Software Architecture Design Report

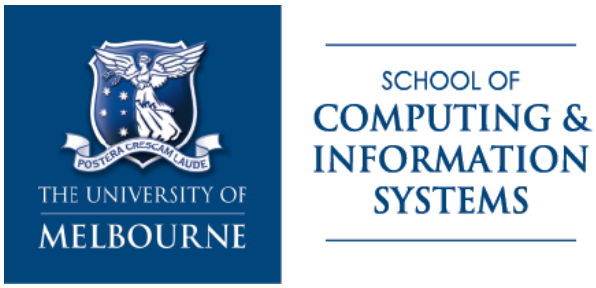
SWEN90007 Semester 2 - Part 3

TomcatBypass

Marketplace System

Team: Team 2

Name	UoM ID	UoM Username	GitHub	Email
James Hollingsworth	915178	jhollingwor	JaeholsUni	jhollingwor@student.unimelb.edu.au
James Vinnicombe	916250	jvinnicombe	jvinn	jvinnicombe@student.unimelb.edu.au
Sable Wang-Wills	832251	lawsonw1	saybb	sable.w@student.unimelb.edu.au
Thomas Woodruff	834481	twoodruff	twoodruff01	twoodruff@student.unimelb.edu.au



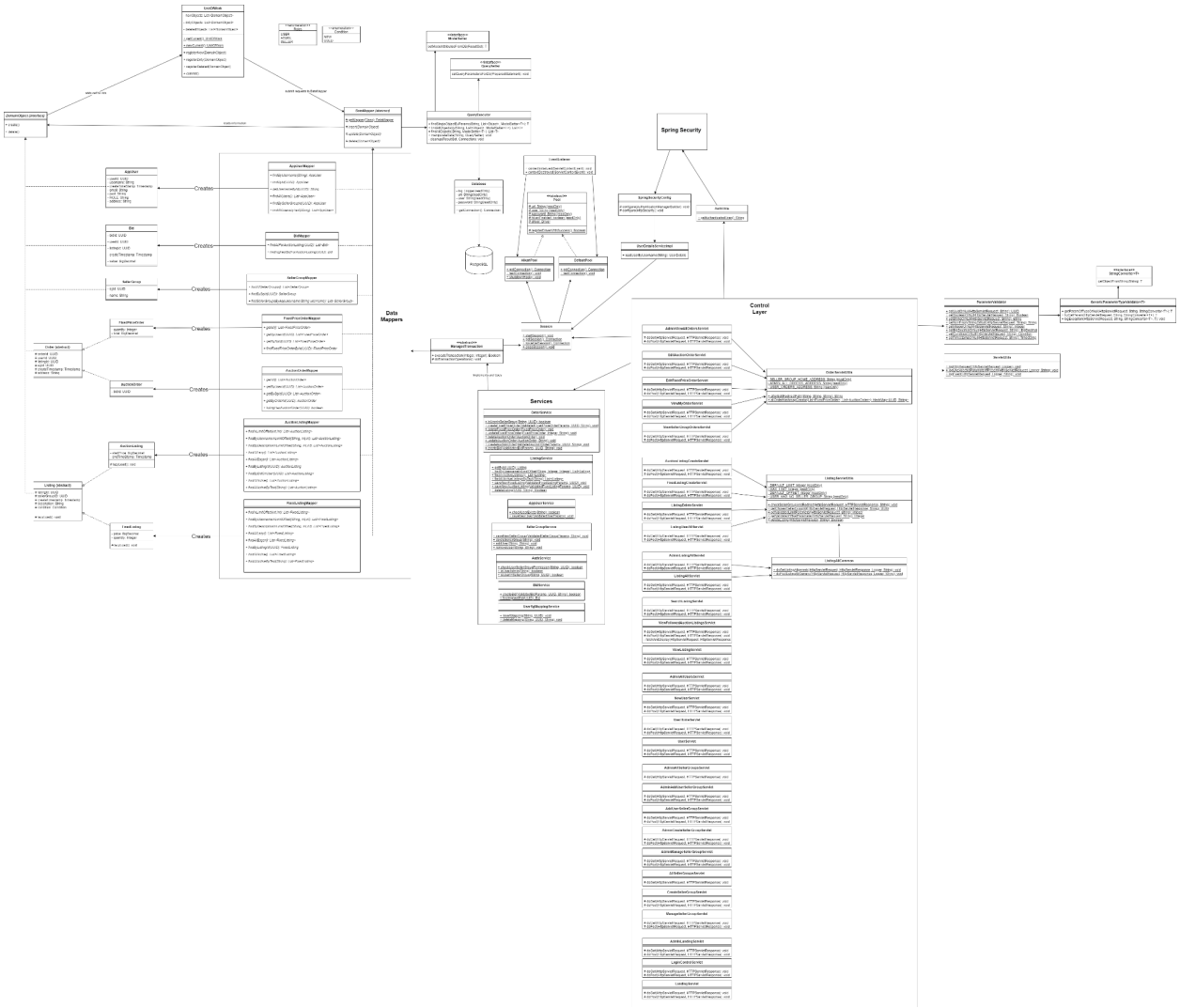
Contents

1. Overview of System Architecture

In this section, the architecture of the system is briefly revisited, as some changes were made.

1.1. Class Diagram

[Full Definition]



1.2. System Layers

This section divides the main components of the system into layers, and roughly describes how they interact for reference.

1.2.1. Database Layer

The Database Layer is responsible for all interactions between the Business Rules Layer and the database. All SQL statements are defined and executed within the Database Layer and results are passed back to the Business Rules Layer.

Mappers

Mappers function by preparing SQL statements and returning model objects back to the Business Rules Layer. Mappers also function in the opposite way allowing classes to pass model objects to the Mapper to be updated or inserted into the database. The Mappers function as end points to be accessed by the Business Rules objects.

Utils

Utils provide database connectivity for the Mappers and transaction management for the Business Rules layer. Classes in Utils manage database connections, and provide an interface for executing transactions on the Database.

1.2.2. Business Rules Layer

The Business Rules Layer is responsible for capturing the structure of system information through models and performing operations on these objects.The Business Rules Layer also interfaces with the Database Layer.

Models

Models act as objects to represent the information that is stored in the system in a logical and useful way.

Services

Services capture business rules as operations that can be performed on domain objects. They facilitate the execution of operations by running them and using the Database Layer to persist their effects. They also act as an interface to retrieve information about domain objects.

Enumeration

Enumerations exist to show representations of strict and limited data fields. These fields are then utilised to show aspects of model objects.

1.2.3. Presentation Layer

The Presentation Layer is responsible for displaying the correct information in the system to the user and ingesting user input. The Presentation Layer also interfaces with the Business Rules Layer.

Controllers

Controllers exist to ingest user input and to ensure that the correct information is displayed to the user. These controllers, implemented as servlets, interact with the Business Rules Layer to procure the correct information and direct the user to the intended Server Page ensuring that the Server Page has the procured information.

Server Pages

Server Pages exist to display the information they receive to the user. Through a combination of markup and scripts in JSP they ensure that the information is displayed correctly and clearly to the user and allow for data ingestion and navigation to be directed towards the correct Controllers.

1.3. Directory Structure

All of the source code is contained within the src folder. This folder contains:

- main: the home of the java implementation of the system
- test: a folder containing the junit and locust tests for the system

Within the main folder there is:

- java: all Java is contained within java/com/unimelb/tomcatbypass and within this there are sub directories for different functional components of the java model
- sql: SQL code for creating the database, and adding in test data are situated here
- webapp: all JSPs are contained within the webapp directory organised by url pathway, in addition to other web resources such as css, images and web-inf which contains the xml and jspf files

1.4. Discussion

1.4.1. Why are we using the UserSgMapping domain object?

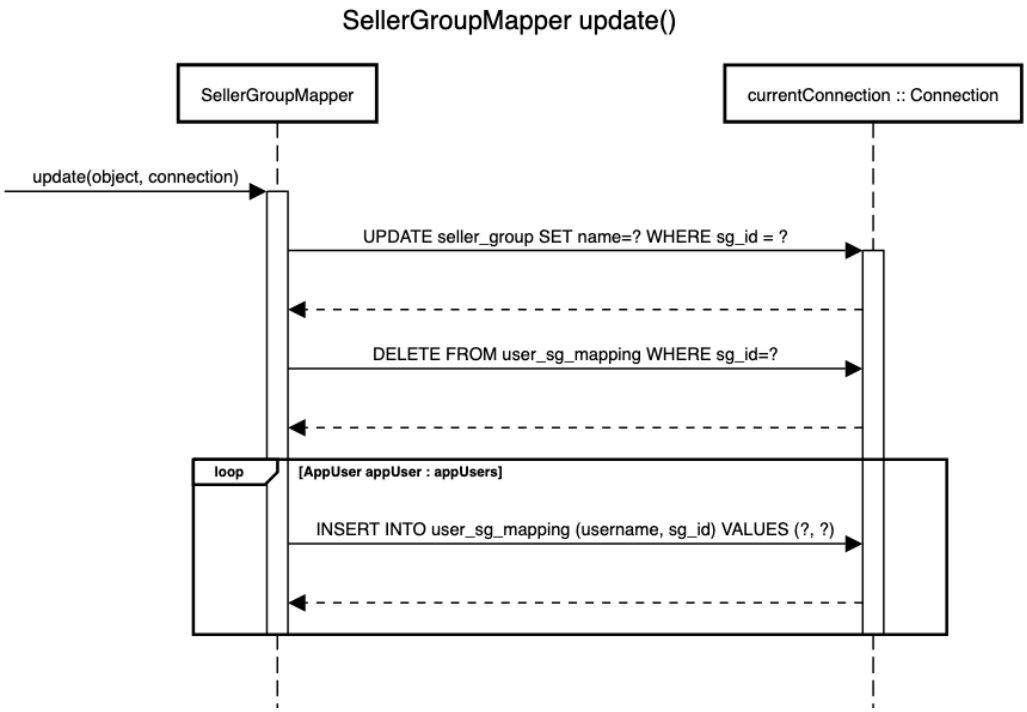
We are aware that adding the UserSgMapping domain object violates the association table mapping pattern and after a discussion with Maria we have decided to keep it. When using the association table mapping pattern there is generally no need to include an in memory object to represent the mapping because the association can easily be represented using references or pointers. Unfortunately in this situation, implementing this pattern greatly increases concurrency complexity and after consulting Maria about this dilemma we have decided to include the UserSgMapping domain object.

Without the UserSgMapping domain object, logic for adding and removing users from seller groups must all be confined to the SellerGroup object's update method. This is due to the Unit of Work pattern including statements for both adding and updating users to be in the same update method for a dirtied object. Simply put, using the UserSgMapping domain object significantly reduces concurrency complexity.

When a user is added to a seller group, in the case where UserSgMapping is not included as an object, we have two options for where to store the relationship in memory.

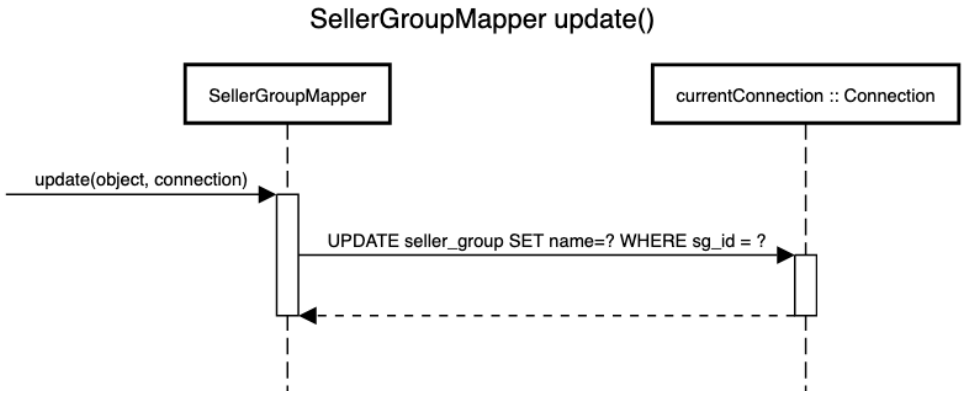
a) For each AppUser domain object, store an ArrayList of SellerGroups that they are part of
b) For each SellerGroup domain object, store an ArrayList of AppUsers that are part of it

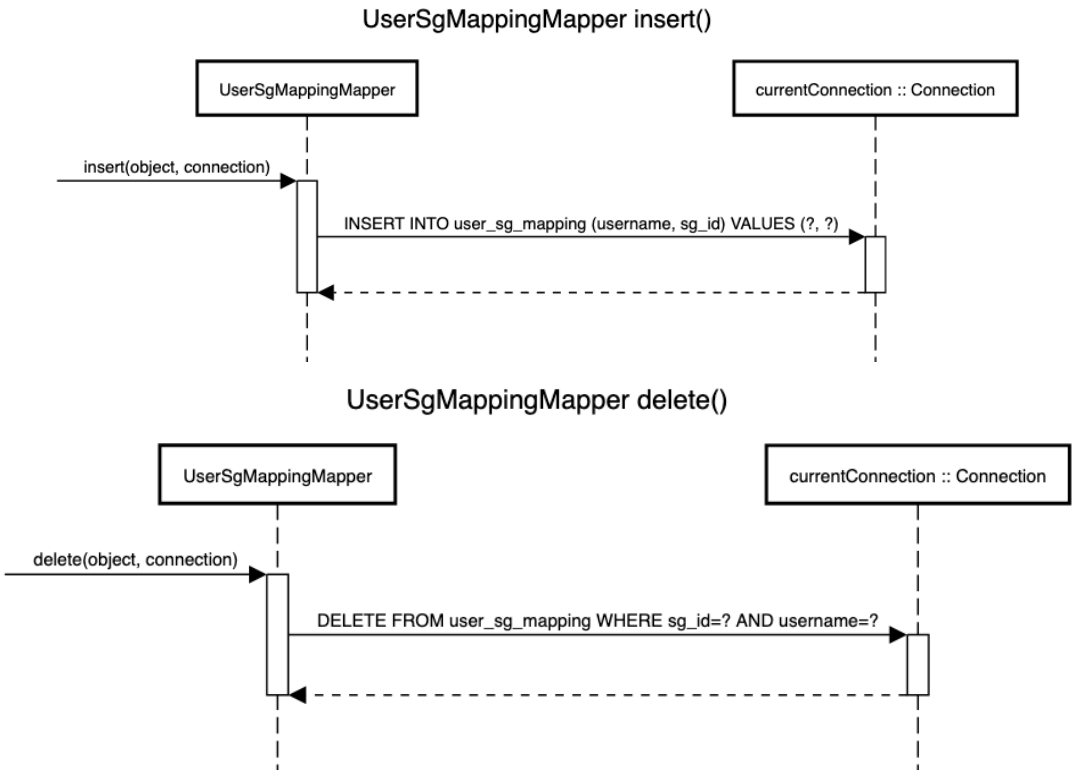
Unfortunately both options conflict with the Unit of Work pattern. The Unit of Work pattern states that if the attributes of an existing domain object are changed, they must be registered as dirty objects. When users are added or removed from seller groups the object which stores the mapping will be registered as dirty (AppUser in option a, SellerGroup in option b). Either way, there must be two different SQL statements that run when registering a dirty object, the first in case the dirty object has mappings removed to delete them from the database, the second in case the dirty object has mappings created to insert them into the database.



This is what updating a SellerGroup using Unit of Work looks like without a UserSgMapping domain object.

When the UserSgMapping domain object is used, the second and third queries don't need to be inside the update method of SellerGroup. This logic can be moved to the insert and delete Unit of Work methods for the UserSgMapping domain object as shown below.





Concurrency is far less complicated when logic for adding and removing users from seller groups is split up like this. With the first implementation where all of the logic is contained in the update method for the SellerGroupMapper, online concurrency control doesn't work because there will be concurrency issues at all isolation levels. Furthermore, using optimistic locking is also not suitable because version numbers can't be tracked when all mappings are entirely deleted then reinserted with the updated mappings. The only solution with the original implementation would be to use pessimistic locking. This transaction affects all database rows associated with the seller group in the user_sg_mapping table, which is impractical because locking modification of seller groups is significantly detrimental to user experience.

2. Concurrency Fundamentals

In this section we briefly define some terms that are used throughout the report.

2.1. Concurrency Issues Considered

The following serialisation anomalies were considered while designing our application.

2.1.1. Dirty Read

PostgreSQL's concurrency architecture renders dirty impossible, however we have discussed the theory here for reference. Dirty reads happen when two or more transactions are occurring concurrently and at least one is reading and at least one is writing. The reading transaction reads from a row that has been written to by the writing transaction that has not been committed yet. The uncommitted data is read by the reading transaction causing a Dirty Read. This is a problem as an uncommitted transaction could be rolled back so the uncommitted data in some contexts should not be read by other transactions.

2.1.2. Inconsistent Read

Inconsistent reads occur when a transaction reads the same data two or more times. During this transaction another transaction makes a commit to the database changing the result of a subsequent read. This results in the reads the transaction makes being inconsistent. Common examples of inconsistent reads include both non-repeatable reads and phantom reads.

Non-Repeatable Read

Non-repeatable reads occur when a transaction reads data multiple times and another concurrent transaction commits to the database. The result can be a transaction reading the same rows from the database more than once with different results. This can cause an issue as the transaction may be anticipating that the data read will be the same as previous reads.

Phantom Read

Phantom reads are similar to non-repeatable reads except functioning over a set of rows. A phantom read occurs when a transaction reads a set of rows based on a condition multiple times within the transaction. Meanwhile a concurrent transaction commits to the database altering the set of rows that satisfy the condition. This results in a subsequent read of these rows by an original transaction containing a different set to a prior read; this is known as a phantom read.

2.1.3. Interference (Lost Update)

A lost update occurs when 2 transactions intend to update the same column and row in a database. They both begin before the update occurs, perform an operation and then write their

new value to the database. However the transaction that writes first will have their data overwritten by the 2nd transaction. This update to the database is considered lost.

2.1.4. Serialisation Anomaly

A serialisation anomaly happens when 2 or more transactions are running concurrently with queries that involve logic across the same set. The issue occurs when committing a group of transactions and the result of those transactions is not the same if they were committed in any order. This anomaly means that depending on the order in which the queries are committed there will be different results in the database. The term ‘serialisation anomaly’ is a catch-all term that covers many concurrency issues within its scope.

2.2. Key Concurrency Definitions

https://www.postgresql.org/docs/current/transaction-iso.html

Some important terms are defined here, as they will be used throughout this report.

2.2.1. System Transaction vs Business Transaction

Business transactions describe a process on the application layer involving business logic. System transactions describe 1 database commit, potentially multiple queries. A business transaction can involve 1 or more system transactions and often involve “user think” time and actions between system transactions if there are multiple.

2.2.2. Isolation Levels

Each use case in our application corresponds to one business transaction with one system transaction. In each case, we have solved the identified concurrency issues solely through the use of online concurrency control. This means our concurrency design, and concomitantly any discussion of it, are entirely reliant on isolation levels.This is why we are identifying each transaction level here, since there would be little to discuss otherwise and they are required knowledge for the remainder of the report.

Read Uncommitted

As far as the SQL standard specification is concerned, read uncommitted specifies that when reads are made during a read uncommitted transaction it will read data from the database given all changes including ones that have not been committed and may be rolled back. This leaves read uncommitted open to Dirty Reads, Non-repeatable Reads, Phantom Reads, and Serialisation Anomalies.

PostgreSQL’s lowest isolation level is read committed, and therefore it does not support the read uncommitted isolation level. For this reason, we have not considered using it.

Read Committed

Read committed is the next highest level of isolation. Read committed requires that all reads read only data that was committed by other transactions before an individual read statement began (usually a SELECT statement). This eliminates the risk of Dirty Reads. Read committed still allows Non-repeatable Reads, Phantom Reads, and Serialisation Anomalies.

Repeatable Read

Repeatable read ensures that when a row is read whenever that row is read again during the transaction it will be the same regardless of if that row has been updated by a concurrent transaction's commit. This does not prevent new rows being added to the database by concurrent transactions. This leaves repeatable read open to: Phantom Reads and Serialisation Anomalies. If repeatable read attempts to modify a row that has been modified since the session began it will have to rollback and retry. Repeatable Read protects against: Dirt Read and Non-repeatable read.

In postgreSQL phantom reads are not permitted. The Repeatable read works with a snapshot of the database taken when the first read occurs making it safe from Phantom Reads.

Serialisable

The strictest level of isolation is serialisable. Serialisable ensures that when a system transaction commits, if concurrent transactions have occurred then no matter the order of all concurrent transactions the resulting commits are the same. If this is not the case, commits will fail until the remaining concurrent commits result in the same database no matter the order of the system transactions. This makes serialisable protected from: Dirty Reads, Non-repeatable Reads, Phantom Reads, and Serialisation anomalies.

3. Managing Concurrency

In this section, we detail how we manage concurrency - what patterns were considered, what was used and how we justified our decisions for each use case.

3.1. Concurrency Patterns Used or Considered

3.1.1. Offline Concurrency Control

Offline concurrency control is a method of implementing concurrency within the application logic of a system rather than through database controls. Offline concurrency can be required for multiple reasons but is primarily used during long business transactions spanning multiple system transactions. This can solve concurrency issues that are beyond the scope of online control through methods such as isolation levels.

Offline concurrency falls into two primary categories: optimistic and pessimistic locking which seek to achieve concurrency control through two different approaches having a varying degree of impact on user experience.

Offline - Optimistic Locking

Optimistic locking allows a user to attempt a transaction that could potentially fail and need to be rolled back. This is done to prioritise user experience. If there is a high likelihood of the transaction succeeding despite concurrent transactions occurring then optimistic locking is the ideal choice of offline locking. A common implementation of optimistic locking is to use versioning on table rows within the database. Each row is assigned a version number and whenever that row is written to, the version number is incremented. When an optimistic system transaction is committed, it confirms that the version number of the row it’s committing is the same as the version it initially read. If this is not the case, the transaction will roll back.

Optimistic locking is ideal for circumstances where a retry of the system transaction is likely to result in success. This provides stronger guarantees compared to serialisable isolation on a database level as it allows the application to check that there has been no changes made to not only rows it may be updating, but also rows it has read since beginning the transaction. This is also particularly important for business transactions that span multiple system transactions where the control cannot be done on a system transaction level.

Optimistic locking can also be appropriate when retries are not likely to result in success if the user input is low and user experience is important. If it is little effort for the user to retry the transaction themselves with new information present then optimistic may still be the correct solution.

In our system we considered implementing optimistic locking control for create, update and delete for fixed price listings. Given the interaction between quantities of fixed price listings and fixed price orders it is important to ensure that lost updates do not occur. Optimistic locking would have worked for this by implementing version control on the listing and order rows and

checking the version of these before committing to the database. However, given the way our application logic handled the calculation of what to write to the listing, using the difference, and that this read was completed in a single system transaction, always updating the listing quantity across all three use cases, online concurrency was also able to handle the issue.

Given that when possible it is safer, simpler and more efficient to use database level concurrency where possible we decided not to include optimistic locking in our system.

Offline - Pessimistic Lock

Pessimistic locking takes a stricter approach to ensuring system concurrency. When a user engages in a business transaction that could potentially change rows in the database these rows have a lock applied to them. No other user is allowed to engage in a business transaction that reads from these rows until the lock is released. The strictness of this depends on the lock type chosen.

Exclusive Write Lock

Exclusive write lock means that only a business transaction that could write to the row is prevented from being engaged while the row is locked. This means that transactions only intending to read from it are permitted. This greatly improves the liveness of the system but opens the risk of inconsistent reads. This additional risk has to be taken into consideration on a case by case basis.

