Concurrency

Concurrency generally is the programs ability to be executed parallelly. This can be encountered for example on the Internet, where two users use the same website at the same time. Concurrent usage of shared resources can lead to different multi-user problems or conflicts, which we tried to took care of.

Optimistic vs. pessimistic

There are two approaches to resolve concurrency conflicts: optimistic and pessimistic. Pessimistic concurrency assumes that the conflicts are going to happen, and often. We thus lock the database, making sure no one will interfere and create a conflict. This of course means that during this lock period, no one will be able to access the database until it is unlocked, which limits the user greatly.

Optimistic on the other hand assumes that conflicts will not happen. The user is let to do what he desires and if he unknowingly creates conflicts, they are resolved afterwards. It does not limit the users in any way, which means it is the best solution for systems with relatively low possibility of conflicts. Also, compared to the pessimistic approach it is often easier to implement.

Each approach has its shortcomings.

Pessimistic can run into 'deadlocks' - a state when each resource waits on the other to make some progress, waiting indefinitely. This can lead to the system not responding and being stuck at the same point. Optimistic on the other hand can resolve into 'livelocks' - states like deadlocks, but where each resource is constantly changing its state in relation to others, but without any progress. A good example of a real-life livelock would be when someone comes across a person in a narrow corridor. They both politely try to move aside for the other person but end up going back and forth the same way, without any progress.

maybe talk about how they can be resolved???

Transactions

Transactions are a single unit of various tasks/queries that gets executed. Each task gets executed individually and based on the outcomes, the transaction results in a success (every task was successful) or a failure (at least one task failed). A database transaction should always be ACID - atomic, consistent, isolated & durable. They are often the basis for pessimistic concurrency control, although they can be part of optimistic concurrency as well.

At several places in the code, we tend to use transactions instead of plain Dapper queries. The reason is simple - performance. This means that transactions are used not only to batch SQL queries together which either succeed or are rollbacked, but they are also executed faster, whether the query is an insert or an update. The reason why using transactions is more time-efficient is because the transaction is always implicitly created, for say a write operation. This means that if one specifies when the transaction starts and ends, SQL does not have to figure it out on its own.

(pic for insert)

(pic for update)

It must be noted that naturally, each transaction has an overhead, meaning the less transactions, the better. Having a single transaction instead of ten of them saves a significant amount of time. But there is a downside, if one query fails, the entire transaction is rollbacked, meaning we cannot wrap everything into a single transaction. Another important part plays the forementioned isolation levels. When implementing transactions, we had to make sure that they are not wrongly blocking other queries to the database. When it comes to the performance of individual isolation levels there is not much of a difference. (pic)

