This project is intended to showcase and demonstrate through code the use of database locking mechanism to control concurrent users request to a queue-like assignment.

The description of a possible project could go like this: Multiple users processing bank loan applications have a list of various type of tasks from various managers.

They can select getWork for any of these tasks and the system will run a very heavy query against the database to retrieve the highest priority 1 to 5 loans matching the task. The query can take anywhere up to 30 seconds to run, with cases the query taking over a minute.

Once a loan gets into a user’s pipeline for a task X the same loan cannot be assigned to a different user for the same task X. The user can disposition the loan for the task and the loan can either be available again for selection if it still matches the criteria, unless the user decided it should be available after a certain amount of time or never again.

The issue arises when multiple users request loans from the same task at about the same time, or at least before previous query and assignment of loan/task/user has been completed.

In a simple non-concurrent implementation, Oliver might click getWork for Task X and a few seconds later Tracy does the same. Let's say loan 123 has the highest priority, and 20 seconds after Oliver requested work loan #123 will get assigned to Oliver to complete task X. Since Tracy's query is still running, and completes a few seconds after Oliver's, he's also going to have loan 123 showing as highest priority and the system will assign loan 123 for task X to Tracy as well.

If we didn't have a clustered environment then java's synchronized methods might have worked, but not in the case of over a dozen nodes in a cluster.

The solution came from the database layer, and somewhat less used and known locking mechanisms. Oracle, MySQL v 8+, and other databases support locking some rows when we select them for update. This is a concise description of the algorithm we implemented (minus some extra complexities and exact names of tables/fields). We came up with a temporary record of who requests what and gets assigned what to not even have to run the long query more than once at a time to avoid exploding waight times and time outs if many users try to get work for the same task. Description below:

1. Record user's request in a separate table (work\_requested) - requesting user id, task id, how many loans they wanted and number of loans actually assigned which is zero for now.

2. `SELECT t.\* from task t where t.task\_id=:taskId for update skip locked` -- the "for update" part in this query will put a lock on the task table row that we want to protect from concurrent assignment of the same row. Skip locked tells to ignore the row if it is locked.

3. If the result is empty it means the row is already locked by some other user. In this case the backend returns a busy flag back to the browser which has a re-check mechanism and will make another call in 5 then 10 then 20 seconds.

4. If the lock was successfully acquired then the query will return exactly one row. In the java code this will continue with the next step of running the heavy weighted query.

5. Once the query returns the list of loans, we look at the work\_requested table for the task\_id that have requested loans but not assigned loans. These are all the requests for work that came while the heavy query was running. Now we don't need to run that query again for additional users, we just start assigning loans (in the work table we write the user id, task\_id and loan\_id + other fields). The assignment of loans starts with the first user in work\_requested table and updating its assigned number. IF there were not enough loans then we decrease the requested number to what could be assigned - which will help in figuring out if any and how much work was assigned.

6. The backend returns to that original user the number of loans assigned and their pipeline reloads with the current view of work table.

7. Now the users that received busy flag will have their browser make another request to recheck the status. That request will only look at the work\_requested table and when it sees that either assigned is greater than zero, or requested is zero it means the long query has completed and the user has either been assigned loans, either there were no loans for the task. The backend will return and the user will know the result. If the query is not complete there will be another retry until the number of retries exceed a maximum.