Exclusive Read Lock

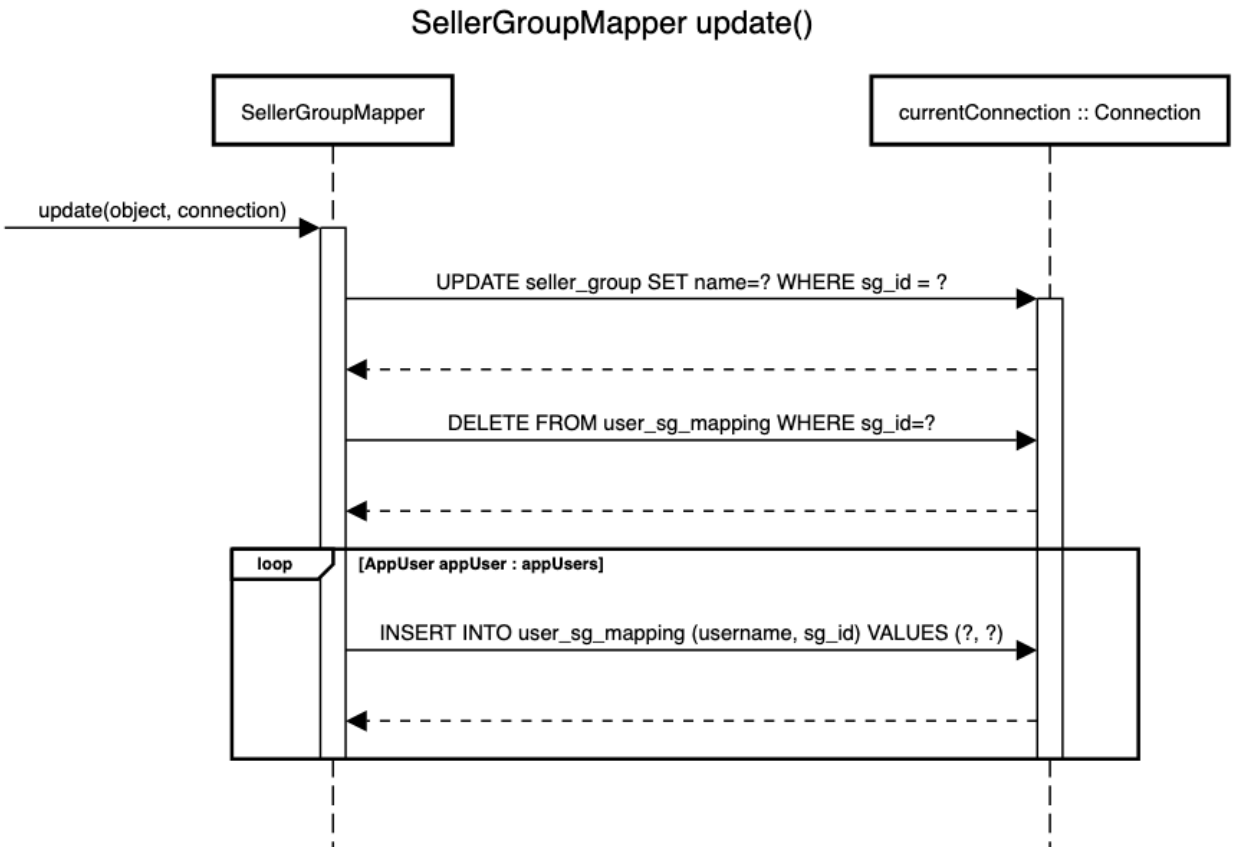
Exclusive read lock means that any business transaction that intends to read from a row must wait until it has been unlocked. This massively reduces the liveness of the application making this wholly unsuitable for common business transactions. However, this protects entirely from inconsistencies in the database.

Read/Write Lock

Read/write lock takes a mixed approach. Similar to a semaphore, there are multiple read locks available and only one write lock available. The read locks and the write lock are kept mutually exclusive so that once the write lock is acquired no one can read until the write lock has been released. This ensures that liveness is improved by allowing multiple transactions to read but consistency is ensured when a transaction takes out the write lock. There are many challenges to implementing this pattern making it only applicable in specific situations.

Pessimistic locks are best used when it is critical that once a user begins a business transaction it be completed successfully. This may be because of important business rules or the result of a long and complex business process. This comes at a significant cost of liveness given that no other users will be able to perform this transaction but this can be required in specific circumstances. However, pessimistic locking can come with a lot of implementation challenges to ensure that the system is still functional in all cases.

In our application we considered implementing pessimistic locking for the use case of adding and removing users from a seller group. In keeping with the Association Table Mapping pattern we intended to have the UserSellerGroupMapping purely be a database object not represented in the application layer. Instead a seller group would contain a list of users that were retrieved via this mapping. The problem arose with concurrent transactions changing the users within a seller group. As we had no UserSellerGroupMapper object the Unit Of Work would consider the seller group dirty. To update this in the database all mapping rows would be removed and replaced with the list in the seller group object. Please see sequence diagram below:



To ensure that updates were not lost from two or more concurrent updates of the same seller group pessimistic locking would be required. Online concurrency controls were not possible given the deletion and creation of new rows would not cause a serialisation error. Likewise deleting and creating rows doesn't allow us to take advantage of version control in Optimistic locking. To stop concurrency issues seller groups would have to be locked pessimistically. We decided against this method of implementation. Seller groups being locked would prevent listings being created under that seller group during any seller group edit. We deemed this to have too large an impact on user experience. Instead we refactored the UserSellerGroupMapping to break the association table mapping pattern in the interest of using more appropriate concurrency controls.

For this reason pessimistic offline concurrency has not been implemented anywhere in our application. Due to its heavy impact on liveness and user experience it is advantageous that our

business transactions did not require it. Forgoing the efficiency of the association table mapping pattern in this instance is worth the tradeoff from a user perspective.

3.1.2. Online Concurrency Control

Online concurrency control relies on the database system to manage concurrent transactions. In SQL this is primarily achieved through isolation levels as described in section 2.2.2. Isolation levels control what kind of data can be read from databases, and can disallow the committing of transactions based on rules as outlined in the SQL standards. Online concurrency is considered to be safer and more efficient than offline as there is less risk for developer error and the control is handled at the lowest layer possible. Online control as implemented in PostgreSQL acts very similarly to optimistic locking in that it allows for potential retries of system transactions, rather than by locking data.

Throughout our application we have entirely used online concurrency control. Given our business rules and application layer logic we determined that it was possible to achieve full concurrency control exclusively through online methods (this is further explained in section 3.3). This was achieved primarily through the use of simple and effective system transactions. Our business rules allowed us to take advantage of the strength of using atomic system transactions meaning that online concurrency could account for all of our issues. Given the preference for online concurrency control where possible we determined that it would be wise to forgo potential offline control solutions and implement isolation level based online locking across our system. This was primarily achieved through patterns found in section 3.2.

3.2. Implementation of Concurrency Control

Concurrency control is coordinated by a number of system components that manage access to the database, the delineation of translation boundaries and a wrapper that coordinates them all.

3.2.1. Session Management

Database connections are managed as Sessions, which are thread-local static records of the current connection being used for database operations. A thread can start a Session, then perform any number of database operations using that Session. This provides a way for threads to execute multiple queries to the database without having to create multiple connections for them, reducing communication overhead and facilitating the implementation of operations as system transactions.

Threads can: (1) start Sessions, which creates a database connection and assigns it as the current session - any pre-existing sessions are closed; (2) get Sessions, which gets a Session if there is one active so that the connection can be used; and (3) close Sessions, releasing the connection and making it inaccessible to future getSession calls. See the corresponding figure for sequence diagrams.

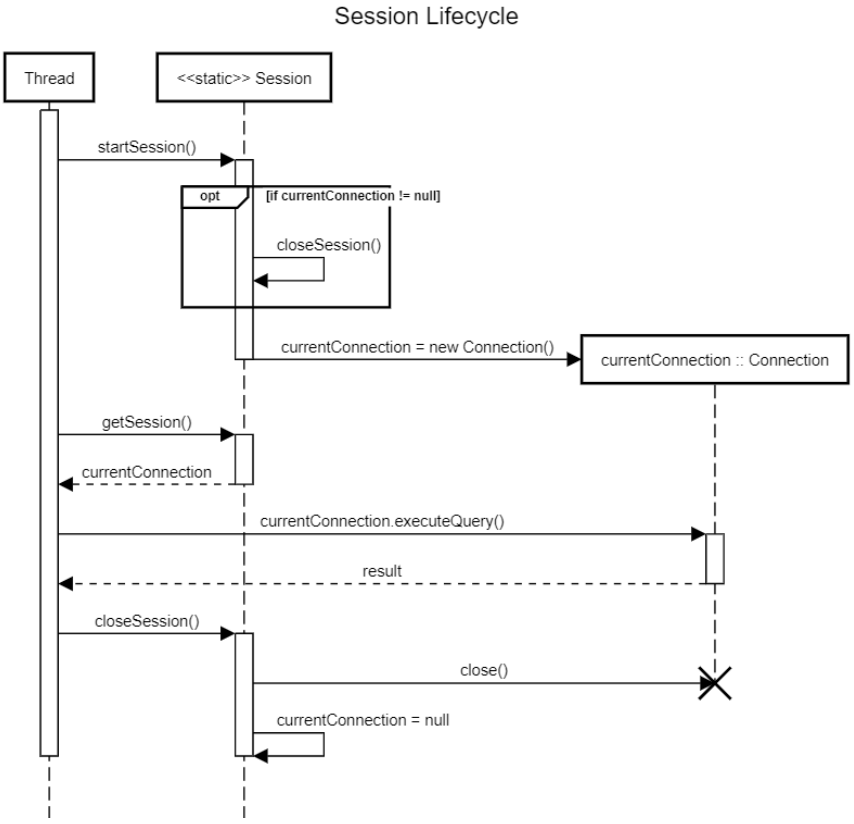


Figure: Sequence Diagram depicting a Session's lifecycle

Every database read and write goes through a Session, so Sessions are started by service layer operations for transaction execution and by control layer Servlets to make reads more efficient. For example, our QueryExecutor fetches the current session for reads and writes (see example below).

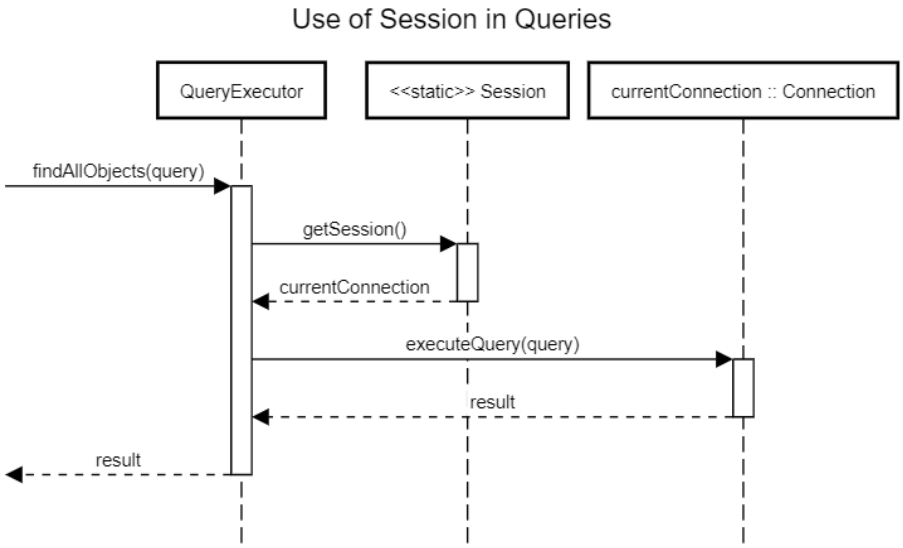


Figure: Sequence Diagram depicting a simplified example of a QueryExecutor using Session

3.2.2. Unit of Work

The Unit of Work pattern is implemented to manage the atomicity of transactions. With a Unit of Work, any changes to domain objects are recorded by it and are collected and persisted on the database as a single system transaction when told to commit. Unit of Work is used for every write operation to the database - all transactions in the system use a Unit of Work object. Changes made to objects are registered in the thread-local UnitOfWork instance:

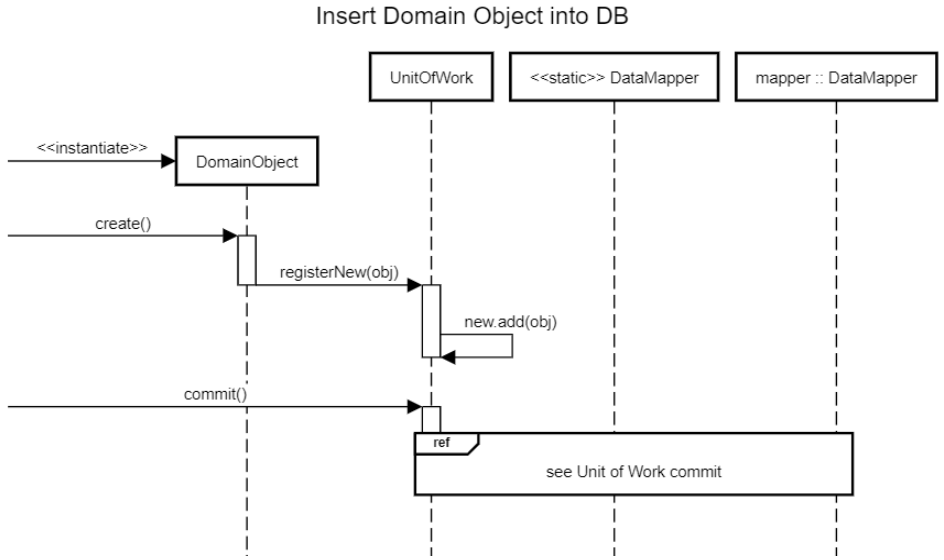


Figure: Sequence Diagram depicting an object being inserted into the database

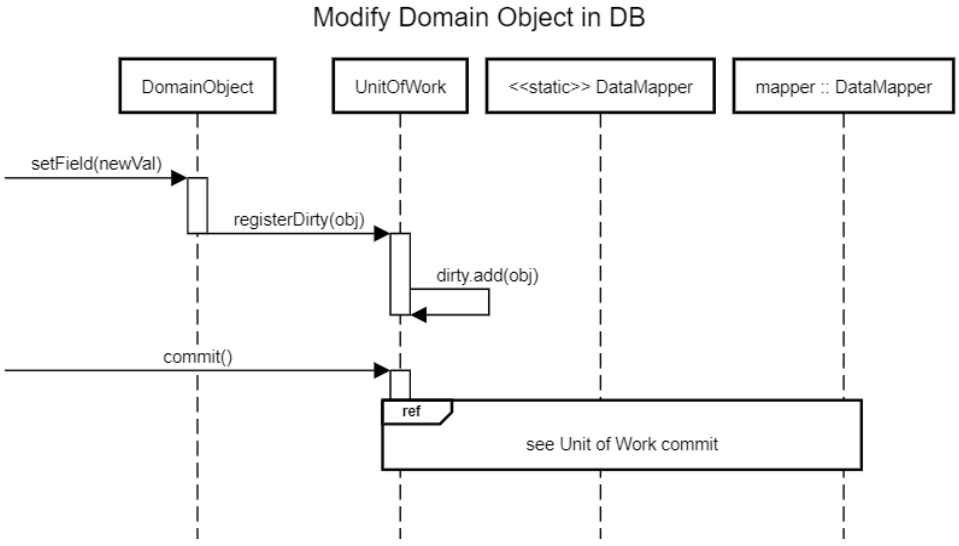


Figure: Sequence Diagram depicting an object being updated

Similarly, Unit of Work must utilise the connection from the current Session in order to execute changes on the database when it commits.

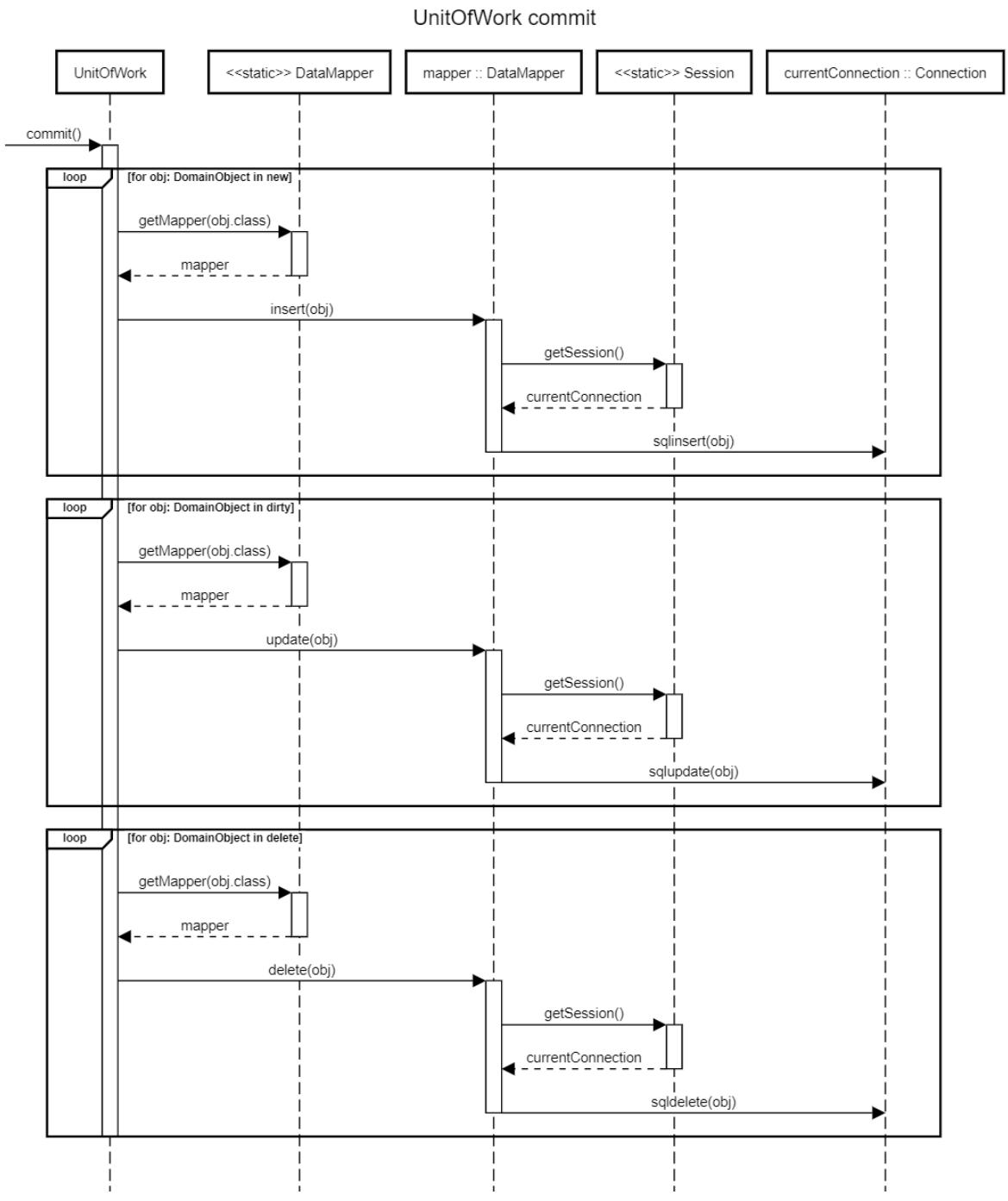


Figure: Sequence Diagram depicting a UnitOfWork committing

3.2.3. ManagedTransactions

The way Unit of Work and Sessions were implemented led to a lot of boilerplate code - a service method would have to start a new session, configure the connection to meet concurrency requirements, do reads, start a Unit of Work transaction, make changes and then commit, handle rollbacks, handle errors and finally, close the session. The majority of these operations are common across all system transactions, so we implemented a ManagedTransaction class to handle all this under the hood, and to delineate transaction boundaries.

A method would just have to create a ManagedTransaction class overriding *doTransactionOperations()* with the action it needs to perform as part of the transaction, and call *executeTransaction()* on it, which returns a boolean value indicating transaction success or failure. See the sequence diagram below to see how it works.

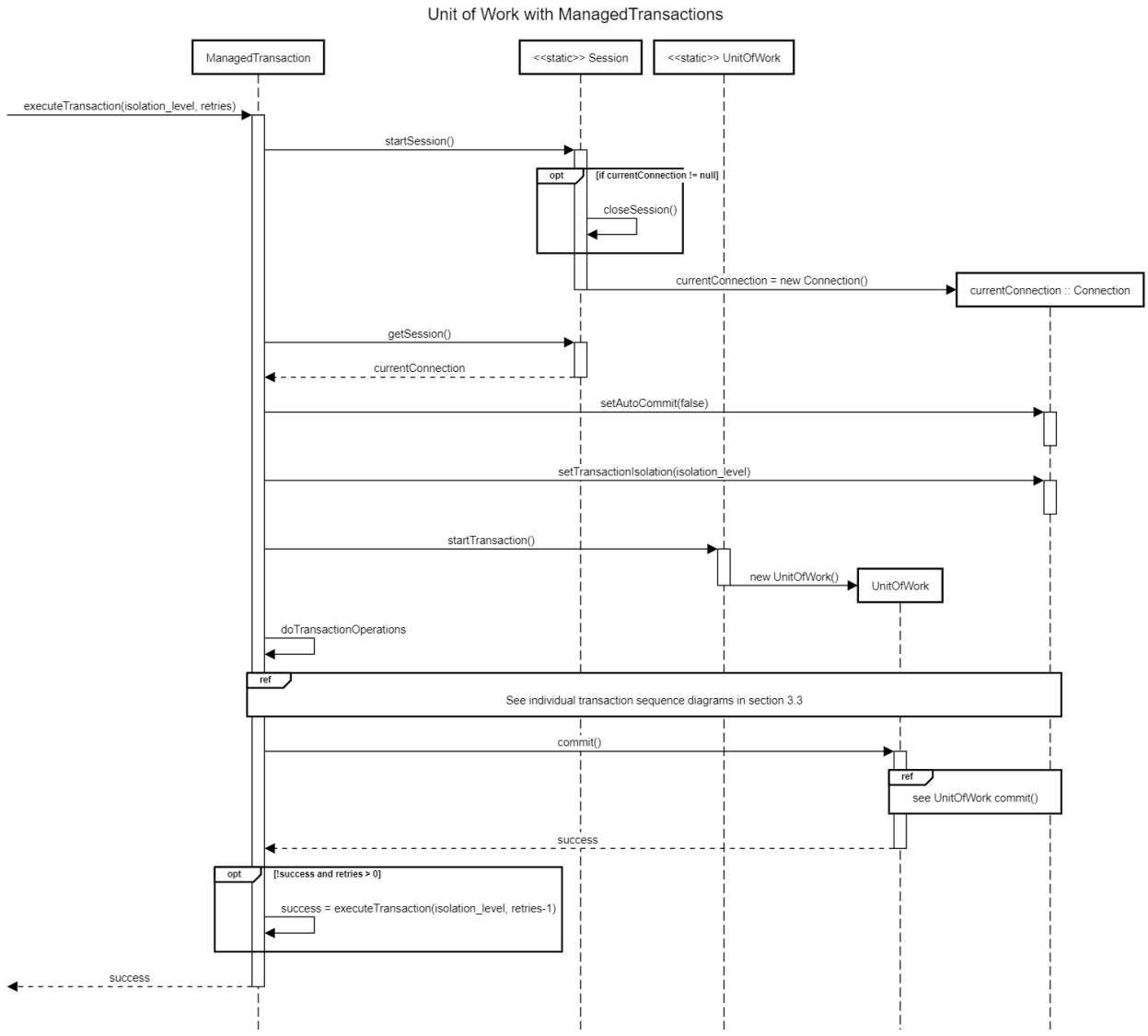


Figure: Sequence Diagram depicting execution of a Managed Transaction

Different use cases require different isolation levels, retry limits and transaction operations, so these are designated by the service that implements the transaction through its call to *executeTransaction*. See section 3.3 for how each transaction handles this.

