the limitation, and the testing can not keep running. That's because we did not use the connection pool. Without a connection pool, each database operation necessitates creating a new database connection via DriverManager. This approach has multiple drawbacks. Establishing a new connection generally involves multiple rounds of network communication and resource allocation, taking up both time and resources. This approach is also ineffective. In high-concurrency environments, database servers may quickly run out of resources as each incoming request attempts to establish new connections with them. Manually managing database connections increases resource leakage risks significantly. If a connection is left open too long, it will continue consuming database resources, further restricting available resources and decreasing the performance of applications. Furthermore, managing connections this way often does not include advanced features like reuse, health checks and automatic reconnections that would enhance the robustness and performance of our share trading system.

1.2 Concurrency Pattern

Our current system setup uses only an optimistic locking mechanism, and it will cause a large number of requests to fail while a few can be successful. Our desired behaviour would be for one transaction to succeed while all of its competitors roll back, providing better concurrency control and data consistency.
This situation indicates two potential issues. First, locking mechanisms might be too restrictive, leading to multiple transactions waiting for the same resource and ultimately all failing due to timeouts or version checks.
To meet our objective of only permitting one transaction at a time to succeed while all others fail, further analysis and redesign of current locking strategies are necessary.

1.3 Domain Model

During our project development, we progressively recognised a series of critical issues in our domain model, which directly impacted the adaptability and efficiency of our database design. Firstly, we failed to consider the relationships between objects adequately when designing the domain model, leading to issues in association table mapping within the database, which could result in data inconsistency and reduced query efficiency. For example, when we try to access the company listing data, we need to query all the tables to get the information we want. More concerning was our omission of explicit many-to-many relationships in the domain model, causing us to miss the correct implementation of association table mapping in the database design; for example, he had to maintain a portfolio table to map all relevant tables together, further exacerbating data redundancy and performance decline. These misguided mappings and design choices have substantially diminished the overall performance of our system.
To address these concerns and optimise the system, we believe there's a necessary need to evaluate our domain model again. We need to ensure every mapping and relationship accurately reflects the actual requirements and aligns with the database properly.

2. Changes Applied

2.1 Use HikariCP to create Connection pool

Prior to adopting HikariCP, we relied on an ineffective connection management system which created a new database connection for every request that came in. Although functional, this approach resulted in significant performance costs, which limited the scalability of our application. With this in mind, we decided on HikariCP: an advanced production-ready connection pooling library for Java applications that delivers optimal scalability and performance. HikariCP provides significant performance gains by replacing our DriverManager.getConnection() method in calls with its DataSource.getConnection(); this enables incoming requests to use already established connections quickly without latency and CPU burden being introduced at each request.
HikariCP's configuration settings enabled us to optimise different aspects of our connection pool. For instance, we adjusted parameters like pool size, connection timeout and idle timeout based on our application's needs for optimal resource utilisation. Furthermore, HikariCP enabled us to set maximum pool sizes, which helped mitigate risks related to exhausting database resources that had previously been an issue with our previous setup.
HikariCP's transition has resulted in a more robust, efficient, and scalable application. Performance metrics indicate a marked decrease in database connection times, which has allowed faster response times and an overall better user experience. Furthermore, these changes have allowed us to better utilise system resources while also being more adaptable for higher volumes of concurrent requests.

3. Future Improvements

3.1 Combining Locks To Maximize Performance

At our previous setup, we experienced a high rate of transaction failures during concurrent scenarios, which was detrimental both in terms of performance and user experience. Most transactions rolled back, with only a minority successfully committing. To address this issue, we implemented a hybrid locking strategy combining both optimistic and pessimistic locking mechanisms in order to mitigate this problem.
Where contention was relatively low, we employed Optimistic Locking. This technique allowed multiple transactions to proceed simultaneously under the assumption that conflicts would be rare.
Pessimistic Locking was employed for sections of code where high contention was anticipated to ensure that when one transaction secured a lock, it would block all subsequent ones until its release was granted. It provides data integrity and transactional consistency in critical operations.
By employing this hybrid approach, we achieved an effective balance between performance and reliability. Employing optimistic locking to enhance throughput for low conflict scenarios and pessimistic locking to safeguard transactional safeguards during critical sections of our application. Extensive testing confirmed this strategy's ability to reduce transaction failure rates significantly while increasing successful commits and improving our overall system performance.