Our concurrency

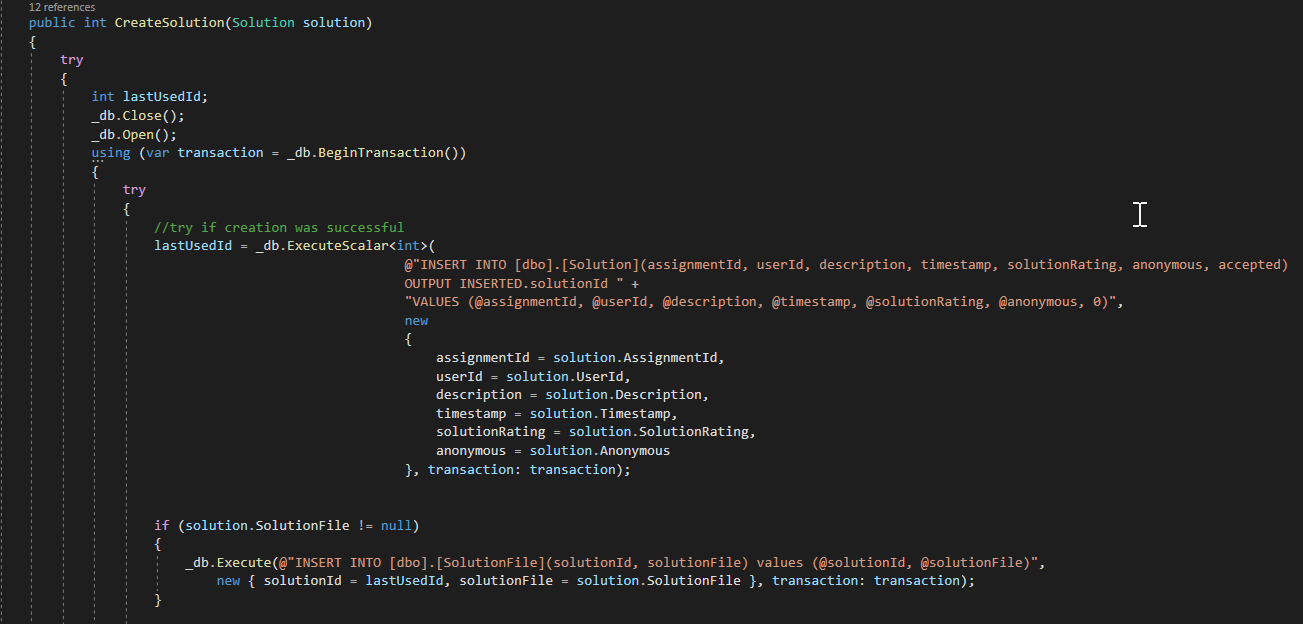
Let us look at how different multi-user issues were solved in our system. For majority of the issues, we chose to use the optimistic concurrency, because it suits our platform the most. There are not that many database updates and we feel like response time is crucial for web solutions so locking the database would at times not be ideal. Of course, if we anticipated that inserting and updating would be crucial for our system, pessimistic solutions would be considered and that is precisely why we use them for specific operations.   
It is important to establish 4 different types of users on our platform:

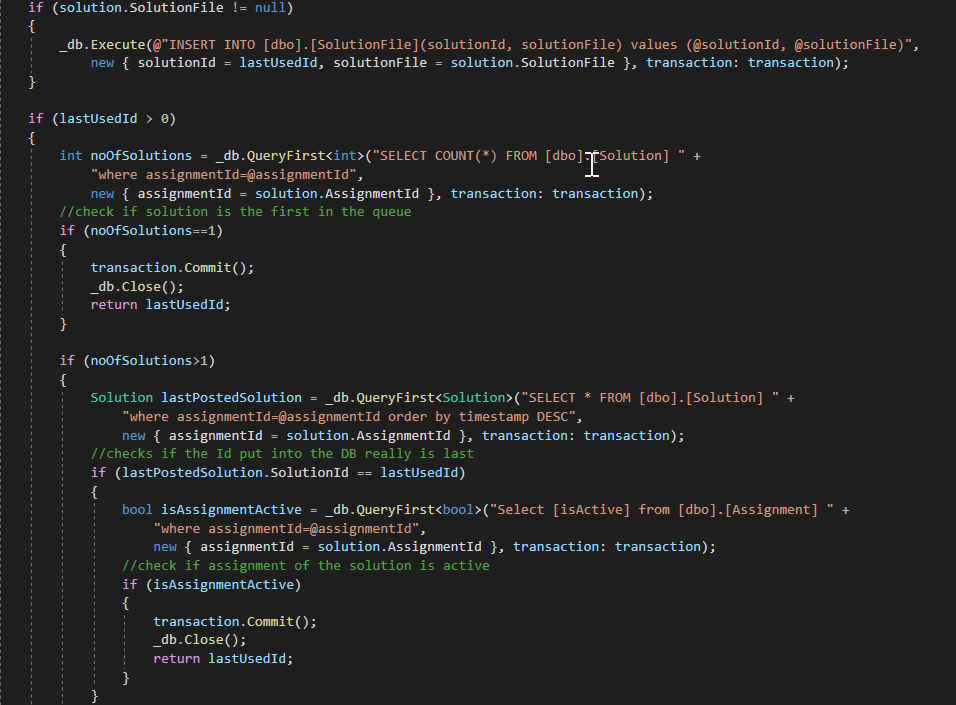
1. a poster - a person who posts an assignment;
2. a solver - a person who provides a solution to the assignment;
3. a basic user - a person who just browses the website;
4. moderator - the only person who would use the dedicated client to moderate the posted assignments/solutions and the forum