3.2.4. System Transactions using Isolation

Using our implemented ManagedTransactions, by selecting an appropriate level of isolation, we can implement sequences of database operations as system transactions. The ManagedTransaction acts as a delineating bound for a system transaction.

Additionally, the use of UUIDs for the primary keys of the Database enables a reliance on foreign key constraints to cause some invalid transaction states to roll back. For example, if operation to create an Auction Order is interleaved with an operation to delete the Auction Listing that Auction Order was created for, and the Listing is deleted in between checking for the existence of the Listing and the committing of the Auction Order, when the create Auction Order operation attempts to commit, the database will detect that there is no Listing corresponding to the Order to be created and rollback the transaction. This is reliable as UUIDs are effectively guaranteed to be unique, so there is no chance that the Listing is deleted and a new Listing is created with the same UUID, making the Order for the wrong Listing. This property of UUIDs greatly simplifies some of the interactions between delete operations and other operations.

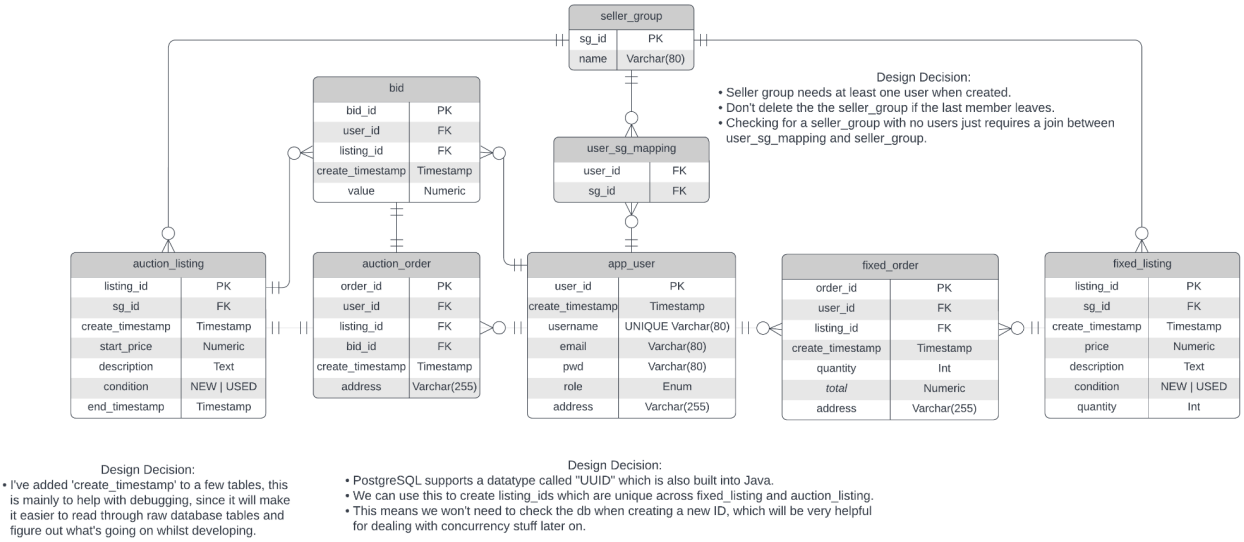
3.3. Identified Concurrency Issues

In this section, we cover potential concurrency issues in the marketplace system, organised by operation. Naturally, only operations that make changes to the database are considered.

3.3.1. Database Diagram

The database diagram is displayed here for referential purposes while reading the justifications throughout section 3.3. Many concurrency issues are results of the interaction between multiple database tables, or are relevant to specific database columns. All database columns and relations are shown below.

[Full definition]



3.3.2. Creating New User

Isolation Level: Read Committed

Description

When creating a new user in the system there is a single business rule to consider

- No two users of the same name can exist

Usernames uniquely identify each user and if 2 or more users are attempting to create a new account with the same username this could result in a concurrency issue.

Rationale

The non-concurrent check to ensure that this does not eventuate is twofold. Firstly when creating a user the service method searches the database for users existing with that chosen name, if a user is found the service method will throw an exception and reject the change. Secondly the username acts as a primary key for the AppUser table, this means any attempt to insert a new user with a taken name will also result in an exception being thrown.

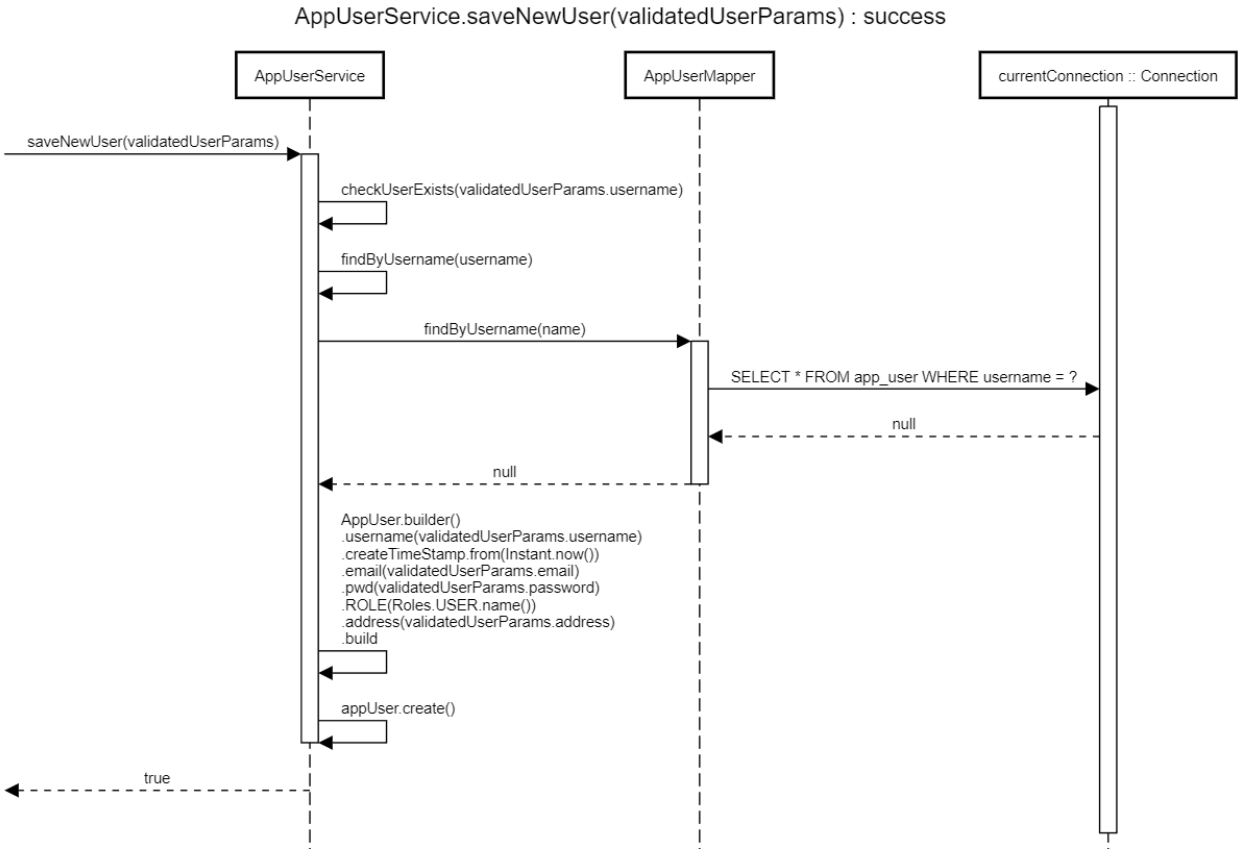
Given that creating a new user does not modify a row within the table, offline concurrency controls are not appropriate as there is nothing to lock, nor version control for a non-existent row. For this reason we have determined that online concurrency control will be optimal.

Implementation

Using our ManagedTransaction class we have implemented creating a new user by setting Transaction Isolation level to read committed. Read committed was selected as when checking if there is a user of the same name there is no need to control for repeatable read. Despite the opportunity for a user to take the username during the transaction, username is a primary key so an exception will be thrown. There is no need to attempt a retry of the query as it would only succeed if the table row for the user was deleted which is not possible in the system.

Diagrams

Sequence Diagram for saving a new user: This shows the double protection of checking if the user exists and usage of the username as a primary key.

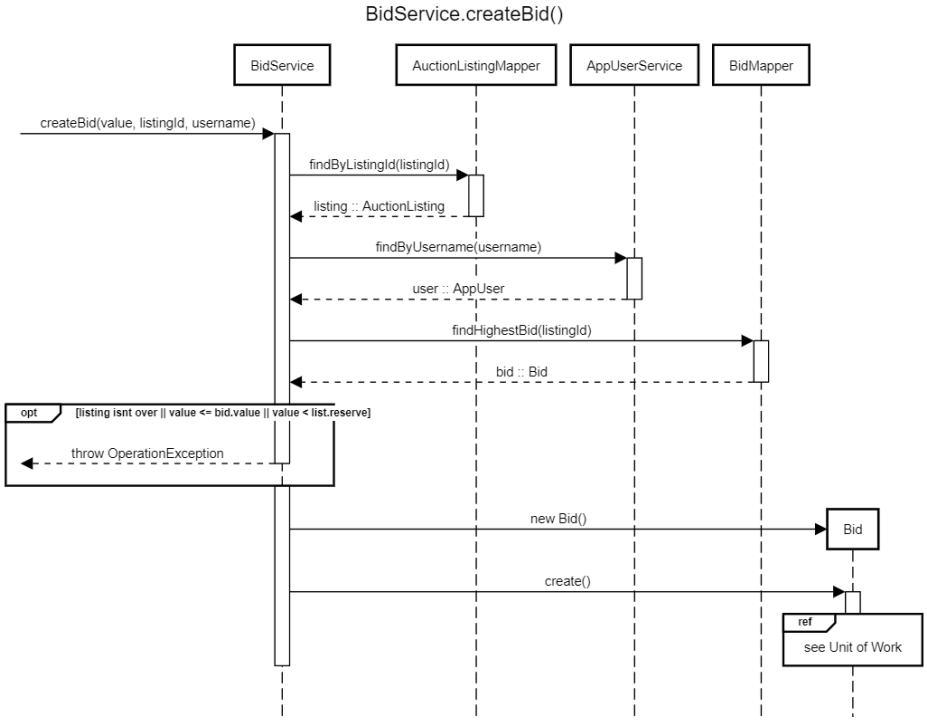


3.3.3. Creating a Bid

Isolation Level: Read Committed

Description

Users can create Bids for Auction Listings, as long as the Auction's end time has not elapsed and the Bid's value is greater than the Auction's reserve price. While it is relevant for the Bid's value to be higher than the highest Bid, violating this will not cause any issues with the outcome of an Auction.



Sequence Diagram - overview of doTransactionOperation for creating a bid

Rationale

The concurrency requirements imposed by the specification of this operation are not strict, as both of the important checks are against fields in the Auction Listing, which are static as listings cannot be edited. If they are valid when checked then they are going to be valid at commit time. As such, this only necessitates that valid, committed data is to be read.

Another soft requirement is that a new Bid should have a higher value than the current winning Bid, but not strictly validating this does not affect the outcome of an auction, as even if two Bids are created with the same value, the earliest Bid still wins. Validating this as part of the read portion of the transaction does, however, reduce the number of irrelevant Bids created in the system, so the check can still be included.

As such, there is no real need to implement concurrency control for this feature, so we can default to a single system transaction to handle it.

Implementation

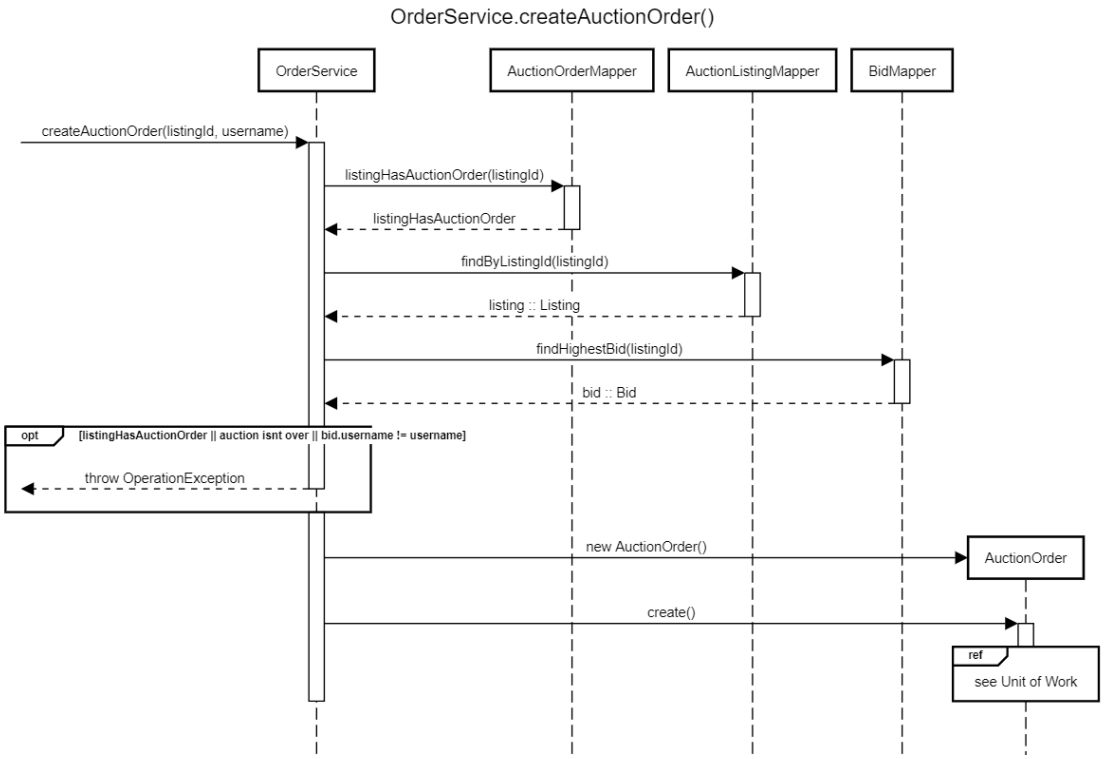
This was implemented as a ManagedTransaction, with isolation level read_committed and no retries, as read_committed does not cause rollbacks.

The chosen isolation level is the weakest, and ensures that data read is not uncommitted. Each row is only read once, so there is no need for any stricter requirements.

3.3.4. Create Auction Order

Isolation Level: Serializable

Description



Sequence Diagram - overview of doTransactionOperation for creating an auction order

Users that have won an auction can create an order for said auction - that is to say, if a User has the highest bid for an auction which has ended, they can create at most one order for that auction. The prerequisites for creating an auction order are summarised as follows:

- There are no existing Auction Orders for the Auction
- The highest value valid Bid for the Auction is owned by the User
- The Auction's end timestamp has elapsed

The main precondition that can cause concurrency issues is that there can only be one Auction Order for an Auction, as the other conditions are stable.

Rationale

Two of the preconditions are stable - once they hold, they should never change in a way that they no longer hold. Once they have been verified as true, they are going to remain true no matter what transactions are executed, with the exception of a listing deletion, which would cause the transaction to fail on commit due to foreign key constraints - an auction order requires

the listing ID provided to match a valid auction listing. As such, there are no concurrency issues with these.

Checking whether an Auction Order exists does necessitate some concurrency control, however, as two transactions executing to create an Auction Order for a single Auction Listing need to result in only one Auction Order being created. A check needs to be performed before commit, and then that needs to remain true at commit. While this could be handled by pessimistic locking of the auction order table, the only way this issue can occur is if the same user makes an Auction Order twice for the same listing at the same time, an occurrence that is extremely unlikely. That means this online optimistic approach will result in less downtime than pessimistic strategies might cause with low chance to waste work due to the rare nature of the conflict actually occurring.

It is clear then, that the transaction can be handled with a single system transaction, as there are only simple preconditions that need to be checked on execution in order to satisfy its validity. The remaining problem, then, is to choose an appropriate isolation level.

Implementation

This was implemented using a ManagedTransaction, with isolation level serialisable and 5 retries. Auction Order creation transactions are the only ones that will cause serialisation issues, and those are relatively uncommon in the system, so just retrying the transaction is very likely to result in a valid execution - i.e. we fail the precondition or we commit.

Checking whether there is an existing Auction Order for an Auction Listing is a complex join query whose result is dependent on the number of rows returned, rather than the contents of said rows. As such, Phantom Reads are an issue, meaning that the Serialisable isolation level has to be chosen.

3.3.5. Update / Delete Auction Order

Isolation Level: Repeatable Read

Description

When updating or deleting an auction order there is a single business rule to consider:

- The user is authorised to delete the order
    - The user is an administrator
    - The user is the associated application user of the order
    - The user is part of the seller group associated with the order

This could change because an administrator removed the user from a seller group they are part of during the update or delete transaction. This can cause a concurrency issue.

Rationale

To determine if a user is permitted to update or delete the auction order a service method is called that validates the user's access to the auction order. Multiple database reads are made during this method to verify that the user either created the order, is an administrator or is part of the seller group.

Offline concurrency controls aren't a suitable solution for this problem. If we have to use pessimistic locks on all rows touched during the checking process this would damage the performance and user experience as the effect would be far reaching. Whenever a user attempts to update their order then no user would be able to edit the seller group associated with that order. As we presume changing the address of an order will be a relatively common occurrence we cannot lock the seller group during this time.

Likewise using optimistic concurrency checks would work but could require an inordinate amount of retries since updating a seller group version doesn't necessarily mean that the specific user in question was removed.

Given these factors we determined that online concurrency control through isolation levels would be optimal.

Implementation

Using our ManagedTransaction class we decided to implement the isolation level of repeatable read with 2 retry attempts. Repeatable read was chosen for the following reason.

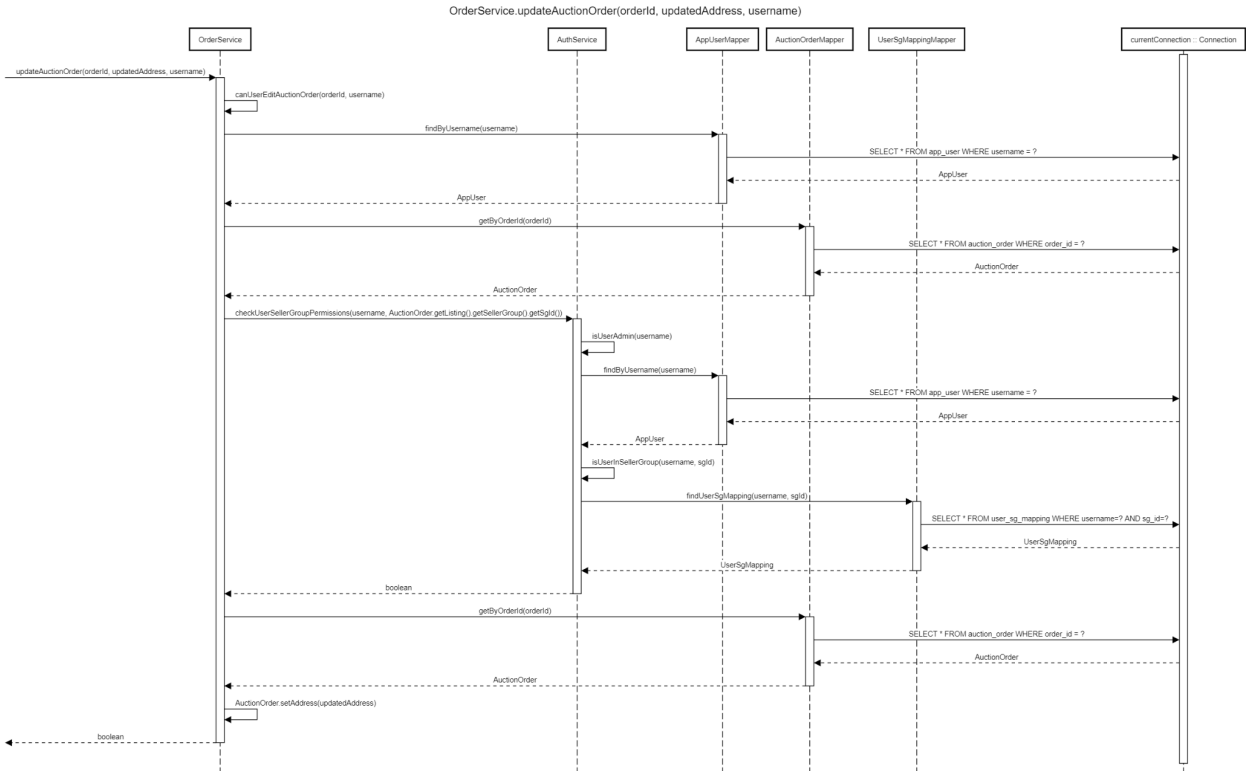
Our application business rules state that for the purposes of authorisation if the user is authorised when they begin their request to edit then they are considered authorised for the request. This means that a user becoming unauthorised during the system transaction should not cause the transaction to fail.

Repeatable reads take a snapshot of the database after the first read and will assess the user's validity on this snapshot. This means that if the seller group is changed during the processing of this transaction the resulting commits will not affect the outcome of the transaction. Retries have been added for the case where the auction order may have been edited, this would cause the transaction to have to try again.

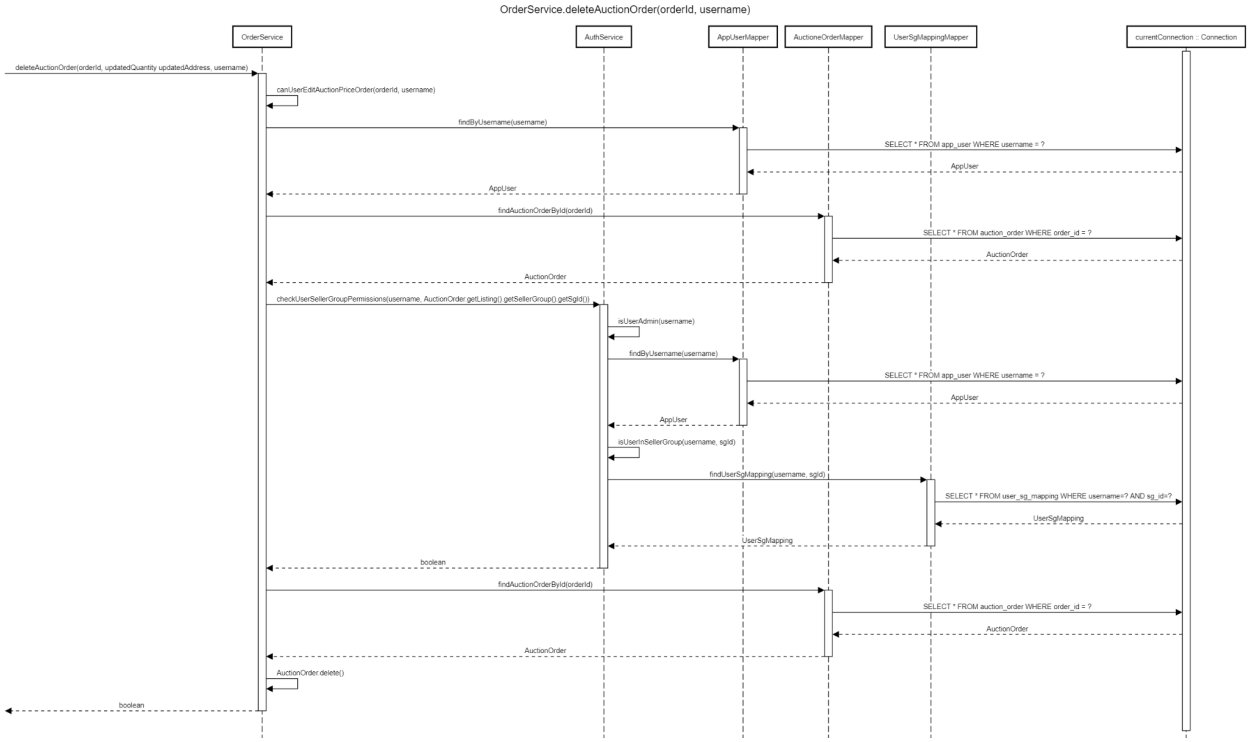
The retry attempt would write over the previously written data causing a theoretical lost update. However, given for updating an auction order the only field is the address we consider this acceptable. Whatever the most recently committed address should be used as the delivery address. This should not be a common occurrence as it would be strange for multiple people to try and change the address of an order to different addresses given only a seller group member or the relevant user will be able to.

