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

It is inspired from one of my previous projects and a sample description could go like this: Multiple users processing bank loan applications can request to work on various tasks created by managers. Each work request pulls a loan that has the highest priority to be completed for that particular task. Only one user can have a particular loan within a particular task, so let’s say user Oliver does Get Work for 3 loans on a task called “Verify user submitted bank account”, then he’ll get loans that match the criteria “bank statements submitted but not verified” and maybe the 3 loans with the oldest submission time will be assigned to Oliver. He will see the loans on his “pipeline” of work.

In our case the getWork for any of these tasks might be running a very heavy query against the database, which can take anywhere up to 30 seconds or even longer. As mentioned, one loan can be assigned to only one user per task, users can’t step on each other’s toes completing the same task for the same loan. Once the task is completed the user can disposition the loan with either: complete, return to queue after x time with x from 0 to 72 hours, or never return to queue. When the loan is completed or returned to queue after x time expired the loan can be picked up again if it still matches the criteria. So the getWork function should make sure the loan is not in someone’s queue OR if it’s been dispositioned earlier then return after x time expired.

The main issue though arises when multiple users request work for one task at about the same time. In a simple non-clustered implementation users Oliver and Tracy might click getWork 3 seconds apart, the long query runs and both users get the loan #123 assigned for the same task. This scenario would have happened very often in our application and we necessarily needed to avoid it.

If we didn't have a clustered environment then java's locking mechanism might have worked, for example:

1. get a lock on an immutable Integer object for the specific task ID: *synchronize(taskIdInt)*

2. run the query and put the results in a list as a temporary cache

3. assign the loans

4. unlock the object

5. let other threads acquire the lock and assign loans to other threads from the cached list until an expiration timer hits. When expiration timer hits just re-run the query and everything continues from the beginning.

This scenario did not work for us though as we had a clustered environment and each node has its own JVM and doesn’t know about other threads in other nodes.

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.

Below is a concise description of the algorithm we implemented (minus some extra complexities and exact names of tables/fields):

1. When user requests work record user's request in a separate table: work\_requested `[ requesting user id, task id, # of requested loans, # of actually assigned loans]`. This table can later tell us if loans were assigned (e.g req=1, assign=0), if no loans were available (req=0, assign=1) or loans were successfully assigned (req=1, assign=1).

2. Run a query similar to `**SELECT t.\* from task t where t.task\_id=:taskId for update skip locked**`. "for update" part in this query will put a lock on the table row of the task that we are trying to protect from concurrent access. 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, 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 next step of running the heavyweight 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 (req=1, assign=0). These records came while the heavy query was running. We dind’t have to run the heavy query again for concurrent users, we just start assigning loans from the result (i.e. insert records in `work [user\_id, loan\_id, task\_id, …]`). 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 re-check the status. That request will only look at the work\_requested table and based on the result it will either show user work has been assigned (req=2, assig=2), no loans found (req=0, assign=0) or partial work assigned (req=2, assign=2).