The most common multi-user/concurrency issue we came across are:

### **Multiple users trying to post an answer to a solution at the same time**

The solution we have come up with is using a pessimistic solution. When user confirms the solution to an assignment, he locks the database for a brief moment. This would mean that anyone else trying to post a solution at the same time would be prompted to submit his solution again. Like this, there would always be a clear queue of who posted the solution first. Even though this solution locks the database, we do not see a way it would reach a deadlock.

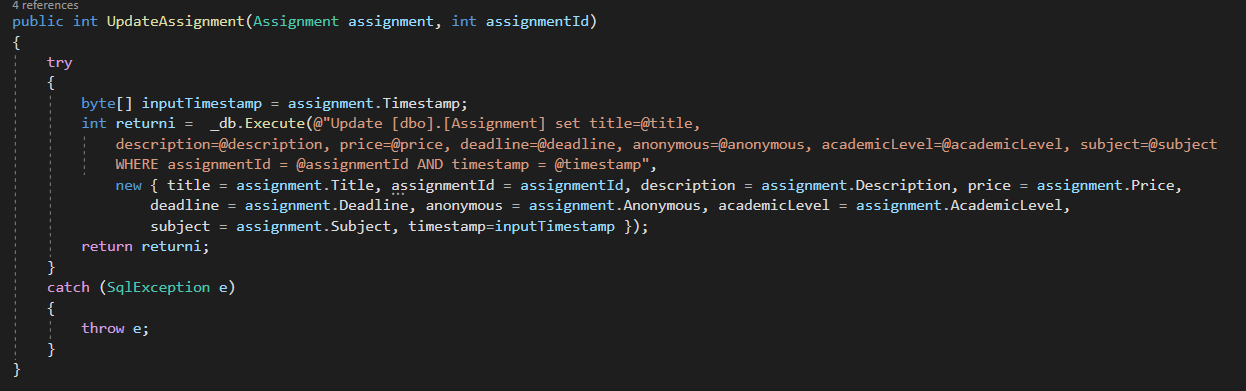




Shouldn’t there be an isolation level??

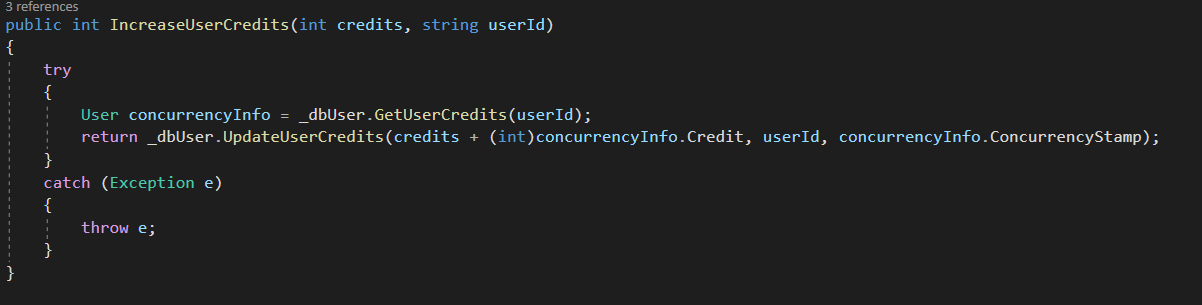
### **User is updating the assignment at the same time as the moderator**