Diagram

Sequence diagram for updating an auction order: Shown here is the check to ensure that the user is authorised and the subsequent unit of work implementation of updating



Sequence diagram for deleting an auction order: Shown here is the check to ensure that the user is authorised and the subsequent unit of work implementation of deletion



3.3.6. Edit / Create / Delete Fixed Price Order

Isolation Level: Serializable

Description

When editing, deleting, and creating a fixed price order there are a few business rules to consider.

- The order quantity is always 1 or greater
- Quantity of orders for a listing + quantity of the listing will always total the original quantity of the listing
- The listing quantity is always 0 or greater

Editing and deleting a fixed price order also has the additional business rule.

- The user is authorised to edit / delete the order
    - The user is an administrator
    - The user is the associated application user of the order
    - The user is part of the seller group associated with the order

The user's authorisation could change during the transaction due to an administrator removing the user from a seller group. Likewise it is possible for both the order quantity and the fixed price quantity to change during the transaction. This could result in concurrency issues as multiple transactions updating quantities could result in a miscalculation by the application layer causing incorrect updates to the quantities for the order and listing.

Rationale

The rationale for validating a user is able to edit the order is the same as above for 3.3.5 Update / Delete Auction Order.

To ensure that the order quantity and listing quantity is never updated to be an invalid number we can add an additional condition into the UPDATE SQL statement. This guarantees that no update will ever occur making order quantity less than 1 or listing quantity less than 0.

However this does not stop the problem of 2 valid updates going through concurrently and one of them being lost due to the latter overwriting the former.

Pessimistic offline concurrency control would account for this locking the listing and the order while someone is editing or cancelling. However this would prevent people from purchasing from the listing while someone is editing or cancelling an order relevant to that listing. This is not suitable given the impact on the user experience. We expect that editing or cancelling an order will be a common occurrence especially for a popular listing. Given this occurrence rate it would be infeasible to stop users being able to buy from the listing or any other user editing their order while a single user edits

Optimistic offline concurrency control could also be appropriate for preventing this issue. However, given that the update takes place within a single system transaction there is no need to record the version throughout the business transaction. If we decided to use versioning it would only increase database and application logic complexity.

For these reasons we have determined that online concurrency control is the correct solution.

Implementation

Given that the process of editing, cancelling or purchasing a fixed price order has the user determine what quantity they require (in the case of cancelling this amounts to 0) then makes the updates within a single system transaction using online concurrency is ideal.

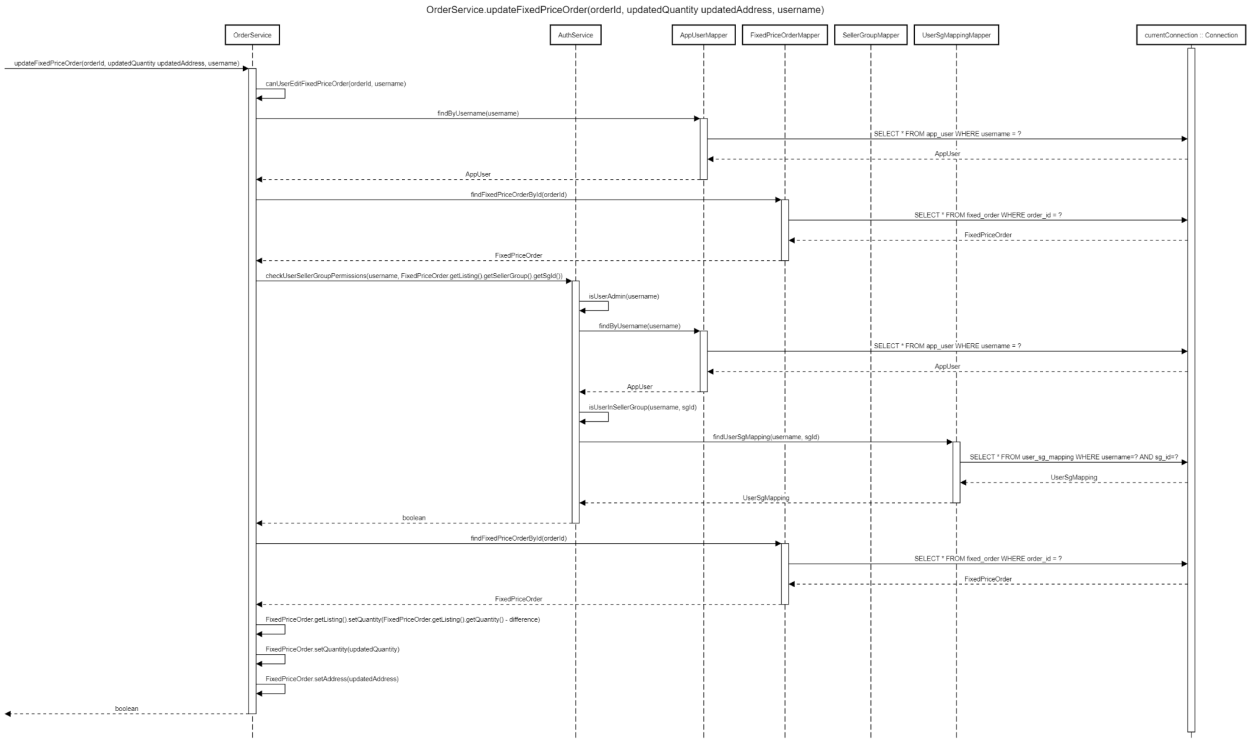
We set the isolation level to serializable using our managed transaction class. This class determines how the fixed listing quantity should be adjusted given the new required quantity of the order. Because of the isolation level, if after the transaction takes its first read (taking a snapshot of the database) either of the rows of the order or the listing have been updated the transaction will fail due to a serialisation error.

All 3 transactions that can change the quantity of a fixed listing or of a fixed order change the quantity of the fixed listing. This ensures that if any of them occur concurrently the transaction will roll back.

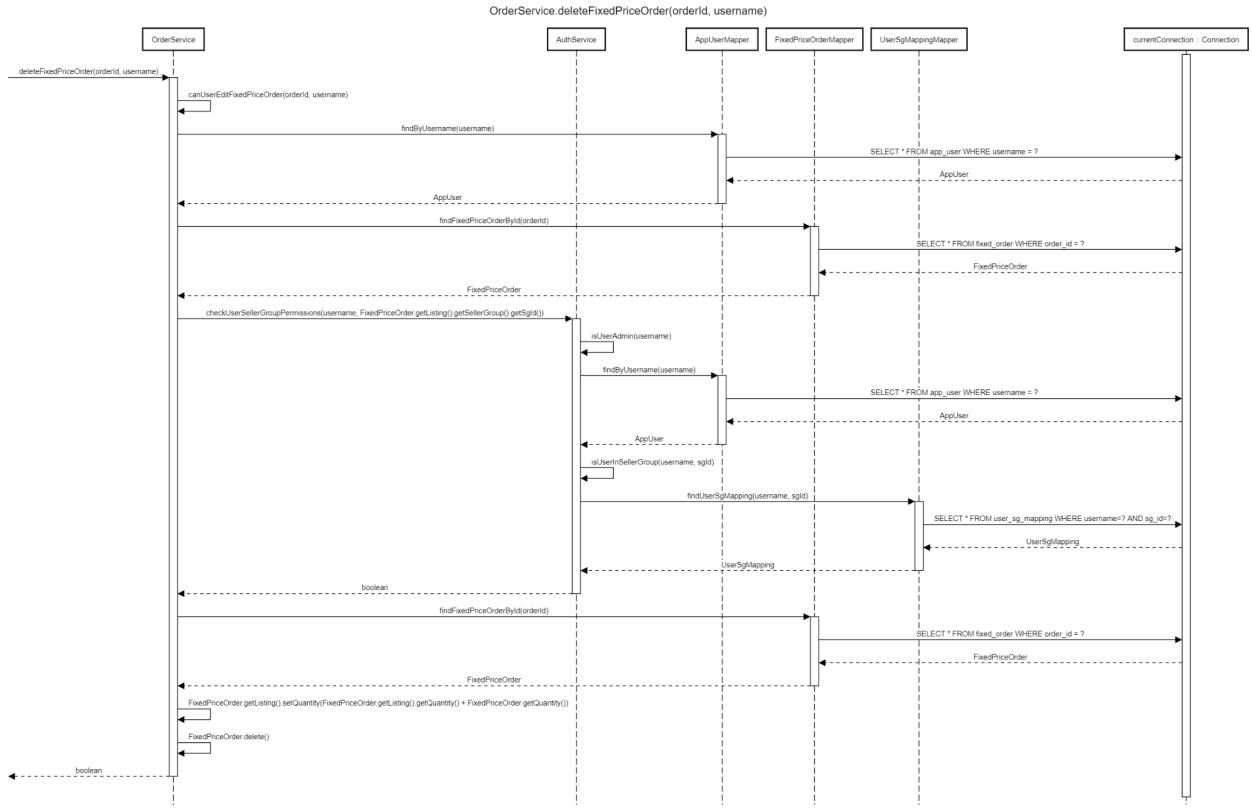
The reason we have allowed 5 potential retries is because even if there is a serialisation error it is still likely that the adjustment in order quantity will be a valid request so 5 attempts are made each starting from the beginning before the user is notified their adjustment attempt failed.

Diagram

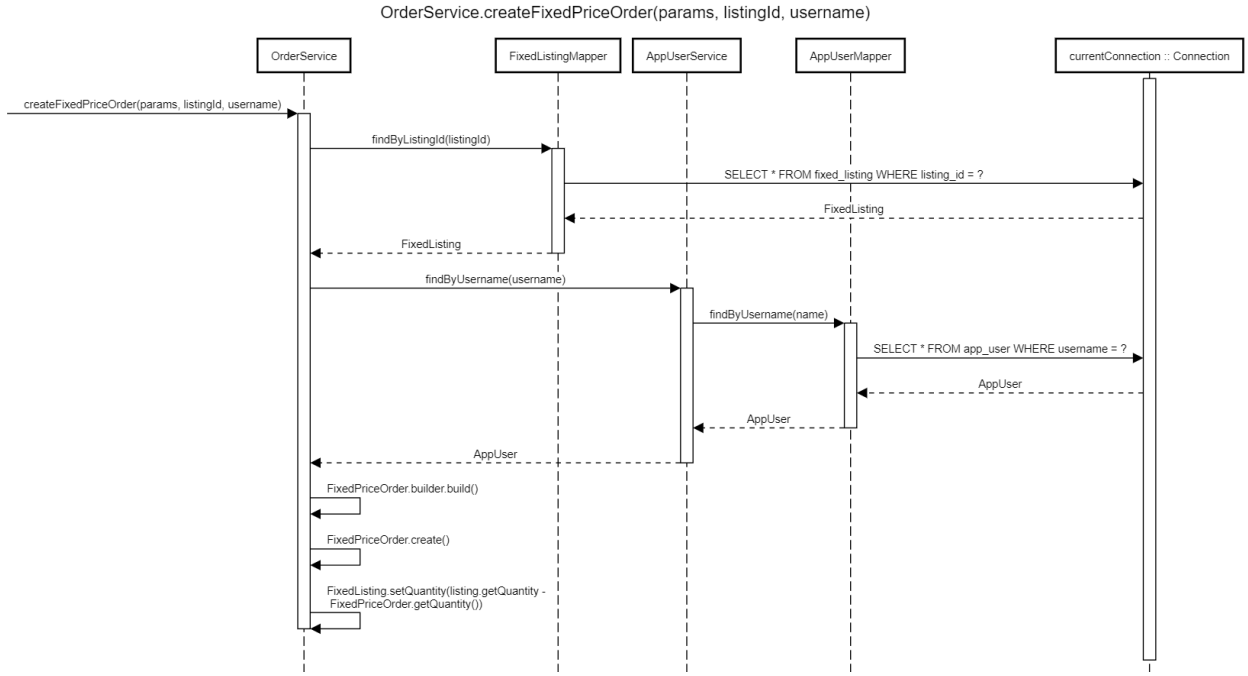
Sequence diagram for updating a fixed price order: Shown here is the check to ensure that the user is authorised and the subsequent unit of work implementation of updating both the order and the fixed price listing with the difference.



Sequence diagram for deleting a fixed price order: Shown here is the check to ensure that the user is authorised and the subsequent unit of work implementation of deleting the order and updating the listing using the difference.



Sequence diagram for creating a fixed price order: Shown here is the unit of work implementation of creating the order and updating the listing using the difference.



3.3.7. Create a Listing

Isolation Level: Repeatable Read

Issue Description

Fixed price & auction listings are similar enough to be referred to collectively as listings here.

These business rules are relevant:

- Listings with exactly matching fields are allowed, apart from primary keys of course.
- A user must be in a seller group to create a listing.
- A user can only add listings to seller groups they are a member of.
- Users cannot be deleted.
- Seller group names cannot be changed after creation.
- If a user is in a seller group when they create a listing, that listing should still be created even if the user is removed from the group concurrently, unless the group was deleted.

Consider the worst case scenarios for this transaction:

- A user tries to add a listing to one of their seller groups, and is removed from that group after the transaction has verified them as a member but before the new listing is saved.

Rationale

There are two ways of dealing with the above issue:

1. Prevent users from being removed from a seller group once they start creating a listing.
2. Allow this to happen.

The first option would require pessimistic offline concurrency control: the application would need to identify when a user was creating a listing, and lock the rows for all their seller groups for the duration of the edit. This would prevent any concurrent access, but would significantly reduce the liveness of the application from the perspective of editing seller groups. It would lock both seller_group and user_sg_mapping tables whenever someone created a listing, which would ruin the liveness of the application for other use cases and be rather counterintuitive to users.

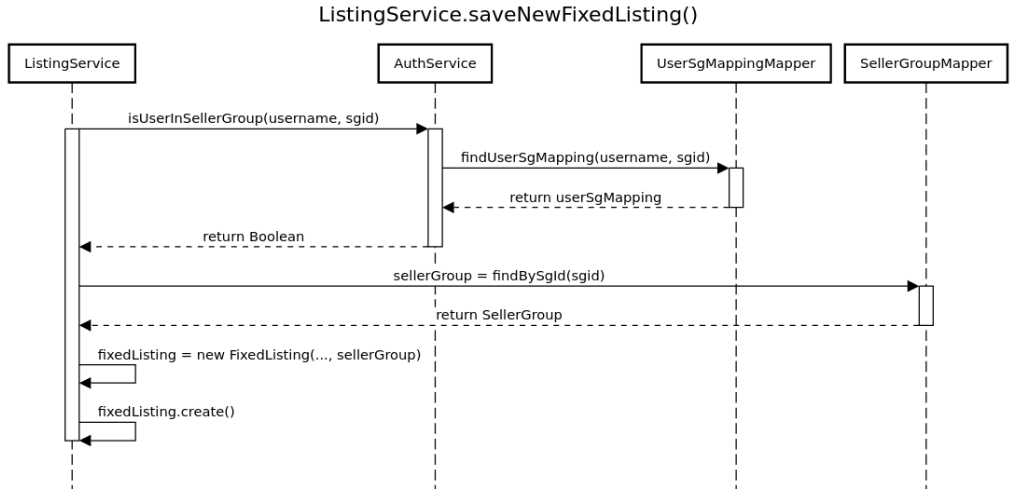
The second option can be implemented as a single system transaction using online concurrency control. Since a user is unlikely to remove themselves from a seller group while trying to add a new listing to it, the most likely source of this conflict would be an admin or other user removing a user from a seller group concurrently. In this case, the listing would just be created anyway. This is because our application business rules state that for the purposes of authorisation if the user is authorised when they begin their request to edit then they are considered authorised for the request. This means that a user becoming unauthorised during the system transaction should not cause the transaction to fail.

This use case is a core aspect of the system and requires extremely high throughput. We expect that users would not routinely be removed from seller groups; perhaps each group would

only have users removed from it once a month, so the likelihood of conflicts would be very low. Due to the low probability of conflicts, low cost of conflicts, and significant performance impacts associated with pessimistic control, we have chosen option 2.

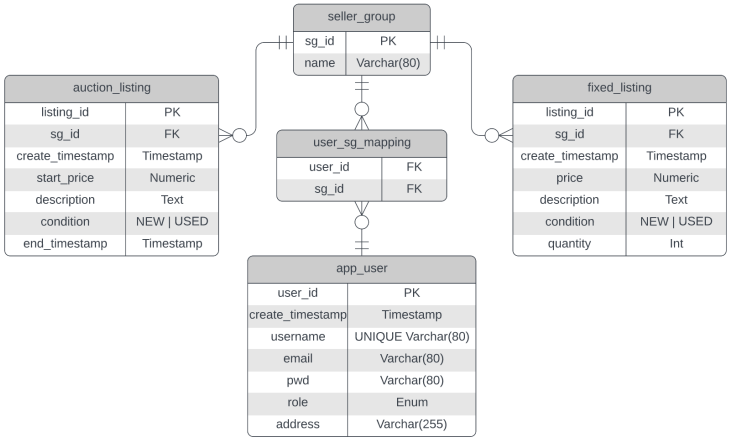
Implementation

As with other cases requiring concurrency control, this use case is implemented through a ManagedTransaction. The relevant method calls are summarised below.



The result of the call to UserSgMappingMapper is used to verify the user's membership, and the call to SellerGroupMapper returns the actual SellerGroup domain object.

Importantly, we cannot rely on violation of foreign key constraints to cause rollbacks in the case where a user is removed from a seller group, since user ids are not referenced in either of the fixed_listing or auction_listing tables, as shown in the below excerpt from our database. This is the main reason for allowing listing creation to occur if a user is concurrently removed from a seller group.



In all but one scenario in which this operation fails, the transaction should not be retried. The special case is when a seller group's name is changed during the transaction. This will cause a

rollback, and the transaction will fail. We have set the retry amount to one to accommodate this, even though it is not a use case we have implemented.

3.3.8. Delete a Listing

Isolation Level: Repeatable Read

Description

Deleting a listing is polymorphic and applies to both fixed price and auction listings.

These business rules are relevant:

- The administrator can delete any listings.
- For non-administrator users, they must be a member of the same seller group as the one linked to the listing they want to delete.
- If a user was authorised to delete a listing when they first requested the deletion, then the request should be honoured.

Rationale

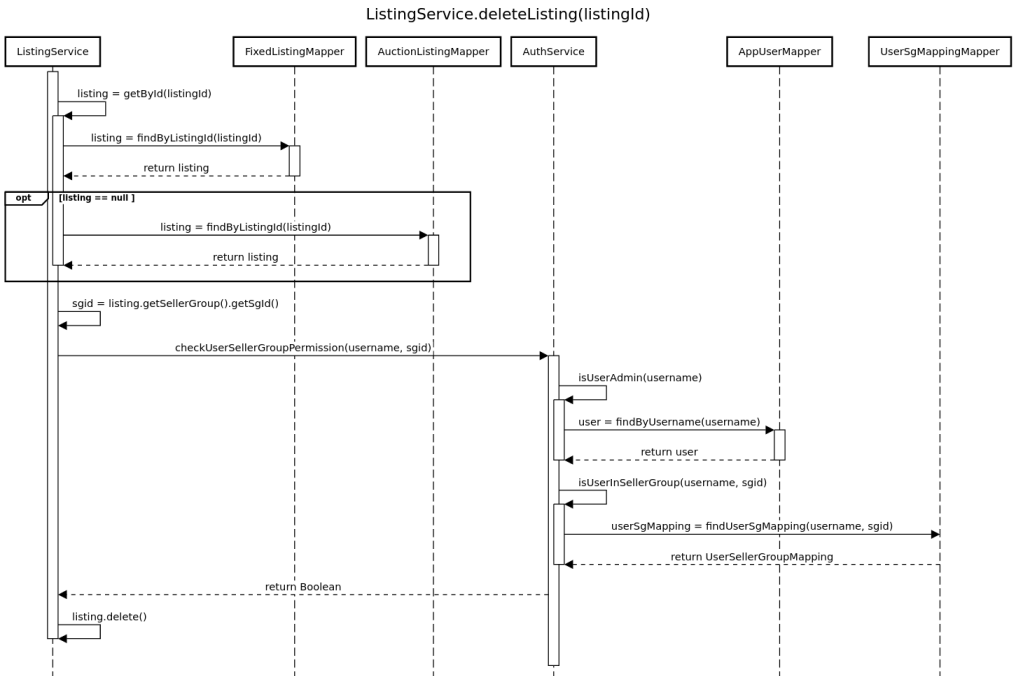
Consider the worst case scenarios for this transaction:

- A user is removed from the seller group they are trying to delete a listing from after the transaction has verified them as a member of it, but before the transaction has deleted the listing.

The rationale for this use case is identical to that of 3.3.6 (Create a Listing).

Implementation

The following diagram shows the calls made to delete a fixed price listing.



As shown above, the process of authentication is similar to that for creating a listing, which is the main reason why this case has the same isolation level of Repeatable Read. We want a snapshot of the database from before the transaction began so that we can allow users to delete anything they would have been able to delete at the instant the transaction started.

The additional calls to the FixedListingMapper and AuctionListingMapper are not particularly relevant here, since we have to use Repeatable Read on account of the points made in the previous paragraph. It is worth mentioning that if the calls to the listing mappers were the only interaction with the database before deletion, we could probably get away with using an isolation level of Read Committed. That is unfortunately not the case here.

Compared to Creation of Listings, we are less concerned with foreign key constraint violations. If a listing doesn't exist in the database and we try to delete it, it really doesn't matter since the operation is idempotent. For this reason, and because listings are more likely to be edited than seller groups, we have set the retry amount for this transaction to 5 to allow for the increased expected frequency of changes to listings which we may try to delete.

3.3.9. Create a Seller Group

Isolation Level: Read Committed

Description

Seller groups of the same name can't be created because the seller groups name is set to unique in the database. The only issue which could arise is that a user could be deleted while creating a new seller group.

Rationale

This is a relatively simple use case and consequently does not require any concurrency control. It isn't a problem if a seller group with the same name is created because the database transaction will be rejected on the violation of a unique constraint. As a result, the transaction that commits first will simply be given priority and the slower transactions will be rejected by violating the unique name constraint. There is no need for any concurrency control in this case.

