Extra: Concurrency Consideration Report

Topicus Team 1

Umair Aamir Mirza (s2928558)
Martin Demirev (s2965046)
Condu Alexandru-Stefan (s2769549)
Narendra Setty (s2944200)
Alexandru Lungu(s3006301)
Teodor Pintilie (s2920344)

Introduction:

This is a brief report to address the concurrency considerations made for the scope of this project itself. It takes into account the application server's capacity, the structure of the database, and any functionality that we implemented to address concurrency.

Application Server Capacity:

According to the Apache Tomcat 10.1 documentation (Hanik, F.), the *maxThreads* property for the Tomcat Server is set to 200 by default, and *minThreads* (active, alive threads at all times) is set to 25. With this in mind, to scale the application to a greater user base towards the deployment, these values can easily be adjusted so that the server can accommodate a larger concurrent user-base. However, in our case, we were unfortunately already experiencing issues with Tomcat's recognition of the database driver, and amongst other issues, the development team decided that it was a better decision to in fact avoid performing changes to the internal structure of the server.

Database Structure and Queries:

As the user base can be accounted for by the server's capacity, the need for concurrent transactions is handled in the DAO methods respectfully. The use of *Connection* objects and *PreparedStatement* objects in Java enabled us to do the following:

1. Set an appropriate isolation level for the transaction.
2. Perform query with PreparedStatement.

How we structured such queries with the appropriate isolation levels can be shown below, divided into specific categories. The most important aspect to consider was the

modularity of individual components. For example, the parent dashboard will load specific metadata rather than entire objects, and clicking on a particular registration will then activate the fetch. This way, clients are not using obsolete data, and hence, requests being modular enable greater control over the concurrency of the application.

Query Group:	**Isolation Level:**	**Justification:**
Populating Pages	READ_COMMITTED	READ_COMMITTED guarantees that only committed changes to a similar resource being accessed are shared to the query. This means that the query results in the latest version of the data at the point in time when its execution completes. The concern with non-repeatable reads is addressed with the front-end request structure, where the specific data is actually queried only when it is required.
Specific Data Retrieval	READ_COMMITTED	When this data is required, it is queried. This was the policy adopted on the front-end, through specific request objects that are provided in the *Payload* on the back-end. Thus, it provides the latest version of the object available, and issues such as updating a deleted record are addressed with the URI construction and error reporting to the user.
Specific Data Submission	SERIALIZABLE	If the scenario is considered where a School Administrator changes the status of a Registration while a Parent works on it, when the results are submitted, the issue of data fields being changed will not arise. The specific construction of requests in the *Payload* folder prevents this from happening because an admin can only change the status of a registration, whereas a parent is only limited to changing the data fields. However, to prevent the case of two School Admins altering the same Registration, SERIALIZABLE is the appropriate transaction level. Hence, this accounts for any data overlapping being minimized, and SERIALIZABLE ensures that transactions are allowed to carry to completion.

Application Functionality:

As stated earlier, the *Payload* folder contains application request and response structure, which is done in a way that minimizes the risk of concurrent transactions overlapping and contaminating the database. Only components that are to be edited by a user are editable, blocking other user groups from editing similar components. What this achieves is an application that is well structured in its requests and responses, and essentially enables concurrent usability.

Additionally, with the knowledge of Hikari Connection Pools, in the early stages of the project, we attempted to create a basic implementation for the ConnectionPools. The design for this can be found in the folder *src/main/java/org/Topicus/Utility/Builders*, the file is called *ConnectionPool*. It was not used in the end due to implementation schedule time constraints, but provides an understanding of how we would have implemented the utilization of connection pooling to facilitate concurrency. Additionally, at the stage in the project where the pools were developed, we also did not request if Hikari Connection Pools could be used (as it was believed that the list of allowed packages was final), but had the development team inquired further, this could have been successfully implemented.

Citations:

Hanik, F. (2022, October 3). *Apache Tomcat 10 Configuration Reference (10.0.27) - The Executor (thread pool)*. Apache.org. https://tomcat.apache.org/tomcat-10.0-doc/config/executor.html