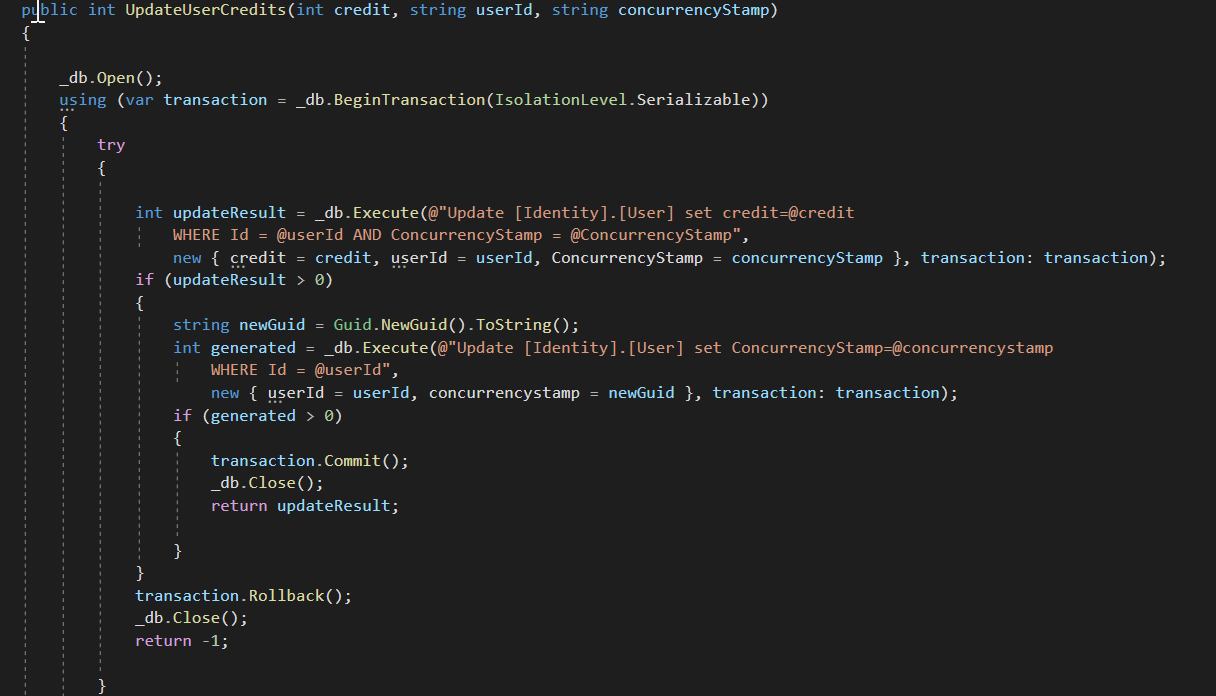
This scenario we decided to solve using a timestamp column of a SQL rowversion type, which is newly generated whenever the assignment in the database is changed. We get the initial timestamp from the Razor page when the user loads it, because that represents the state of information the user is presented with. When he then updates the assignment, we compare this timestamp with the one actually in database. If they match, the update goes through, but if they do not, the user is prompted to try again, so he is aware of the changes made in the meanwhile.



### **User is adding credits to his account and at the same time moderator accepts the credit return request and adds the credits to user’s account**

We decided to solve this problem with a possible lost update again with optimistic concurrency. Every single any credit-related update happens, Users concurrencyStamp is checked. The concurrencyStamp is a GUID (globally unique identifier) that we get before the update to have the latest information. This stamp is compared with the one in the database and if it matches, in a transaction with the update a new concurrencyStamp is generated. If the stamp was changed in the meanwhile, the user is notified to try again.