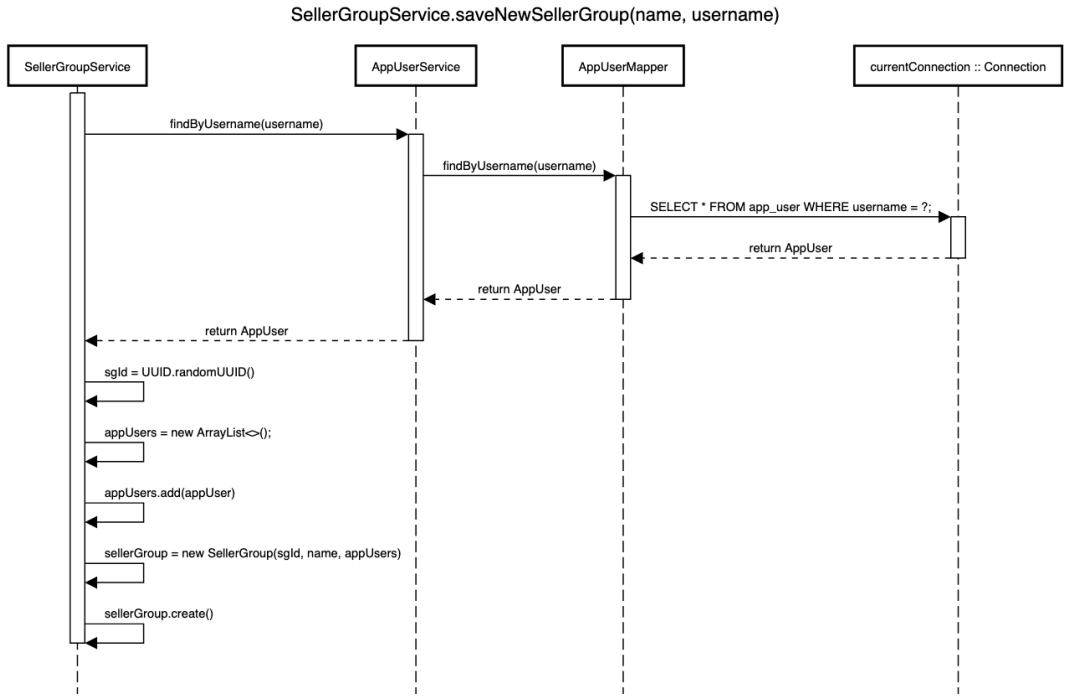
A potential problem that could arise is that a user could be deleted by an admin whilst they are creating a seller group. Deleting the user will cascade delete the user seller group mappings and so a seller group will be created which has no users in it. The presence of seller groups with no users in them is benign and thus does not require strict concurrency control.

Implementation

Read committed has been used for this business transaction with no retries. No retries are required because there isn't any possibility of a rollback. Read committed is used because stricter isolation levels are unnecessary due to the lack of concurrency issues for this transaction. Read committed is effectively identical to using read uncommitted and no concurrency control because Postgres doesn't allow dirty reads.

Diagram

The following diagram shows the calls made to save a new seller group.



3.3.10. Delete a seller group

Isolation Level: Read Committed

Description

In order to delete a seller group, the deleting user must either be an admin or be in the seller group that is being deleted. Whilst this is still a relatively simple transaction, there are a few issues worth considering. The user could be deleted by an admin whilst they are trying to delete a seller group, the user could be removed from the group they are trying to delete during the transaction and the seller group could be deleted by another user before this transaction commits.

Rationale

The first two considerations are where a user loses permission to delete a seller group, because during the transaction to delete the seller group the user has either been deleted by an admin or another user has removed them from the seller group they are trying to delete. We have decided that it is acceptable for users to still be able to delete the seller group because it is of little consequence that a user will have this permission for a very short period of time after it is revoked.

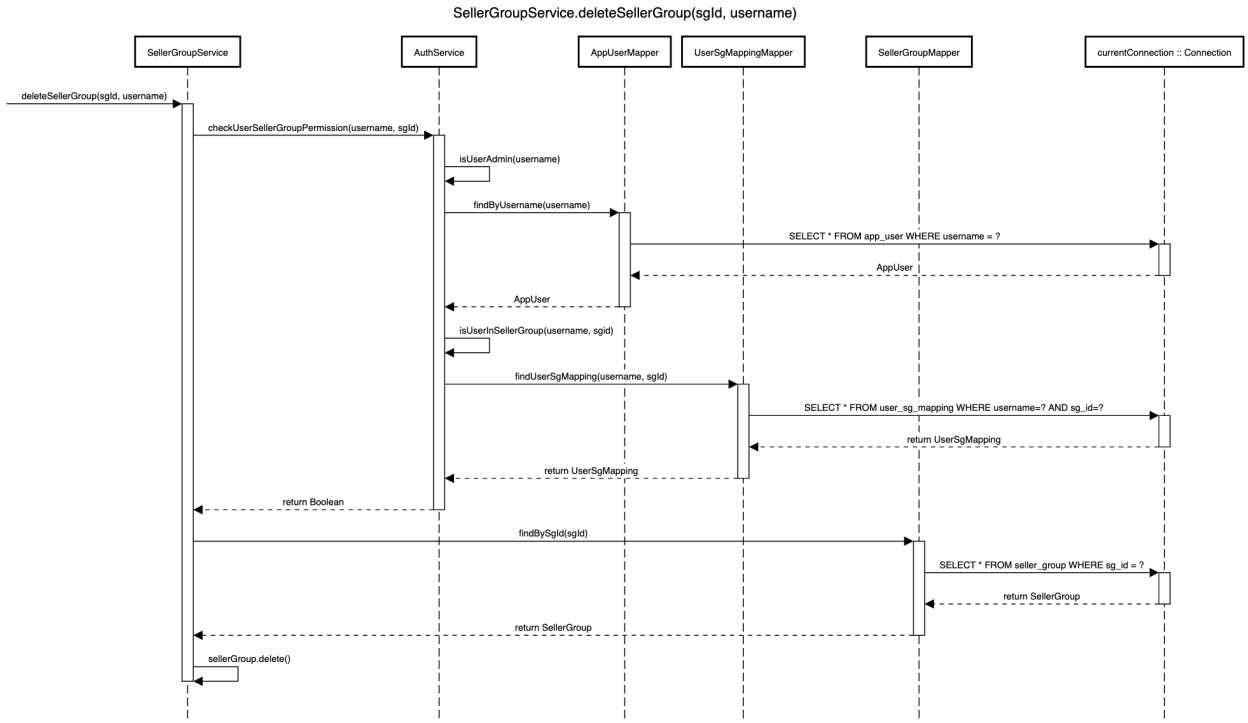
The other concurrency consideration is that the seller group could be deleted by another user before the transaction commits. In this situation, if the transaction has not reached the stage where the deleting user's permissions are checked, the user will simply be denied because the mapping will be deleted on cascade and so the checkUserSellerGroupPermission method will deny the current transaction the ability to delete the seller group. Even if the transaction has passed the stage where permissions are checked, the seller group object will be null and so the empty Unit of Work will be aborted.

Implementation

Read committed has been used for this transaction with no retries. As discussed above, the potential concurrency issues are benign and so the weakest form of concurrency control in Postgres is used. Retries aren't needed because this transaction will never rollback.

Diagram

The following diagram shows the calls made to delete a seller group.



3.3.11. Add and remove user from a seller group

Isolation Level: Read Committed

Description

Due to their similarity, the business transactions for adding users and removing users from a seller group have been combined. The user which is adding or removing another user from a seller group will be referred to as the acting user, the user which is being added or removed will be referred to as the target user.

In order to add or remove a user from a seller group, the initiating user must either be an admin or a member of the seller group at the start of the transaction. With this in mind, there are a few concurrency considerations worth mentioning. The removing user could lose permission to remove the user they want to because the user is either deleted by an admin or they themselves are removed from the seller group before they can remove the user they want to. Additionally, the seller group could be deleted before the target user could be added or removed. Finally, the target user could already be added or removed from the seller group before the transaction commits.

Rationale

In the case where the acting user loses permissions to add or remove the target user either because they are deleted by an admin or removed from the seller group, we have decided that we will still allow the target user to be added or removed. If the acting user had permission at the start of the transaction, according to our application business rules, the transaction can go ahead even if the user's permission was revoked.

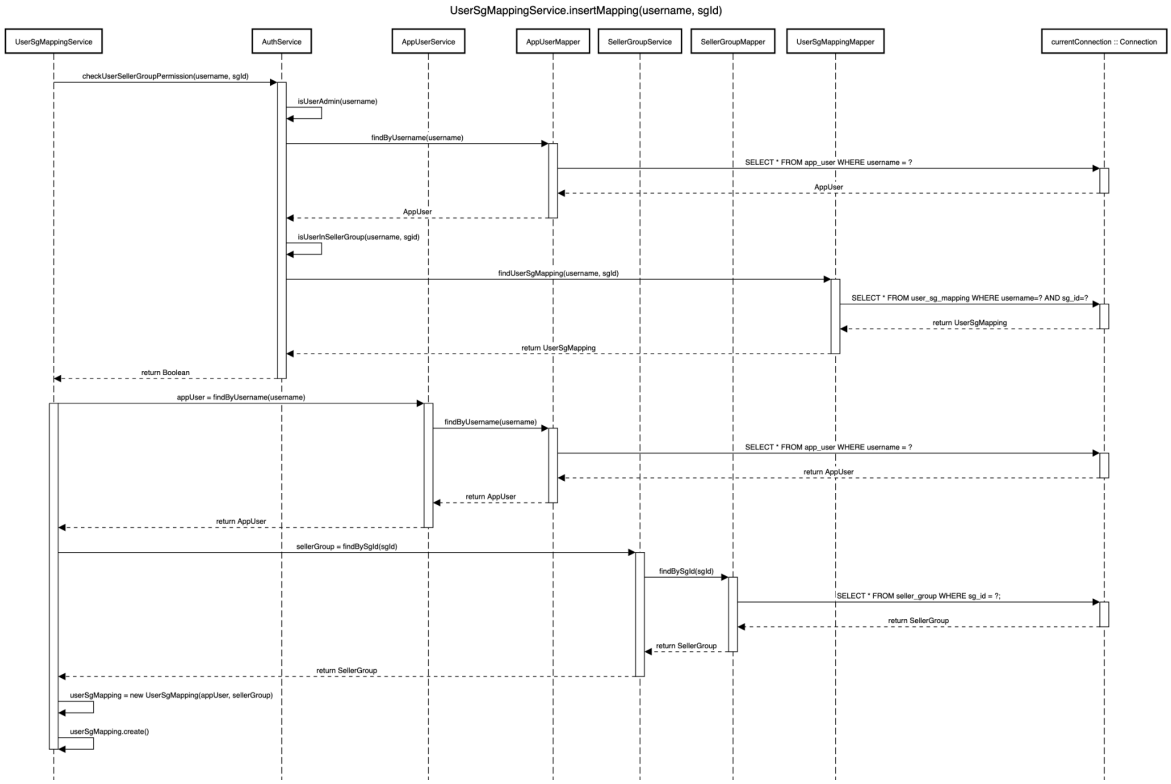
If the seller group or target user is added or removed by another transaction before the target user is added or removed it is of little consequence because the Unit of Work will abort and the transaction will rollback as the operation has already been completed by another transaction.

Implementation

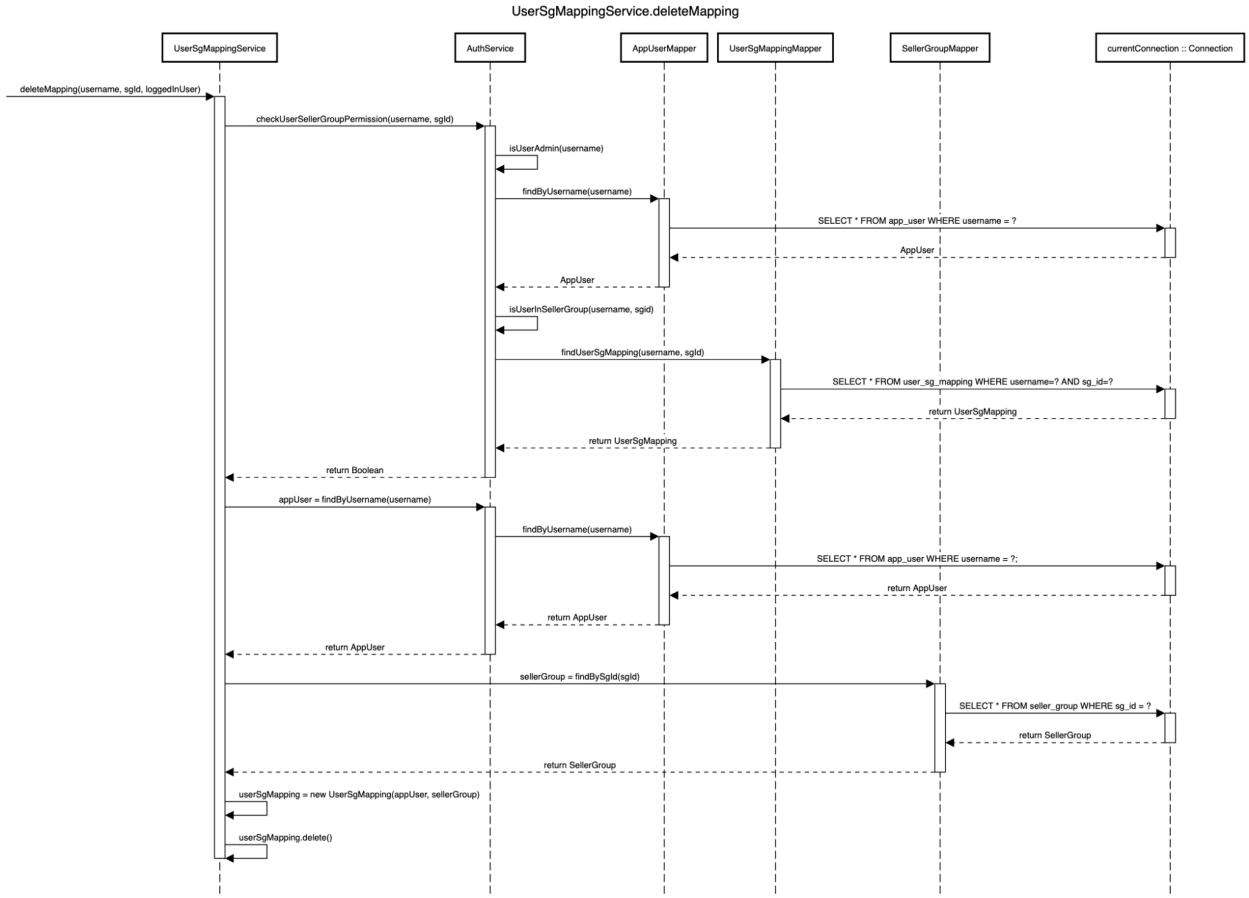
Read committed has been used for this transaction with no retries. There are no situations where concurrent transactions can cause problems and so the weakest isolation level has been used. The only situation where rollbacks will occur is when the same operation has already been committed by another transaction and there is no need for any retries.

Diagrams

The following diagram shows the calls made to add a user to a seller group.



The following diagram shows the calls made to remove a user from a seller group.



4. Concurrency Testing

4.1. Testing Strategy

4.1.1. JUnit Concurrency Testing

We chose to use JUnit for our primary concurrency testing framework. Through JUnit we can write unit tests that run specific code sections and observe the outcomes. Using JUnit for concurrency testing our strategy involved creating a set of threads that would perform operations on the system using a latching mechanism to start them at the same time. From this we could observe not only the resultant system state but also the logs generated during the test thanks to our comprehensive code logging.

This approach does not guarantee the absence of concurrency issues, since we cannot guarantee that the threads will interleave in a way which causes issues, however it is highly likely to identify issues if any are present through repeated runs.

In our JUnit tests we chose to call service methods to perform the operations. The reason for this is that concurrency issues are caused at the service, mapper and database layers. We are not concerned with control and view layer problems as they are not a result of concurrency. Running the tests from the service layer down also allowed us to focus on the causes of concurrency issues without worrying about UI. It also allows us to create tests using invalid inputs to ensure that the concurrency controls handle all cases.

4.1.2. Locust Concurrency/Load Testing

To implement a more complicated test we used a Python load testing tool called Locust https://locust.io/. The tool allows you to write http tests which can run against your web server programmatically. We decided against using Apache Jmeter because we wanted to implement some complicated test cases and we felt that doing so by clicking through a GUI would be limiting.

The main scenario we implemented was a bidding war between multiple users. Inside src/test/python/locustfile.py we defined a user who would register themselves with pseudo-random credentials, then login, then start repeatedly bidding on the same listing. Importantly, the user reads the value of the most recent bid on the auction listing and sends a bid which is $5 more. Locust allows us to create thousands of these users and get them to all compete on the same auction listing at once.

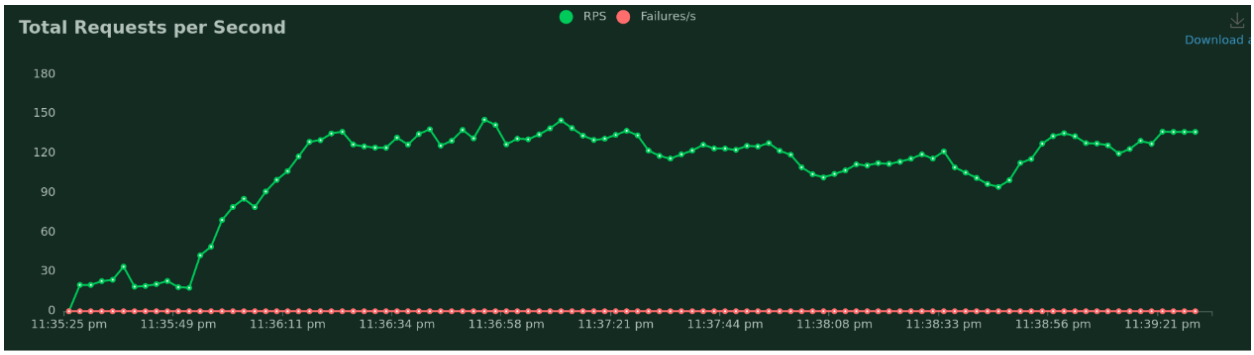
We will expand on this topic in the next submission, however the thing that is relevant to the current submission (i.e. concurrency) is that our system supports thousands of users concurrently registering for the first time, and then also supports them engaging in a bidding war on the SAME listing.

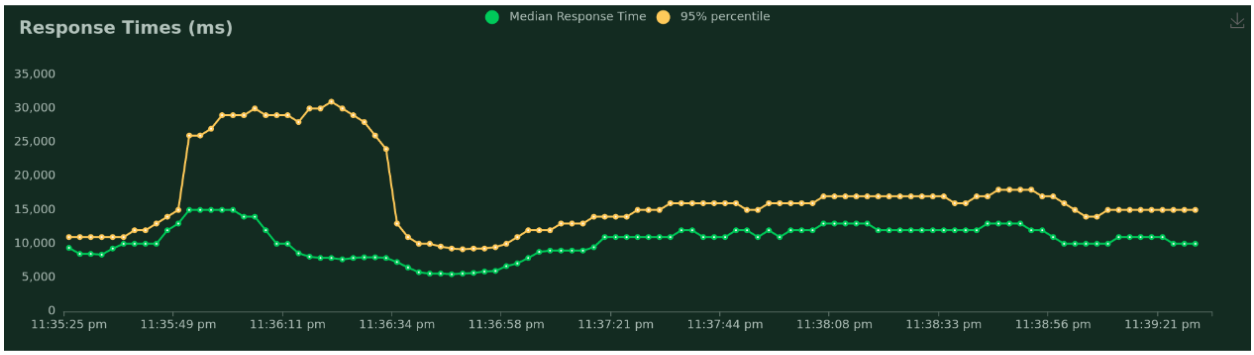
Locust allows you to set a threshold for the maximum number of users you want to test your system with, and a ramp-up number which defines how many of those users should be created per second.

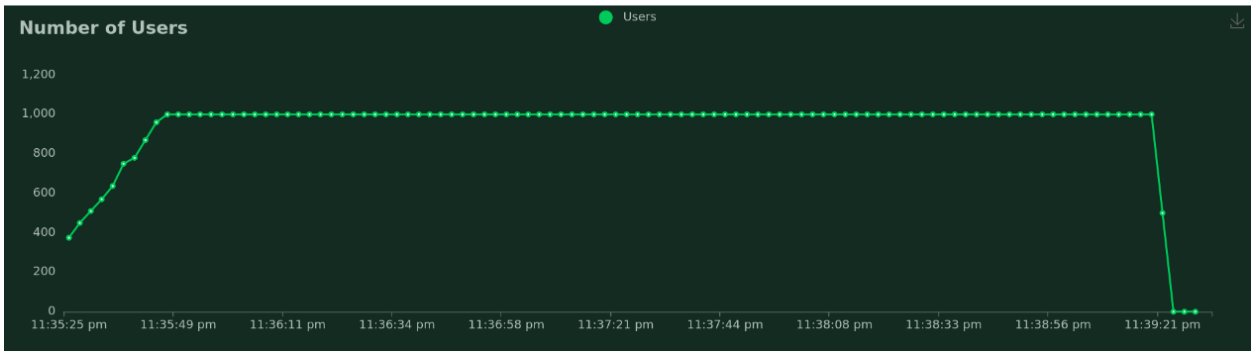
In the following experiment, we allocated 1000 users, with 50 new users being created (and starting to bid against each other) per second.

Due to Python's GIL, this test was run with 4 docker containers all sending the same requests to the server.

Instructions for replicating this are contained in the README.md file, and the results are summarised below.







We see that our application initially struggles to cope with the large number of new users registering, with a peak in response times of around 30 seconds. This is not due to concurrency issues but is a limitation of the hashing algorithm we have used as a part of Spring security.

After this initial performance hit, our application settles down to a response time of around 10 seconds. At this point the bidding war has 1000 users engaged and our concurrency system is being put through its paces. 10 seconds is still a rather long response time, however my computer isn't particularly fast.

The most important point here is the fact that none of the requests fail throughout the entire test. The table below shows this.

Type	Name	# Requests	# Fails	Median (ms)	90%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	/auth/landing-page	6514	0	5100	12000	13000	4910	2	14039	1404	8.4	0
GET	/auth/listing/listing-all	6277	0	8900	14000	17000	7827	8	22922	5351	9.4	0
GET	/auth/listing /view?listing_id=8770a207-6c72-465e-a1ef-46a128c6ad71	5979	0	9300	14000	17000	8052	5	21659	1854	8.3	0
POST	/auth/listing /view?listing_id=8770a207-6c72-465e-a1ef-46a128c6ad71	5678	0	10000	16000	20000	8983	67	28452	1854	8.5	0
POST	/login	1000	0	25000	32000	37000	22490	351	40148	1404	0	0
POST	/logout	1000	0	8800	9400	9700	8605	5525	9929	365	1.1	0
POST	/register	1000	0	13000	16000	18000	12332	4982	20402	365	0	0
	Aggregated	**27448**	**0**	**8800**	**15000**	**27000**	**8150**	**2**	**40148**	**2422**	**35.7**	**0**

From this we can conclude that, even if our application is somewhat slow, its concurrency system does not prevent it from working under load.

4.2. Testing Outcomes

In this section we outline in detail our tests for concurrency. Importantly, for all the tests, the database must be reset between each test.

4.2.1. Creating New User

Chosen Methods Rationale

JUnit was chosen as the primary testing method for this concurrency concern. Given the relative safety and simplicity of the use case a single thread based concurrency test will be sufficient to show safety.

Test Case 1 - Many attempts to create user with the same username

In this JUnit test multiple threads are created that all call the service method to save a new user with the same username. These threads will all be started simultaneously through a latch.

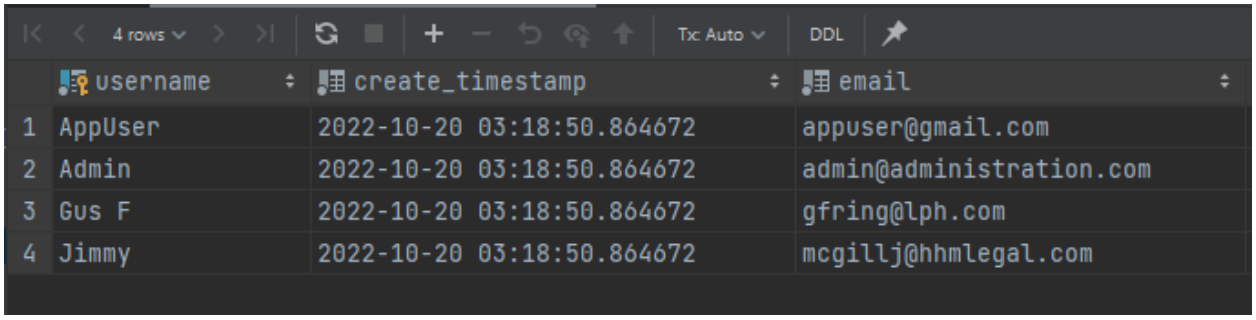
Expected Outcome:

Single new user added to the database with the correct username. We expect to observe rollbacks for the username being taken already.

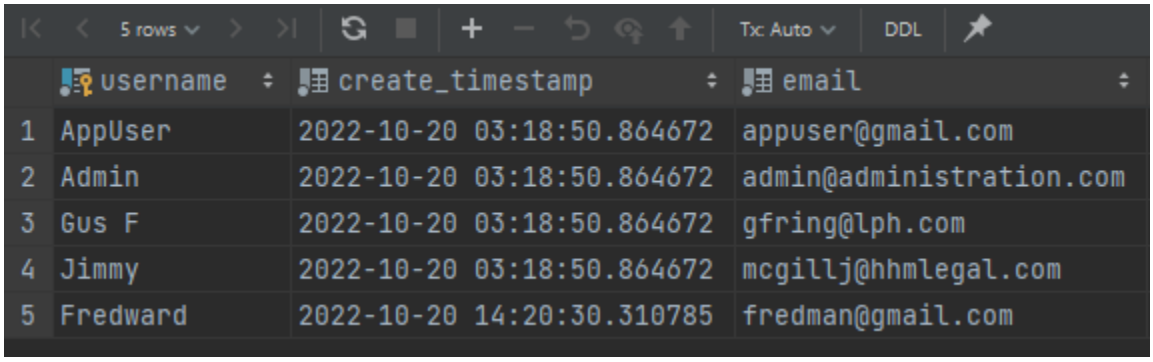
Outcome:

As we can see with the before and after results from the database only a single user was created.

Using the query “SELECT * FROM app_user;”:

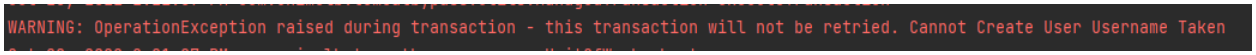


	username	create_timestamp	email
1	AppUser	2022-10-20 03:18:50.864672	appuser@gmail.com
2	Admin	2022-10-20 03:18:50.864672	admin@administration.com
3	Gus F	2022-10-20 03:18:50.864672	gfring@lph.com
4	Jimmy	2022-10-20 03:18:50.864672	mcgillj@hhmlegal.com



	username	create_timestamp	email
1	AppUser	2022-10-20 03:18:50.864672	appuser@gmail.com
2	Admin	2022-10-20 03:18:50.864672	admin@administration.com
3	Gus F	2022-10-20 03:18:50.864672	gfring@lph.com
4	Jimmy	2022-10-20 03:18:50.864672	mcgillj@hhmlegal.com
5	Fredward	2022-10-20 14:20:30.310785	fredman@gmail.com

We also observe that the logs show the correct operations exception:



4.2.2. Creating a Bid

Chosen Methods Rationale

JUnit was chosen as the primary testing method for this concurrency concern. The concurrency requirements for the bid creation operation are virtually non-existent, so the only test created for this was a sanity check.

Test Case 1 - Create many bids for a single listing.

For this test, we create 50 bids for a single auction listing concurrently, each with a different value. For each bid, we choose an initial bid of 5 and increase it by 1 every bid that is made, meaning the final bid is for 54. The even-valued bids are created by “App User” and the others are created by “Jimmy”. This test allows us to check whether the concurrent execution of bids behaves as intended - that is to say, we get the same final bid and a reasonable number of bids in the database.

Expected Outcome:

The highest bid should be for 54 by “App User”. There should be a proportionally small number of bids in the database compared to the number of bids placed, as once a large bid is committed, any lower bid that starts afterwards will fail.

Outcomes:

We use the following SQL query to evaluate the impact of the test:

```
SELECT * FROM bid where listing_id='e3f81946-05d4-4d64-a935-88c80e005d49';
```

Before running the test, we have a small number of bids in the system:

	bid_id	username	listing_id	create_timestamp	value
1	c0925bff-c231-491e-b15a-46b02b1fb63e	AppUser	e3f81946-05d4-4d64-a935-88c80e005d49	2022-10-20 09:59:46.623040	1
2	60972de9-1693-4624-a832-81ff515a8f8f	AppUser	e3f81946-05d4-4d64-a935-88c80e005d49	2022-10-20 09:59:46.623040	0.5

After the test we see a small number of new bids and the highest bid being 54:

	bid_id	username	listing_id	create_timestamp	value
1	c0925bff-c231-491e-b15a-46b02b1fb63e	AppUser	e3f81946-05d4-4d64-a935-88c80e005d49	2022-10-20 10:01:22.149042	1
2	60972de9-1693-4624-a832-81ff515a8f8f	AppUser	e3f81946-05d4-4d64-a935-88c80e005d49	2022-10-20 10:01:22.149042	0.5
3	d2a085be-7f36-477a-9120-06e75e447fe2	Jimmy	e3f81946-05d4-4d64-a935-88c80e005d49	2022-10-20 21:01:25.670818	9
4	9ff41583-6387-4a5d-9fc9-e8d6b6e65ae2	AppUser	e3f81946-05d4-4d64-a935-88c80e005d49	2022-10-20 21:01:25.683819	10
5	9092c933-25db-466b-8472-92ccad63c4e6	AppUser	e3f81946-05d4-4d64-a935-88c80e005d49	2022-10-20 21:01:25.699819	54
6	85703c16-faae-41b3-8931-637cfa3393e7	Jimmy	e3f81946-05d4-4d64-a935-88c80e005d49	2022-10-20 21:01:25.700817	31

Note that this test is non-deterministic - the only consistent factor is that the highest bid should be 54. However, we can see that only four bids were created in this case - it seems that the 54 value bid was created early and that many other bids failed. We can see, however, that the bid for 31 was created after the 54 bid - this is not an issue that affects the outcome of the auction, and we therefore don’t consider this a problem.

4.2.3. Create Auction Order

Chosen Methods Rationale

The operation for creating an Auction Order uses the ‘serialisable’ isolation level, meaning that it should fail in the case that the transaction interleaves with another in a non-serially equivalent manner. The testing for this operation is designed to validate this behaviour in the context of our system. We have a test to ensure that it does indeed fail and prevent concurrency issues, and we also have a test to see whether making a modification that doesn’t impact the transaction causes a rollback.

Test Case 1 - Create multiple Auction Orders for the same Auction.

This simple test just checks the main concurrency concern - ensuring that only one Auction Order is created if multiple requests to do so are made simultaneously. Two threads are created to create the same Auction Order for a completed listing.

In this example the number of orders made is two to make tracking logs easier, but this has been tested with larger numbers with the same result

Expected Outcome:

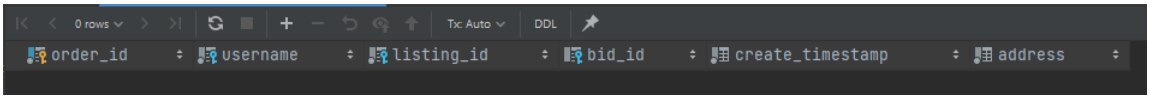
Only one Auction Order will be successfully created. Other requests should either result in a serialisation error and retry, finding that the request is no longer valid and aborting, or start after the new Auction Order is created and abort immediately.

Outcomes:

We use the following SQL query to evaluate the impact of the test:

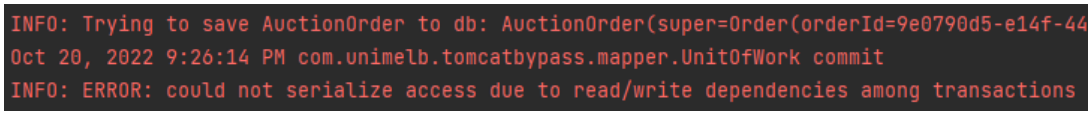
```
SELECT * FROM auction_order where listing_id='01e48a6c-cf8a-49fd-b719-7f81601eb338';
```

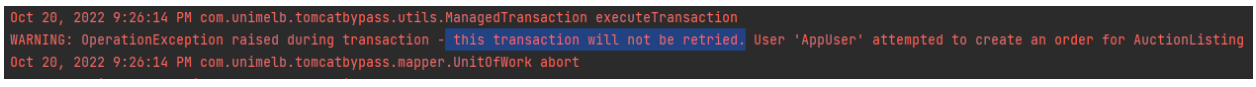
Before running the test, there is no Auction Order for the Listing:



After the test we see only one Auction Order:

And our logs confirm that the other encounters a serialisation error, then attempts to retry and safely fails:





Test Case 2 - Attempt to create an Auction Orders while many Bids are made for another Auction

This test was used to see the impact that choosing the ‘serialisable’ isolation level has on access to the database. There is risk of lost work when using such a strict isolation level, so in order to measure the impact we decided to run create Bid operations for a different listing while creating an Auction Order. These Bid operations are lifted from 4.3.2 Test Case 1. Create Auction Order reads from the Bid table, so if the PostgreSQL system does not intelligently identify that the result of that read cannot change, it may raise a serialisation error.

The test prints out whether the Auction Order was successfully created, and how many Bids succeeded.

Note that this test is primarily to verify the behaviour of PostgreSQL’s isolation level and to verify that it is suitable for this use case.

Expected Outcome:

The create Auction Order operation runs without triggering a retry and is successful. The Bid creation operations are tested elsewhere and are not our concern.

Outcomes:

We get the appropriate output from our test - create Auction Order succeeds:

```
Finished test - 2 out of 50 bids succeeded, and the auction order returned true
```

By doing a search on the output logs, we can verify that the Auction Order operation did not have to retry:

For this execution, it was verified that the create Auction Order was executed in the midst of a number of bids being created. This test, being non-deterministic, needed to be run multiple times to convince us that the result was stable.

4.2.4. Update / Delete Auction Order

Chosen Methods Rationale

JUnit was chosen as the primary testing method for this concurrency concern. Given the relative safety of the two operations given our business rules in relation to user edit / delete authorisation. Especially since we are comfortable with the most recent request completely overwriting any updated address we can use relatively simple and isolated unit tests.

Test Case 1 - Many threads attempt to edit the address of the same auction order with different addresses

A set of threads is created that will attempt to change the address to the integer index of their creation. These threads will be started simultaneously through a latch.

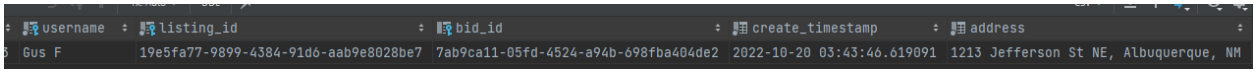
Expected Outcome:

The address of the auction order will be a random number in the range, being the result of the last query to make an edit. We will observe rollbacks given repeatable read.

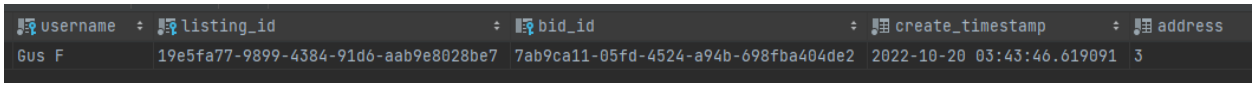
Outcome:

We can see the before and after result of the specific test we can see that the third thread was successful in being the last to have a non rolled back commit:

Using the SQL statement: SELECT * FROM auction_order WHERE order_id = '617fef77-6735-48f8-ad2d-ec8cc373faa3';

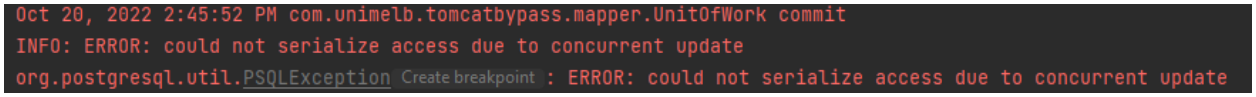


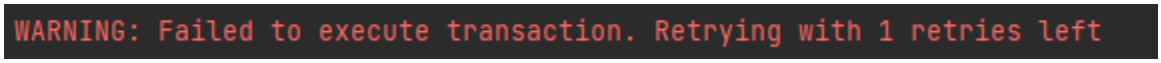
username	listing_id	bid_id	create_timestamp	address
Gus F	19e5fa77-9899-4384-91d6-aab9e8028be7	7ab9ca11-05fd-4524-a94b-698fba404de2	2022-10-20 03:43:46.619091	1213 Jefferson St NE, Albuquerque, NM



username	listing_id	bid_id	create_timestamp	address
Gus F	19e5fa77-9899-4384-91d6-aab9e8028be7	7ab9ca11-05fd-4524-a94b-698fba404de2	2022-10-20 03:43:46.619091	3

We also observed in the logs commits being rolled back and re-tried due to serialisation issues.





Test Case 2 - Many threads attempt to cancel the same auction order

A set of threads is created that will attempt to cancel the same auction order.

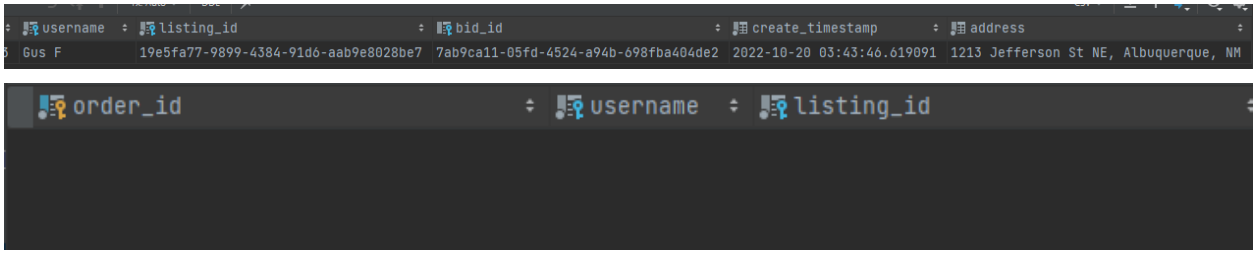
Expected Outcome:

The auction order will be removed from the database. The other transactions will fail and roll back.

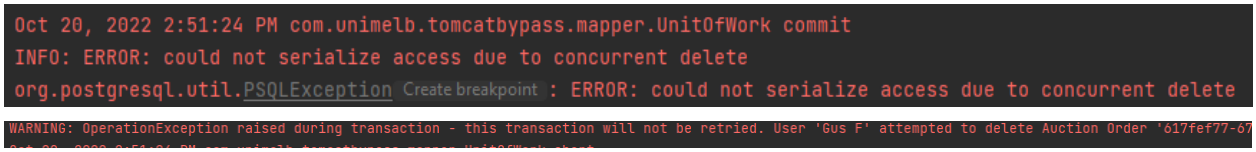
Outcome:

It is clear from the database results before and after the test that the order has indeed been cancelled

Using the SQL statement: SELECT * FROM auction_order WHERE order_id = '617fef77-6735-48f8-ad2d-ec8cc373faa3';



We also observe the 2 expected causes of rollback, serialisation and the lack of an order to cancel.



4.2.5. Edit / Create / Delete Fixed Price Order

Chosen Methods Rationale

JUnit testing was chosen as the primary method for evaluating the concurrency safety of these 3 concurrency concerns. These 3 operations represent some of the most complex concurrency in the system as a whole. Impacting multiple rows to change the quantities. We determined that through careful selection of thread creation we could create a suite of tests that allowed us to evaluate all possible interleavings of these 3 operations and the effects it would have on the database. Careful construction of JUnit tests with multiple runs will allow us to be confident in the consistency of the database.

Test Case 1 - Many threads attempt to edit a fixed price order in the same way

This is the most basic editing test. Multiple threads are set to edit the quantity of an existing order to the same number. The threads will be started simultaneously by a latch.

Expected Outcome:

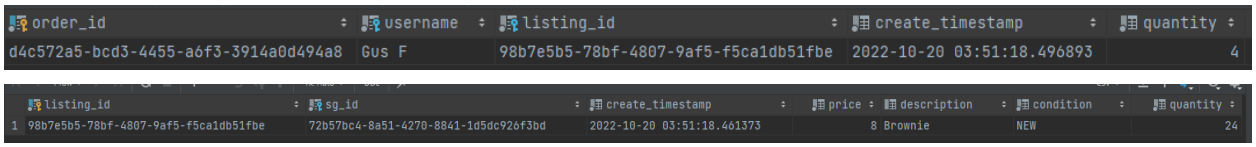
The order quantity should be set to the chosen number. The listing quantity should be updated to show that the new quantity is equal to the old quantity subtract the increase in order quantity. We also expect some serialisation roll backs to occur

Outcome:

We can observe that the original quantity of the listing is 24 and the original order quantity is 4. We are going to set the new amount to 10 so we expect to see the listing have 18 quantity.

Using queries:

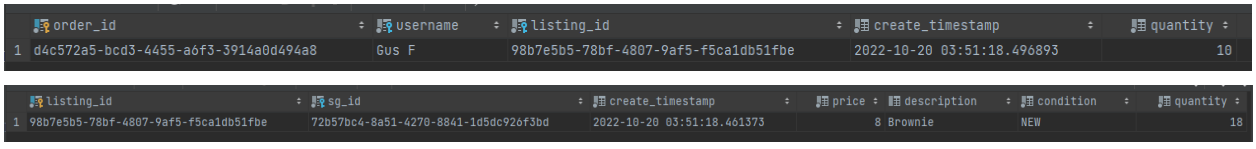
SELECT * FROM fixed_order WHERE order_id = 'd4c572a5-bcd3-4455-a6f3-3914a0d494a8';
SELECT * FROM fixed_listing WHERE listing_id = '98b7e5b5-78bf-4807-9af5-f5ca1db51fbe';



order_id	username	listing_id	create_timestamp	quantity
d4c572a5-bcd3-4455-a6f3-3914a0d494a8	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 03:51:18.496893	4

	listing_id	sg_id	create_timestamp	price	description	condition	quantity
1	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	72b57bc4-8a51-4270-8841-1d5dc926f3bd	2022-10-20 03:51:18.461373	8	Brownie	NEW	24

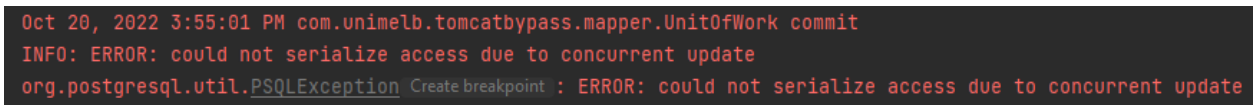
We can see our expected results after running the test:



	order_id	username	listing_id	create_timestamp	quantity
1	d4c572a5-bcd3-4455-a6f3-3914a0d494a8	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 03:51:18.496893	10

	listing_id	sg_id	create_timestamp	price	description	condition	quantity
1	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	72b57bc4-8a51-4270-8841-1d5dc926f3bd	2022-10-20 03:51:18.461373	8	Brownie	NEW	18

We see our expected serialisation rollbacks as well:



```
Oct 20, 2022 3:55:01 PM com.unimelb.tomcatbypass.mapper.UnitOfWork commit
INFO: ERROR: could not serialize access due to concurrent update
org.postgresql.util.PSQLException Create breakpoint : ERROR: could not serialize access due to concurrent update
```

Test Case 2 - Many threads attempt to edit the quantity of a fixed price order in a controlled randomised way

Threads are created and assigned a random integer within a range to edit the quantity of the order to. In this test case we have set the minimum and maximum order quantities to -5 and 30 respectively. The threads will be started simultaneously by a latch.

Expected Outcome:

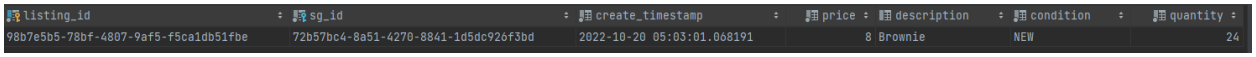
The new order quantity will be a valid (>0) number within our specified range. The listing quantity should be updated to show that the new quantity is equal to the old quantity subtract the increase in order quantity. We also expect some serialisation roll backs to occur

Outcome:

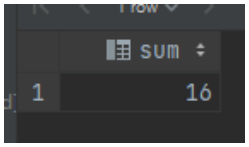
We can observe that the original quantity of the listing is 24 and the original sum of quantities of orders is 16 with the order in question having a quantity of 4.

Using queries:

SELECT * FROM fixed_listing WHERE listing_id = '98b7e5b5-78bf-4807-9a';
SELECT SUM(quantity) FROM fixed_order WHERE listing_id='98b7e5b5-78bf-4807-9af5-f5ca1db51fbe';
SELECT * FROM fixed_order WHERE order_id = 'd4c572a5-bcd3-4455-a6f3-3914a0d494a8';



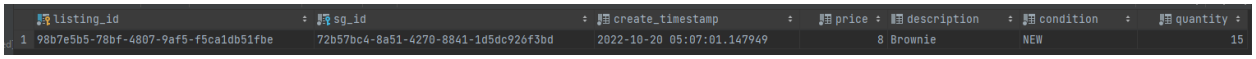
listing_id	sg_id	create_timestamp	price	description	condition	quantity
98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	72b57bc4-8a51-4270-8841-1d5dc926f3bd	2022-10-20 05:03:01.068191	8	Brownie	NEW	24



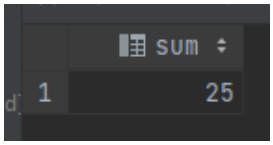
	sum
1	16

order_id	username	listing_id	create_timestamp	quantity
1 d4c572a5-bcd3-4455-a6f3-3914a0d494a8	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:03:01.108479	4

After the test we can observe the new fixed listing quantity is 15. As expected with this result the sum of quantities of orders is 25 and the edited order quantity is 13:



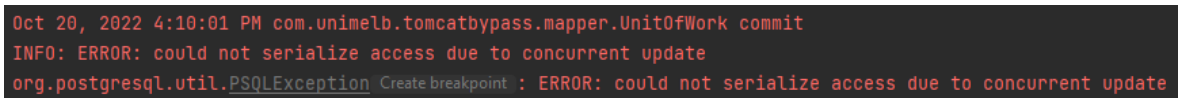
	listing_id	sg_id	create_timestamp	price	description	condition	quantity
1	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	72b57bc4-8a51-4270-8841-1d5dc926f3bd	2022-10-20 05:07:01.147949	8	Brownie	NEW	15



	sum
1	25

order_id	username	listing_id	create_timestamp	quantity
1 d4c572a5-bcd3-4455-a6f3-3914a0d494a8	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:07:01.172189	13

We also observe the expected serialisation rollbacks:



```
Oct 20, 2022 4:10:01 PM com.unimelb.tomcatbypass.mapper.UnitOfWork commit
INFO: ERROR: could not serialize access due to concurrent update
org.postgresql.util.PSQLException Create breakpoint : ERROR: could not serialize access due to concurrent update
```

Test Case 3 - Many threads attempt to buy (create a new) fixed price order with controlled randomised quantities

Threads are created and assigned a random integer within a range to buy new orders of the listing with the given quantity. The threads will be started simultaneously by a latch.

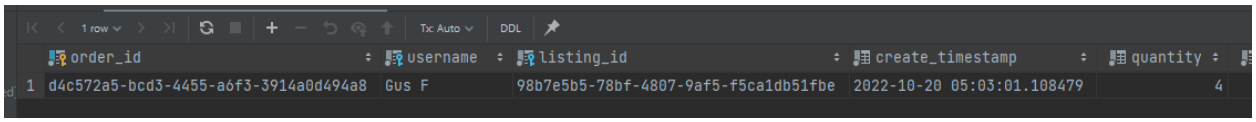
Expected Outcome:

New orders will be created with valid (>0) quantities from within our specified range. The total quantity of all orders in the system for the listing in addition to the remaining quantity of the listing should be the same before and after the test has proceeded. We expect rollbacks from serialisation issues and invalid quantities.

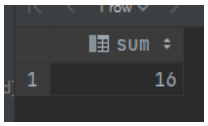
Outcome:

We can see the original quantity of the listing is 24 and the original sum of quantities of the orders is 16. It is also worth noting the original set of orders in the system.