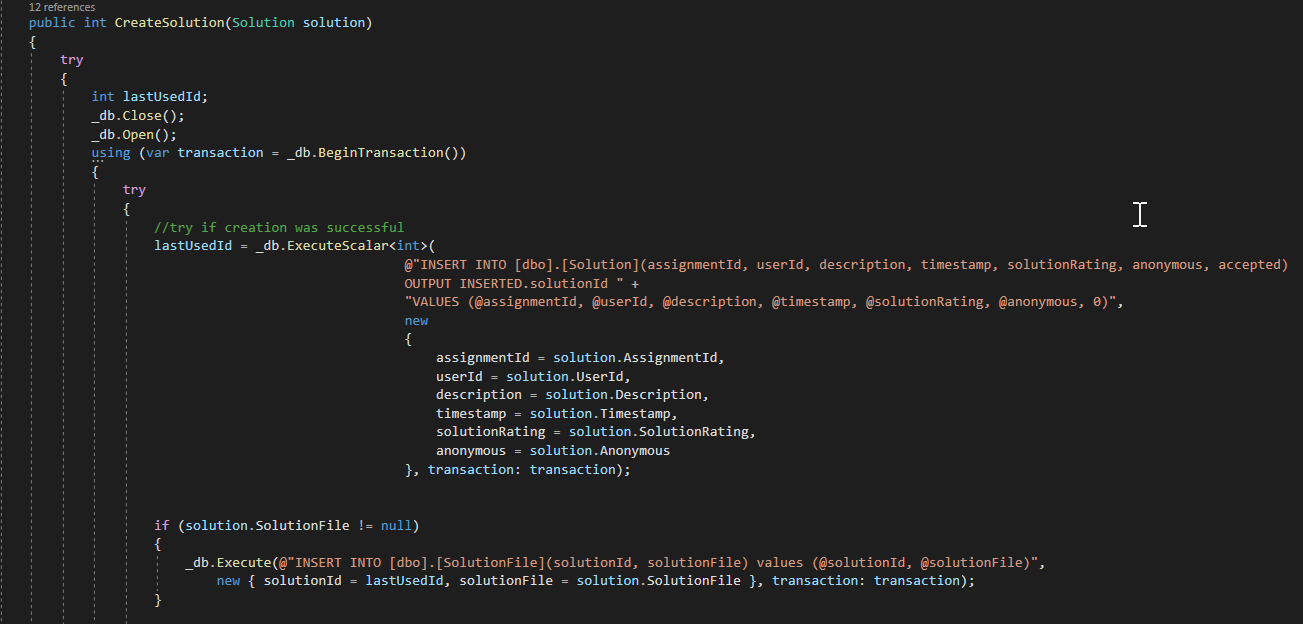


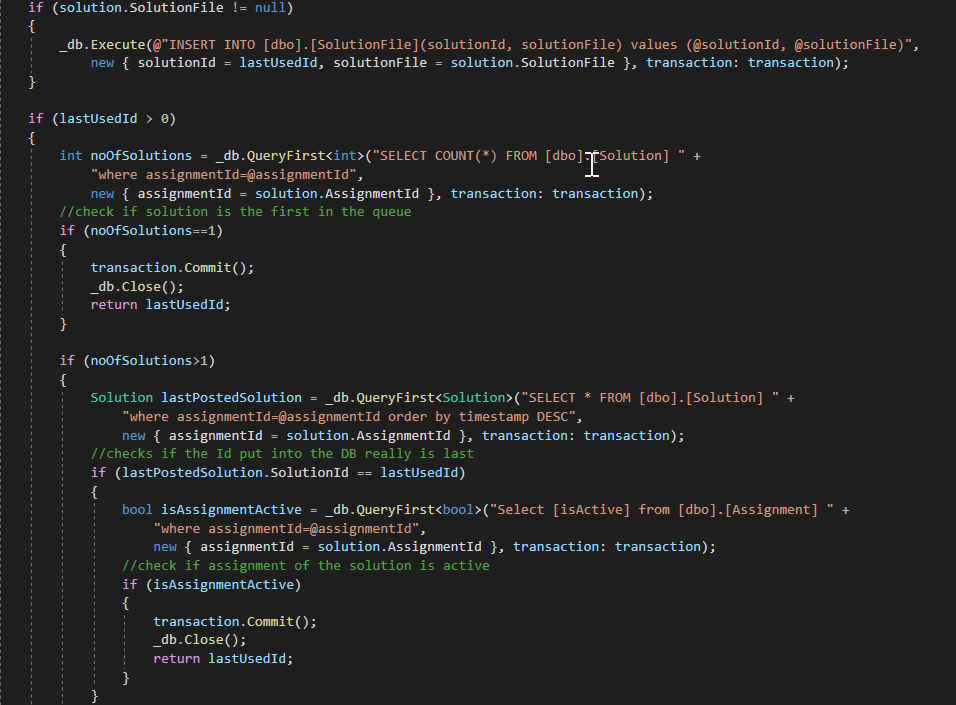
### **Poster deleting a post while the solver is answering it, leaving the solver’s solution “hanging up in the air” as there is no post to associate it with**