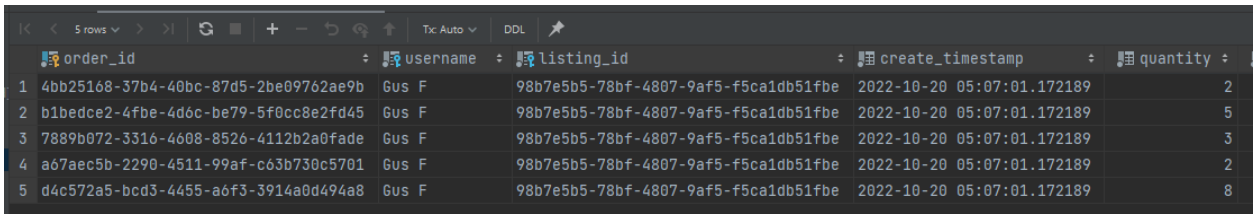
SELECT * FROM fixed_listing WHERE listing_id = '98b7e5b5-78bf-4807-9a';
SELECT SUM(quantity) FROM fixed_order WHERE listing_id='98b7e5b5-78bf-4807-9af5-f5ca1db51fbe';
SELECT * FROM fixed_order WHERE listing_id = '98b7e5b5-78bf-4807-9af5-f5ca1db51fbe';



	order_id	username	listing_id	create_timestamp	quantity
1	d4c572a5-bcd3-4455-a6f3-3914a0d494a8	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:03:01.108479	4

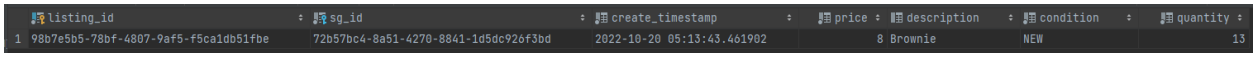


	sum
1	16

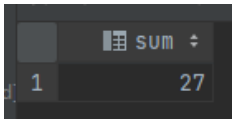


	order_id	username	listing_id	create_timestamp	quantity
1	4bb25168-37b4-40bc-87d5-2be09762ae9b	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:07:01.172189	2
2	b1bedce2-4fbe-4d6c-be79-5f0cc8e2fd45	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:07:01.172189	5
3	7889b072-3316-4608-8526-4112b2a0fade	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:07:01.172189	3
4	a67aec5b-2290-4511-99af-c63b730c5701	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:07:01.172189	2
5	d4c572a5-bcd3-4455-a6f3-3914a0d494a8	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:07:01.172189	8

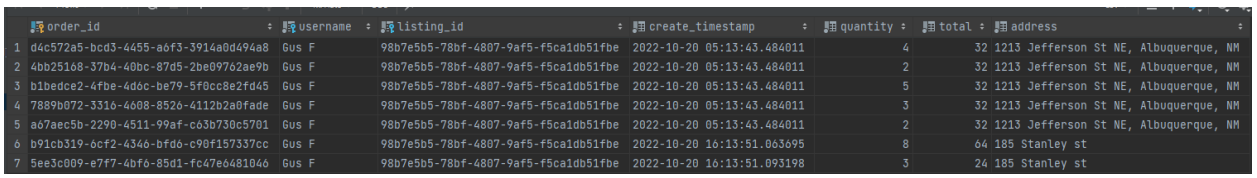
After the test we can observe that the listing quantity has decreased to 13, as expected the new sum across orders is 27, we can see this additional 11 quantity is made up by new orders:



	listing_id	sg_id	create_timestamp	price	description	condition	quantity
1	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	72b57bc4-8a51-4270-8841-1d5dc926f3bd	2022-10-20 05:13:43.461902	8	Brownie	NEW	13

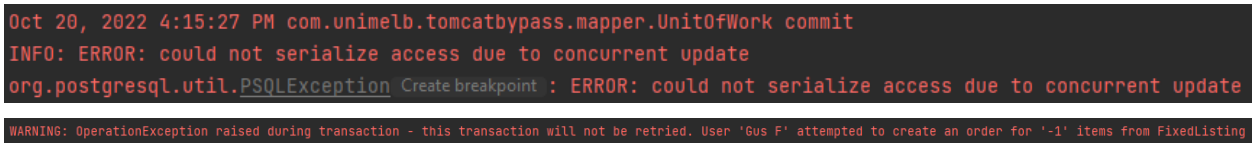


	sum
1	27



	order_id	username	listing_id	create_timestamp	quantity	total	address
1	d4c572a5-bcd3-4455-a6f3-3914a0d494a8	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:13:43.484011	4	32	1213 Jefferson St NE, Albuquerque, NM
2	4bb25168-37b4-40bc-87d5-2be09762ae9b	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:13:43.484011	2	32	1213 Jefferson St NE, Albuquerque, NM
3	b1bedce2-4fbe-4d6c-be79-5f0cc8e2fd45	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:13:43.484011	5	32	1213 Jefferson St NE, Albuquerque, NM
4	7889b072-3316-4608-8526-4112b2a0fade	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:13:43.484011	3	32	1213 Jefferson St NE, Albuquerque, NM
5	a67aec5b-2290-4511-99af-c63b730c5701	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:13:43.484011	2	32	1213 Jefferson St NE, Albuquerque, NM
6	b91cb319-6cf2-4346-bfd6-c90f157337cc	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 16:13:51.063695	8	64	185 Stanley st
7	5ee3c009-e7f7-4bf6-85d1-fc47e6481046	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 16:13:51.093198	3	24	185 Stanley st

We can also observe the expected rollback and invalid quantity rollbacks



Test Case 4 - Many threads attempt to cancel a set of fixed price orders

Threads are created and assigned orders to cancel from a set list of orders within the database. The threads will be started simultaneously by a latch.

Expected Outcome:

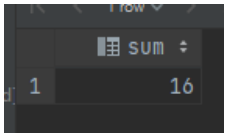
It is possible that not all orders will have enough rollback attempts to be cancelled but we should see all if not the vast majority of the orders cancelled. The listing quantity should be equal to the original quantity plus the combined quantities of all the cancelled orders. We expect to see rollbacks for serialisation errors

Outcome:

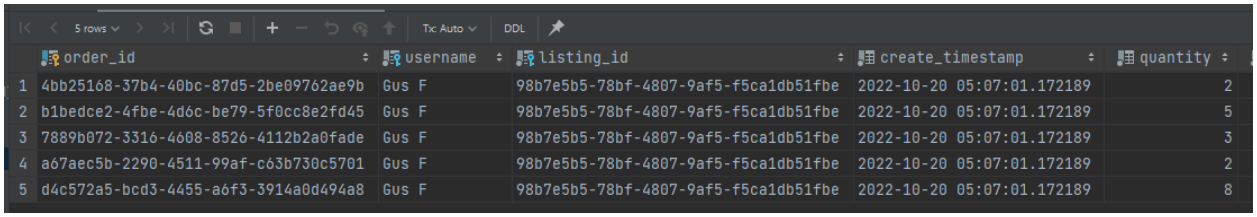
We can see the original quantity of the listing is 24 and the original sum of quantities of the orders is 16. It is also worth noting the original set of orders in the system.

SELECT * FROM fixed_listing WHERE listing_id = '98b7e5b5-78bf-4807-9a';
SELECT SUM(quantity) FROM fixed_order WHERE listing_id='98b7e5b5-78bf-4807-9af5-f5ca1db51fbe';
SELECT * FROM fixed_order WHERE listing_id = '98b7e5b5-78bf-4807-9af5-f5ca1db51fbe';

	order_id	username	listing_id	create_timestamp	quantity
1	d4c572a5-bcd3-4455-a6f3-3914a0d494a8	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:03:01.108479	4

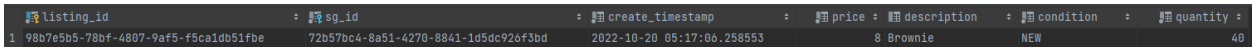


	sum
1	16

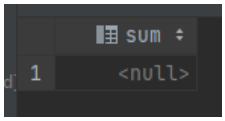


	order_id	username	listing_id	create_timestamp	quantity
1	4bb25168-37b4-40bc-87d5-2be09762ae9b	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:07:01.172189	2
2	b1bedce2-4fbe-4d6c-be79-5f0cc8e2fd45	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:07:01.172189	5
3	7889b072-3316-4608-8526-4112b2a0fade	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:07:01.172189	3
4	a67aec5b-2290-4511-99af-c63b730c5701	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:07:01.172189	2
5	d4c572a5-bcd3-4455-a6f3-3914a0d494a8	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:07:01.172189	8

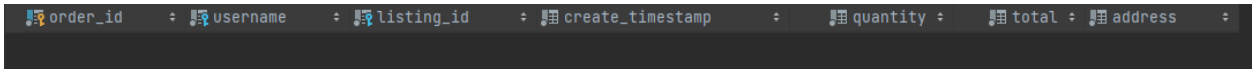
After the transaction we see that the quantity of the listing is 40, the sum of the orders is 0 and as expected all of the orders have been cancelled from the system



	listing_id	sg_id	create_timestamp	price	description	condition	quantity
1	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	72b57bc4-8a51-4270-8841-1d5dc926f3bd	2022-10-20 05:17:06.258553	8	Brownie	NEW	40

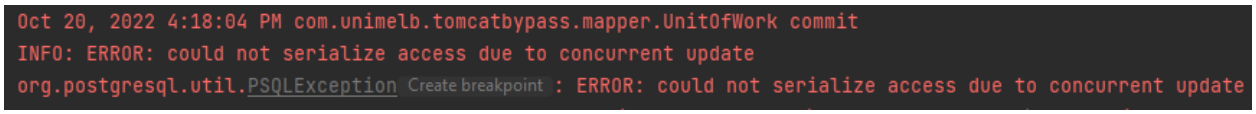


	sum
1	<null>



order_id	username	listing_id	create_timestamp	quantity	total	address

We also observe the expected serialisation rollback



```
Oct 20, 2022 4:18:04 PM com.unimelb.tomcatbypass.mapper.UnitOfWork commit
INFO: ERROR: could not serialize access due to concurrent update
org.postgresql.util.PSQLException Create breakpoint : ERROR: could not serialize access due to concurrent update
```

Test Case 5 - Many threads attempt to edit the quantity of a fixed price order in a controlled randomised way while another set of threads buys new orders in a controlled randomised way

Threads are created and assigned a random integer within a range to edit the quantity of the order to. More threads are created and assigned a random integer within a range to buy new orders of the listing with the given quantity. The threads will be started simultaneously by a latch.

Expected Outcome:

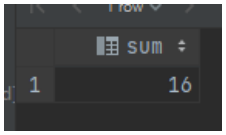
The new order quantity will be a valid (>0) number within our specified range. New orders will be created with valid (>0) quantities from within our specified range. The total quantity of all orders in the system for the listing in addition to the remaining quantity of the listing should be the same before and after the test has proceeded. We expect rollbacks from serialisation issues and invalid quantities

Outcome:

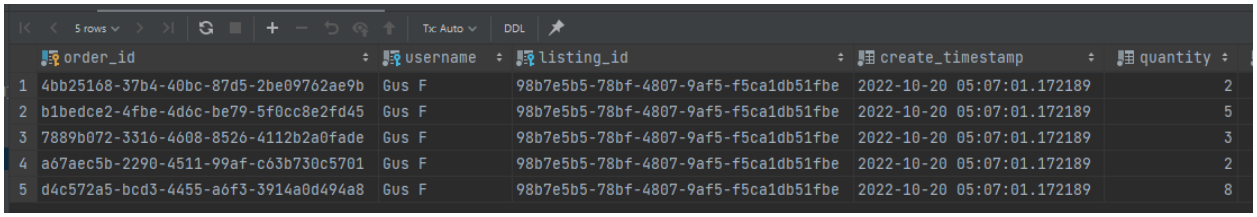
We can see the original quantity of the listing is 24 and the original sum of quantities of the orders is 16. It is also worth noting the original set of orders in the system.

SELECT * FROM fixed_listing WHERE listing_id = '98b7e5b5-78bf-4807-9a';
SELECT SUM(quantity) FROM fixed_order WHERE listing_id='98b7e5b5-78bf-4807-9af5-f5ca1db51fbe';
SELECT * FROM fixed_order WHERE listing_id = '98b7e5b5-78bf-4807-9af5-f5ca1db51fbe';

	order_id	username	listing_id	create_timestamp	quantity
1	d4c572a5-bcd3-4455-a6f3-3914a0d494a8	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:03:01.108479	4

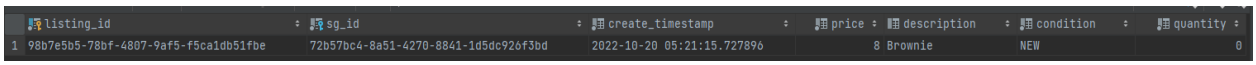


	SUM
1	16



	order_id	username	listing_id	create_timestamp	quantity
1	4bb25168-37b4-40bc-87d5-2be09762ae9b	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:07:01.172189	2
2	b1bedce2-4fbe-4d6c-be79-5f0cc8e2fd45	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:07:01.172189	5
3	7889b072-3316-4608-8526-4112b2a0fade	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:07:01.172189	3
4	a67aec5b-2290-4511-99af-c63b730c5701	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:07:01.172189	2
5	d4c572a5-bcd3-4455-a6f3-3914a0d494a8	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:07:01.172189	8

After running the test we observe that the quantity of the listing is 0 (wow all sold out) the sum of the quantities of orders is 40 and we can see that additional quantity made up on the edited original and new orders.



	listing_id	sg_id	create_timestamp	price	description	condition	quantity
1	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	72b57bc4-8a51-4270-8841-1d5dc926f3bd	2022-10-20 05:21:15.727896	8	Brownie	NEW	0

	sum
1	40

	order_id	username	listing_id	create_timestamp	quantity	total	address
1	4bb25168-37b4-40bc-87d5-2be09762ae9b	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:21:15.751095	2	32	1213 Jefferson St NE, Albuquerque, NM
2	b1bedce2-4fbe-4d6c-be79-5f0cc8e2fd45	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:21:15.751095	5	32	1213 Jefferson St NE, Albuquerque, NM
3	7889b072-3316-4608-8526-4112b2a0fade	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:21:15.751095	3	32	1213 Jefferson St NE, Albuquerque, NM
4	a67aec5b-2290-4511-99af-c63b730c5701	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:21:15.751095	2	32	1213 Jefferson St NE, Albuquerque, NM
5	752cebd2-97c1-4906-9563-ec09c7e10608	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 16:21:52.852294	15	120	185 Stanley st
6	d4c572a5-bcd3-4455-a6f3-3914a0d494a8	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:21:15.751095	13	32	185 Stanley st

We also observe the expected serialisation rollback and invalid quantity rollback:

```
Oct 20, 2022 4:23:37 PM com.unimelb.tomcatbypass.mapper.UnitOfWork commit
INFO: ERROR: could not serialize access due to concurrent update
org.postgresql.util.PSQLException Create breakpoint : ERROR: could not serialize access due to concurrent update
```

```
WARNING: OperationException raised during transaction - this transaction will not be retried. User 'Gus F' attempted to create an order for '24' items from FixedListing
```

Test Case 6 - Many threads attempt to buy new fixed price orders in a controlled randomised way, while other threads cancel a set of fixed price orders

Threads are created and assigned a random integer within a range to buy new orders of the listing with the given quantity. More threads are created and assigned orders to cancel from a set list of orders within the database. The threads will be started simultaneously by a latch.

Expected Outcome:

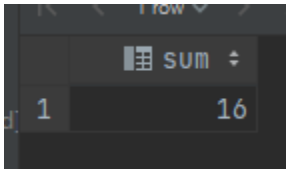
New orders will be created with valid (>0) quantities from within our specified range. It is possible that not all orders will have enough rollback attempts to be cancelled but we should see all if not the vast majority of the orders cancelled. The total quantity of all orders in the system for the listing in addition to the remaining quantity of the listing should be the same before and after the test has proceeded. We expect rollbacks from serialisation issues and invalid quantities

Outcome:

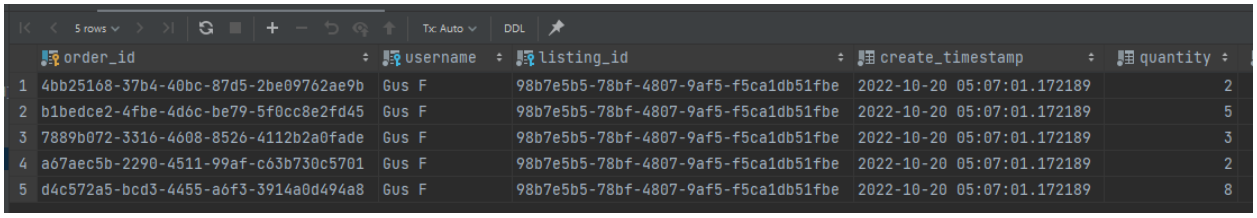
We can see the original quantity of the listing is 24 and the original sum of quantities of the orders is 16. It is also worth noting the original set of orders in the system.

SELECT * FROM fixed_listing WHERE listing_id = '98b7e5b5-78bf-4807-9a';
SELECT SUM(quantity) FROM fixed_order WHERE listing_id='98b7e5b5-78bf-4807-9af5-f5ca1db51fbe';
SELECT * FROM fixed_order WHERE listing_id = '98b7e5b5-78bf-4807-9af5-f5ca1db51fbe';

	order_id	username	listing_id	create_timestamp	quantity
1	d4c572a5-bcd3-4455-a6f3-3914a0d494a8	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:03:01.108479	4

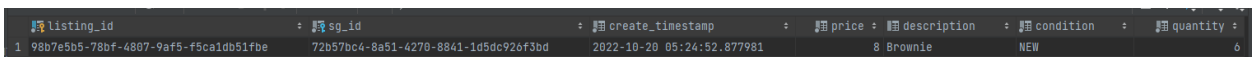


	sum
1	16

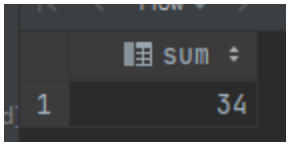


	order_id	username	listing_id	create_timestamp	quantity
1	4bb25168-37b4-40bc-87d5-2be09762ae9b	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:07:01.172189	2
2	b1bedce2-4fbe-4d6c-be79-5f0cc8e2fd45	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:07:01.172189	5
3	7889b072-3316-4608-8526-4112b2a0fade	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:07:01.172189	3
4	a67aec5b-2290-4511-99af-c63b730c5701	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:07:01.172189	2
5	d4c572a5-bcd3-4455-a6f3-3914a0d494a8	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:07:01.172189	8

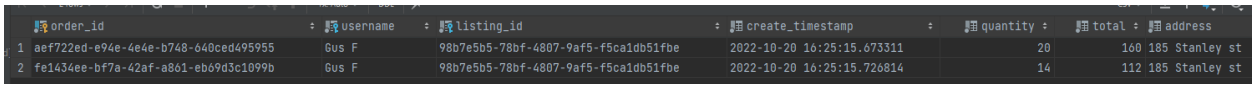
After the test we can see that the quantity of the listing is 6, as expected the total quantity across orders is 34 all made up of newly created orders as the initial orders have been cancelled.



	listing_id	sg_id	create_timestamp	price	description	condition	quantity
1	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	72b57bc4-8a51-4270-8841-1d5dc926f3bd	2022-10-20 05:24:52.877981	8	Brownie	NEW	6

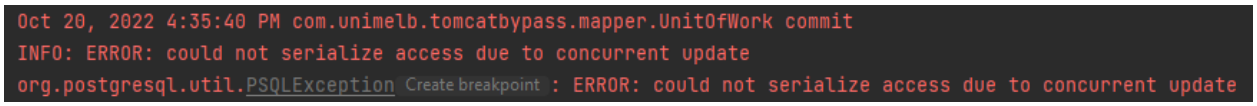


	sum
1	34



	order_id	username	listing_id	create_timestamp	quantity	total	address
1	aef722ed-e94e-4e4e-b748-640ced495955	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 16:25:15.673311	20	160	185 Stanley st
2	fe1434ee-bf7a-42af-a861-eb69d3c1099b	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 16:25:15.726814	14	112	185 Stanley st

We can also observe that we get the expected serialisation rollbacks and invalid quantity rollback:



```
Oct 20, 2022 4:35:40 PM com.unimelb.tomcatbypass.mapper.UnitOfWork commit
INFO: ERROR: could not serialize access due to concurrent update
org.postgresql.util.PSQLException Create breakpoint : ERROR: could not serialize access due to concurrent update
```

```
WARNING: OperationException raised during transaction - this transaction will not be retried. User 'Gus F' attempted to create an order for '23' items from FixedListing
```

Test Case 7 - Many threads attempt to edit the quantity of a fixed price order in a controlled randomised way, while other threads cancel a set of fixed price orders

Threads are created and assigned a random integer within a range to edit the quantity of the order to. More threads are created and assigned orders to cancel from a set list of orders within the database. The threads will be started simultaneously by a latch.

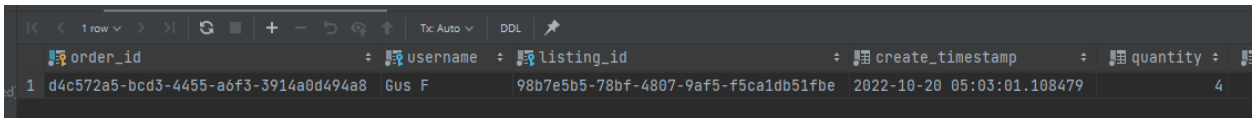
Expected Outcome:

The new order quantity will be a valid (>0) number within our specified range. It is possible that not all orders will have enough rollback attempts to be cancelled but we should see all if not the vast majority of the orders cancelled. The total quantity of all orders in the system for the listing in addition to the remaining quantity of the listing should be the same before and after the test has proceeded. We expect rollbacks from serialisation issues and invalid quantities

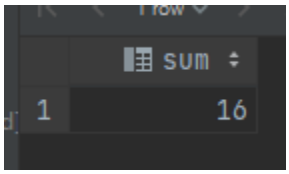
Outcome:

We can see the original quantity of the listing is 24 and the original sum of quantities of the orders is 16. It is also worth noting the original set of orders in the system.

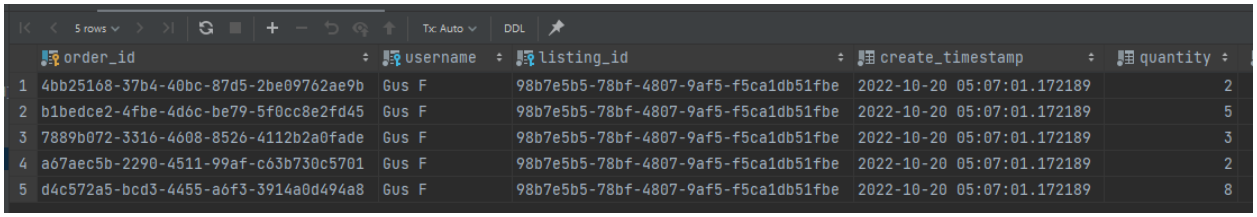
SELECT * FROM fixed_listing WHERE listing_id = '98b7e5b5-78bf-4807-9a';
SELECT SUM(quantity) FROM fixed_order WHERE listing_id='98b7e5b5-78bf-4807-9af5-f5ca1db51fbe';
SELECT * FROM fixed_order WHERE listing_id = '98b7e5b5-78bf-4807-9af5-f5ca1db51fbe';



	order_id	username	listing_id	create_timestamp	quantity
1	d4c572a5-bcd3-4455-a6f3-3914a0d494a8	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:03:01.108479	4

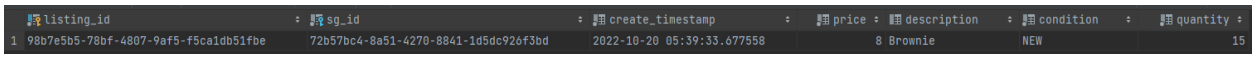


	sum
1	16

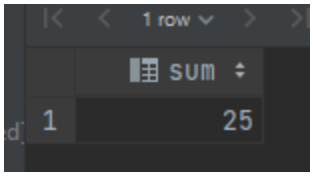


	order_id	username	listing_id	create_timestamp	quantity
1	4bb25168-37b4-40bc-87d5-2be09762ae9b	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:07:01.172189	2
2	b1bedce2-4fbe-4d6c-be79-5f0cc8e2fd45	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:07:01.172189	5
3	7889b072-3316-4608-8526-4112b2a0fade	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:07:01.172189	3
4	a67aec5b-2290-4511-99af-c63b730c5701	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:07:01.172189	2
5	d4c572a5-bcd3-4455-a6f3-3914a0d494a8	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:07:01.172189	8

After the test we can see that the listing quantity is 15, as expected the sum of quantities from the orders is 25 all being a result of the only uncancelled order



	listing_id	sg_id	create_timestamp	price	description	condition	quantity
1	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	72b57bc4-8a51-4270-8841-1d5dc926f3bd	2022-10-20 05:39:33.677558	8	Brownie	NEW	15



	order_id	username	listing_id	create_timestamp	quantity
1	d4c572a5-bcd3-4455-a6f3-3914a0d494a8	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:39:33.702329	25

We also observe the expected serialisation rollbacks in this particular test case we didn't not see any invalid update quantities occur. This is a result of the randomised set of potential quantities not choosing ones that were too high or too low during this test.

```
Oct 20, 2022 4:41:31 PM com.unimelb.tomcatbypass.mapper.UnitOfWork commit
INFO: ERROR: could not serialize access due to concurrent update
org.postgresql.util.PSQLException Create breakpoint : ERROR: could not serialize access due to concurrent update
```

Test Case 8 - Many threads attempt to edit the quantity of a fixed price order in a controlled randomised way, while other threads cancel a set of fixed price orders, while another set of threads buys new orders in a controlled randomised way

Threads are created and assigned a random integer within a range to edit the quantity of the order to. More threads are created and assigned orders to cancel from a set list of orders within the database. Additional threads are created and assigned a random integer within a range to buy new orders of the listing with the given quantity.

Expected Outcome:

The new order quantity will be a valid (>0) number within our specified range. New orders will be created with valid (>0) quantities from within our specified range. It is possible that not all orders will have enough rollback attempts to be cancelled but we should see all if not the vast majority of the orders cancelled. The total quantity of all orders in the system for the listing in addition to the remaining quantity of the listing should be the same before and after the test has proceeded. We expect rollbacks from serialisation issues and invalid quantities

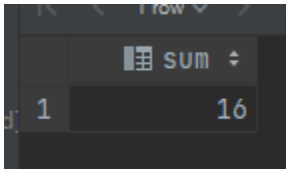
Outcome:

The final test brings together all others and has all 3 possible transactions interleave with each other.

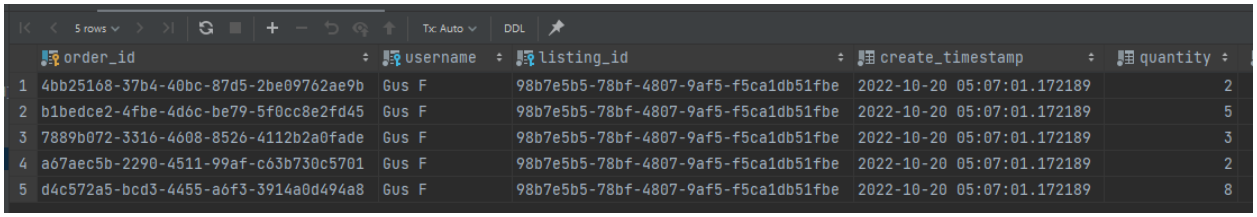
We can see the original quantity of the listing is 24 and the original sum of quantities of the orders is 16. It is also worth noting the original set of orders in the system.

SELECT * FROM fixed_listing WHERE listing_id = '98b7e5b5-78bf-4807-9a';
SELECT SUM(quantity) FROM fixed_order WHERE listing_id='98b7e5b5-78bf-4807-9af5-f5ca1db51fbe';
SELECT * FROM fixed_order WHERE listing_id = '98b7e5b5-78bf-4807-9af5-f5ca1db51fbe';

	order_id	username	listing_id	create_timestamp	quantity
1	d4c572a5-bcd3-4455-a6f3-3914a0d494a8	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:03:01.108479	4



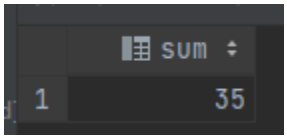
	sum
1	16



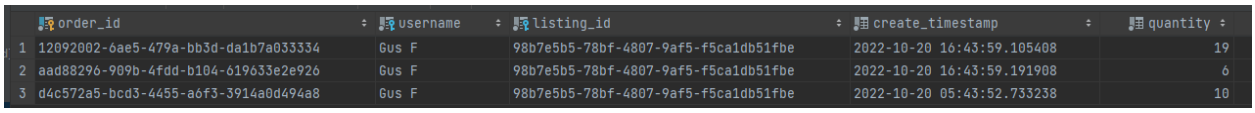
	order_id	username	listing_id	create_timestamp	quantity
1	4bb25168-37b4-40bc-87d5-2be09762ae9b	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:07:01.172189	2
2	b1bedce2-4fbe-4d6c-be79-5f0cc8e2fd45	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:07:01.172189	5
3	7889b072-3316-4608-8526-4112b2a0fade	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:07:01.172189	3
4	a67aec5b-2290-4511-99af-c63b730c5701	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:07:01.172189	2
5	d4c572a5-bcd3-4455-a6f3-3914a0d494a8	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:07:01.172189	8

After the test we can see the listing quantity is 5, as expected the sum across orders is 35, this is made of a combination of some new orders and the only order we did not cancel which was being edited multiple times.

	listing_id	sg_id	create_timestamp	price	description	condition	quantity
1	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	72b57bc4-8a51-4270-8841-1d5dc926f3bd	2022-10-20 05:43:52.703691	8	Brownie	NEW	5

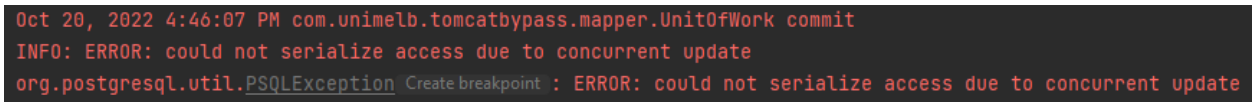


	sum
1	35

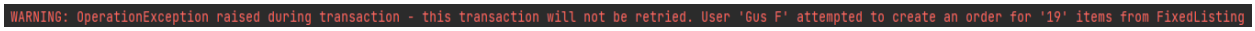


	order_id	username	listing_id	create_timestamp	quantity
1	12092002-6ae5-479a-bb3d-da1b7a033334	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 16:43:59.105408	19
2	aad88296-909b-4fdd-b104-619633e2e926	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 16:43:59.191908	6
3	d4c572a5-bcd3-4455-a6f3-3914a0d494a8	Gus F	98b7e5b5-78bf-4807-9af5-f5ca1db51fbe	2022-10-20 05:43:52.733238	10

We also saw the expected serialisation rollbacks and invalid quantity rollbacks



```
Oct 20, 2022 4:46:07 PM com.unimelb.tomcatbypass.mapper.UnitOfWork commit
INFO: ERROR: could not serialize access due to concurrent update
org.postgresql.util.PSQLException Create breakpoint : ERROR: could not serialize access due to concurrent update
```



```
WARNING: OperationException raised during transaction - this transaction will not be retried. User 'Gus F' attempted to create an order for '19' items from FixedListing
```

Conclusion

This set of tests is very extensive and highlights the key factors in fixed listing and order concurrency. This part of our system is most vulnerable to concurrency issues given the interplay between the 2 tables and the risk for lost update. Through these tests we can show that all possible interleavings and combinations result in no concurrency issues and our theoretical concurrency protection works as intended.

4.2.6. Create a Listing

Chosen Methods Rationale

Since creation and deletion of fixed price and auction listings are so similar, only auction listing tests are discussed in detail here.

The tests for this operation have assertions and also call other service methods in order to force concurrency issues to occur. To allow some of the assertions, the boolean return value returned by the call to saveNewAuctionListing was used. The same auction and fixed listing parameters are used for all tests. At some point in all these tests, listings are repeatedly created using these same parameters, however their IDs are distinct. This is allowed by our business rules. This simplifies assertions, since querying the database for a listing with the chosen description will return all the listings that have been saved in a single unit test.

Importantly, and as for most other unit tests, the database must be reset between each test.

Test Case 1 - Create Auction Listings

This test creates 50 threads to each create a listing with the same parameters except for the listing id, and starts them all at the same time.

Expected Outcomes:

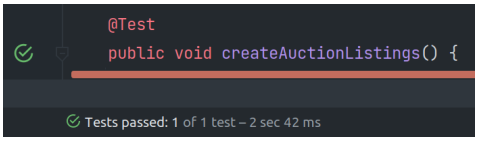
- All listings will be successfully created.
- The number of new listings in the db will be the same as the number of successes we recorded from the call to the listing method.

The following image shows the relevant assertions:

```
assertEquals(successes.size(), NUMBER_OF_THREADS);  // This is just a sanity check.

// All listings should have been created successfully.
assertEquals(NUMBER_OF_THREADS, successes.stream().filter(pred -> pred.equals(true)).count());

// We want the number of new listings in the db to be the same as the number of successes we recorded.
assertEquals(successes.stream().filter(pred -> pred.equals(true)).count(), listings.size());
```

Outcomes:

This test passed.



Test Case 2 - Create Fixed Listings

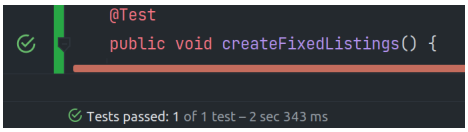
See description for test 1.

Expected Outcome:

See expected outcome for test 1.

Outcomes:

This test passed.



Test Case 3 - Create Auction Listings but Delete Seller Group

This test is quite similar to the last two, however we create a thread which deletes the seller group the user is trying to associate their new listing with, hopefully while they're creating one of the 50 new listings. The idea is to allow some of the listings to be created before the seller group is then deleted, and after that all of the listing creations should fail.

Expected Outcomes:

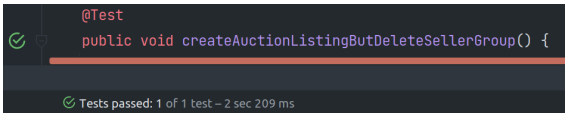
- Deletion of the seller group should be successful.
- All the listings should be deleted, due to cascading deletes.
- Any attempts to create a listing after the seller group is deleted should fail.

The following image shows the relevant assertions:

```
assertEquals(createSuccesses.size(), NUMBER_OF_THREADS);  // This is just a sanity check.

// We should have been able to delete the chosen seller group.
assertEquals(deleteSuccesses.peek(),  actual: true);

assertEquals(listings.size(),  actual: 0);
```

Outcomes:

This test passed.



Test Case 4 - Create Fixed Listings but Delete Seller Group

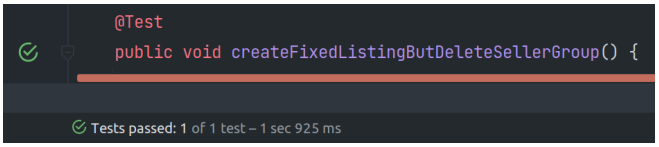
See description for test 3.

Expected Outcomes:

See expected outcome for test 3.

Outcomes:

This test passed.



Test Case 5 - Create Auction Listings but User Removed From Seller Group

This test is almost identical to test cases 4 and 5, except that instead of deleting the seller group, we remove the user from it.

Expected Outcomes:

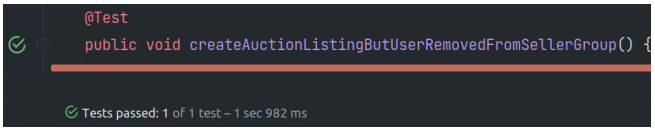
- Each thread should at least try to create a listing.
- The user should successfully be removed from the seller group.
- The number of new listings in the database should be the same as the number of successful listing creation operations.

The following image shows the relevant assertions:

```
assertEquals(createSuccesses.size(), NUMBER_OF_THREADS);  // This is just a sanity check.

// We should have been able to remove the chosen user from the chosen seller group.
assertEquals(removeSuccesses.peek(),  actual: true);

// We want the number of new listings in the db to be the same as the number of successes we recorded.
assertEquals(createSuccesses.stream().filter(pred -> pred.equals(true)).count(), listings.size());
```

Outcomes:

This test passed.



Test Case 6 - Create Fixed Listings but User Removed From Seller Group

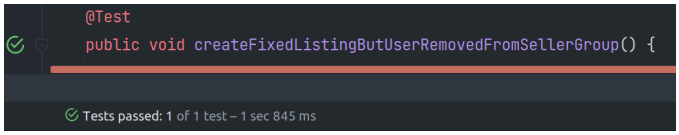
See description for test 5.

Expected Outcomes:

See description for test 5.

Outcomes:

This test passed.



4.2.7. Delete a Listing

Chosen Methods Rationale

This idea behind this test is to see if our roll back of interleaving deletion and creation of listings works properly.

The setup of these tests is rather similar to the tests for creating listings. The implementation is actually rather involved, but the principle is simple. It creates a bunch of listings as setup, then it deletes all of them at the same time, while also trying to remove the user from the associated seller group once. The idea is for some listings to be successfully deleted before the user is removed from the seller group, and for all subsequent deletion of listings to fail.

Importantly, and as for most other unit tests, the database must be reset between each test.

Test Case 1 - Delete Auction Listings but User Removed From Seller Group

Thing

Expected Outcomes:

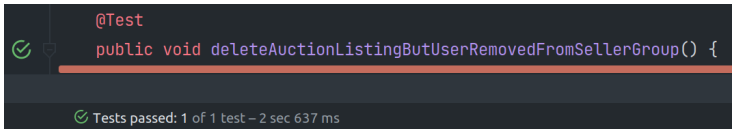
- The number of created listings minus the number of deleted listings should equal the number of leftover listings in the db at the end.

The following image shows the relevant assertions:

```
// Sanity check that all the creates were successful
assertTrue(createSuccesses.stream().allMatch(b → b.equals(true)));
assertEquals(NUMBER_OF_THREADS, createSuccesses.size());
int numCreatedSuccessfully = createSuccesses.size();

// Sanity check that the removal from the seller group was successful.
assertTrue(removeUserSgMappingSuccesses.peek());

Session.startSession();
List<AuctionListing> remainingListings = new AuctionListingMapper().findByDescription(auctionParams.getDescription());
Session.closeSession();

// Now we do the important part:
// Check that the number of created listings minus the number of deleted equals the number of remaining.
int numDeletedSuccessfully = (int) listingDeleteSuccesses.stream().filter(b → b.equals(true)).count();
int numRemainingListings = remainingListings.size();
System.out.println("numCreatedSuccessfully=" + numCreatedSuccessfully + " numDeletedSuccessfully=" + numDeletedSuccessf
assertEquals( expected: numCreatedSuccessfully - numDeletedSuccessfully, numRemainingListings);
```

Outcomes:

This test passed.



Test Case 2 - Delete Fixed Listings but User Removed From Seller Group

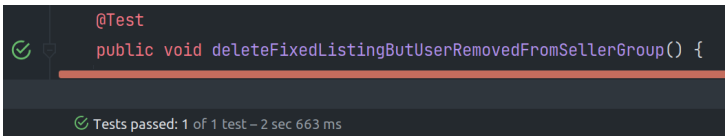
See description for test 1.

Expected Outcomes:

See expected outcome for test 1.

Outcomes:

This test passed.



4.2.8. Create a Seller Group

Chosen Methods Rationale

Due to the simplicity of this transaction testing is only being used as a sanity check. A single JUnit test has been created to ensure that the transaction behaves as expected when it is being executed by many different users.

Test Case 1 - Repeated Seller Group Creates

This test involves using 50 threads to create 50 seller groups of the same name by the same user.

Expected Outcome:

The first thread that commits should succeed and create the seller group, with all other threads being rejected because a seller group of the same name already exists thus violating the database constraint that all seller group names must be unique.

Outcomes:

```
Oct 20, 2022 10:59:04 PM com.unimelb.tomcatbypass.service.SellerGroupService$1 doTransactionOperations
INFO: SellerGroup to save to db: SellerGroup(sgId=9b985a06-033d-4c34-9de4-cd95a2f8980e, name=Best Seller Group,
Oct 20, 2022 10:59:04 PM com.unimelb.tomcatbypass.service.SellerGroupService$1 doTransactionOperations
INFO: SellerGroup to save to db: SellerGroup(sgId=a161fb84-bd2e-45db-81b0-0d87254e9008, name=Best Seller Group,
Oct 20, 2022 10:59:04 PM com.unimelb.tomcatbypass.service.SellerGroupService$1 doTransactionOperations
INFO: SellerGroup to save to db: SellerGroup(sgId=d4134c3a-c12b-4a39-ac82-24270c8b27e9, name=Best Seller Group,
```

```
INFO: ERROR: duplicate key value violates unique constraint "seller_group_name_key"
  Detail: Key (name)=(Best Seller Group) already exists.
org.postgresql.util.PSQLException: ERROR: duplicate key value violates unique constraint "seller_group_name_key"
  Detail: Key (name)=(Best Seller Group) already exists.
```

	sg_id	name
1	def8570f-358d-4d56-85da-b0f9d3440fc6	The Cartel
2	011945c2-a2f0-4d71-b6df-22a0467dc013	Bargain Deals
3	72b57bc4-8a51-4270-8841-1d5dc926f3bd	Silk Road
4	30aaccf5-8223-4db6-b76c-fd811c2423c7	Fantastic Faces
5	a866b2db-b369-42c5-9768-28d330d3040a	Cheap Eats
6	9b985a06-033d-4c34-9de4-cd95a2f8980e	Best Seller Group

As expected, the fastest thread is added to the database and all remaining threads are rejected because a seller group of the same name already exists. The seller group with sg_id='9b985a06-033d-4c34-9de4-cd95a2f8980e' was the first to show in the logs and it is the only seller group which has successfully saved to the database.

4.2.9. Delete a Seller Group

Chosen Methods Rationale

Due to the simplicity of this transaction testing is only being used as a sanity check. A single JUnit test has been created to ensure that the transaction behaves as expected when it is being executed by many different users.

Test Case 1 - Repeated Seller Group Deletes

This test involves using 50 threads to delete 50 seller groups of the same name by the same user.

Expected Outcome:

The fastest thread to commit should successfully delete the seller group, all other threads should be rejected on the basis that either the user doesn't have permission because they aren't in the seller group as it has been deleted or they should be aborted by the Unit of Work because the seller group object that is being committed is null.

Outcomes:

	sg_id	name
1	def8570f-358d-4d56-85da-b0f9d3440fc6	The Cartel
2	011945c2-a2f0-4d71-b6df-22a0467dc013	Bargain Deals
3	72b57bc4-8a51-4270-8841-1d5dc926f3bd	Silk Road
4	30aaccf5-8223-4db6-b76c-fd811c2423c7	Fantastic Faces
5	a866b2db-b369-42c5-9768-28d330d3040a	Cheap Eats

	sg_id	name
1	011945c2-a2f0-4d71-b6df-22a0467dc013	Bargain Deals
2	72b57bc4-8a51-4270-8841-1d5dc926f3bd	Silk Road
3	30aaccf5-8223-4db6-b76c-fd811c2423c7	Fantastic Faces
4	a866b2db-b369-42c5-9768-28d330d3040a	Cheap Eats

```
WARNING: OperationException raised during transaction - this transaction will not be retried. username=AppUser does not have permission to
 delete seller group with ID=def8570f-358d-4d56-85da-b0f9d3440fc6
Oct 20, 2022 11:11:57 PM com.unimelb.tomcatbypass.mapper.UnitOfWork abort
INFO: Aborting empty unit of work, not a big deal.
Oct 20, 2022 11:11:57 PM com.unimelb.tomcatbypass.utils.ManagedTransaction executeTransaction
WARNING: OperationException raised during transaction - this transaction will not be retried. username=AppUser does not have permission to
 delete seller group with ID=def8570f-358d-4d56-85da-b0f9d3440fc6
Oct 20, 2022 11:11:57 PM com.unimelb.tomcatbypass.mapper.UnitOfWork abort
INFO: Aborting empty unit of work, not a big deal.
```

The test behaves as expected. The first thread deletes the seller group and the remaining threads have no effect on the basis that they either don't have permission since the seller group doesn't exist or because the Unit of Work commit aborted since the seller group to delete is null.

4.2.10. Add User to a Seller Group

Chosen Methods Rationale

Due to the simplicity of this transaction testing is only being used as a sanity check. A single JUnit test has been created to ensure that the transaction behaves as expected when it is being executed by many different users.

Test Case 1 - Repeated Mapping Inserts

This test involves using 50 threads to add a user 50 times to the same seller group by the same user.

Expected Outcome:

The first thread should add the user to the seller group and the remaining threads should be rejected by the database because the record already exists.

Outcomes:

	username	sg_id
1	AppUser	def8570f-358d-4d56-85da-b0f9d3440fc6
2	AppUser	011945c2-a2f0-4d71-b6df-22a0467dc013
3	Admin	72b57bc4-8a51-4270-8841-1d5dc926f3bd
4	Admin	30aaccf5-8223-4db6-b76c-fd811c2423c7
5	Admin	a866b2db-b369-42c5-9768-28d330d3040a
6	Jimmy	def8570f-358d-4d56-85da-b0f9d3440fc6

	username	sg_id
1	AppUser	def8570f-358d-4d56-85da-b0f9d3440fc6
2	AppUser	011945c2-a2f0-4d71-b6df-22a0467dc013
3	Admin	72b57bc4-8a51-4270-8841-1d5dc926f3bd
4	Admin	30aaccf5-8223-4db6-b76c-fd811c2423c7
5	Admin	a866b2db-b369-42c5-9768-28d330d3040a

```
INFO: ERROR: duplicate key value violates unique constraint "user_sg_mapping_pkey"
  Detail: Key (username, sg_id)=(Jimmy, def8570f-358d-4d56-85da-b0f9d3440fc6) already exists.
```

As expected, the user is deleted by the first thread and the other 49 threads are rejected because they result in the exact same record being inserted.

4.2.11. Remove User from a Seller Group

Chosen Methods Rationale

Due to the simplicity of this transaction testing is only being used as a sanity check. A single JUnit test has been created to ensure that the transaction behaves as expected when it is being executed by many different users.

Test Case 1 - Repeated Mapping Deletes

This test involves using 50 threads to remove a user from a seller group 50 times

Expected Outcome:

This should delete the mapping from the database.

Outcomes:

	username	sg_id
1	AppUser	def8570f-358d-4d56-85da-b0f9d3440fc6
2	AppUser	011945c2-a2f0-4d71-b6df-22a0467dc013
3	Admin	72b57bc4-8a51-4270-8841-1d5dc926f3bd
4	Admin	30aaccf5-8223-4db6-b76c-fd811c2423c7
5	Admin	a866b2db-b369-42c5-9768-28d330d3040a
6	Jimmy	def8570f-358d-4d56-85da-b0f9d3440fc6

	username	sg_id
1	AppUser	def8570f-358d-4d56-85da-b0f9d3440fc6
2	AppUser	011945c2-a2f0-4d71-b6df-22a0467dc013
3	Admin	72b57bc4-8a51-4270-8841-1d5dc926f3bd
4	Admin	30aaccf5-8223-4db6-b76c-fd811c2423c7
5	Admin	a866b2db-b369-42c5-9768-28d330d3040a

This test case behaves as expected, as shown above the user is removed from the seller group by the first thread to commit. The remaining threads have no effect on the database because the user has already been removed from the seller group.