As previously mentioned, the creation of solutions is done pessimistically. That means that the operation which starts first sooner will finish before the later is started. Essentially there are two scenarios:

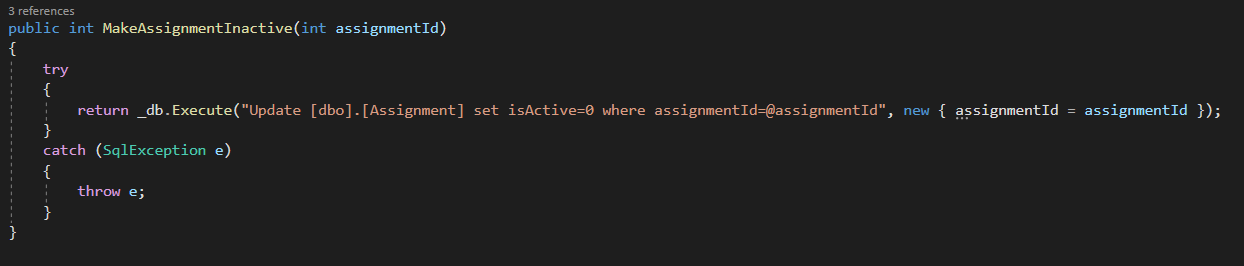
The solver posts a request to the server. The solution has the ID of the assignment attached to it, so the server starts looking for the same ID among the posted assignments. If it finds it, but it is deleted (in reality just disabled, deleted in eyes of the users) or inactive, the solver gets a response from the server, that the assignment has been deleted in the meantime and his solution was therefore not accepted.

If, on the other hand, the server finds an active assignment with a matching assignment ID, it attaches the solution to it, using a transaction for that process), so if there is a delete request from the poster in the meantime, it first lets the solution to be attached and then the post is deleted or disabled. If in fact the post is to be deleted after the transaction that attaches the solution to the post is finished, another transaction is started (for deleting the post) and the post is deleted.





(same pictures as in the first case, maybe redundant???)



### **Poster confirming an answer and paying for a solution while the solver is deleting the solution leaving the poster’s payment “hanging up in the air” as there is no solution to associate it with**

This issue has not been solved yet, although we were awfully close. This scenario which could occur when poster goes through all the solutions, decides to choose a one and pay for it with his credits, while the user who posted the solution decides at the same instance to delete his solution, undesired outcomes could happen.

The solution could be just an extra SQL query in the transaction that would check if indeed the solution the solver receives money for is still active.   
If the solver really would not want for his solution to be accepted, we could solve that by connecting the two parties right after the payment happens, where there would be a chance for them to discuss the solution and solve the possible problems, perhaps by agreeing to a refund.

### **Moderator deleting someone’s post for say abusing platform rules while the solver is answering that post, leaving the solver’s solution “hanging up in the air” as there is no post to associate it with**

The solution for this problem is be the same solution we use for the problem where poster deletes a post that is being answered at the same time, as the logic behind it is the same, just the user who is deleting the post is different (poster vs moderator).

### **Moderator banning a user while some poster is confirming an answer and paying for a solution which was posted by the banned user, leaving the poster’s payment “hanging up in the air” as there is no solution to associate it with**

The solution for this problem could be the same solution we use for the problem where the solver is deleting him solution, as the logic behind it is the same, just that in this case, the solution is not deleted, but the solver is banned by a moderator, which means none of his assignments or solutions can be used.

Unfortunately, we have not found the time to implement a feature which would enable the moderator to ban users.

### **Two users registering with the same credentials at the same exact moment**

We were deciding between optimistic and pessimistic approach. We decided to handle this issue with pessimistic locking by applying Try/Catch construct, it prevents users and applications from editing data that is being or has been changed. Processes know immediately when a locking violation occurs, rather than after the transaction is complete.

**DUNNO WHAT TO WRITE ABOUT this one**