Music Event System

SWEN90007 Software Design and Architecture

Project: Part 3

Software Architecture Report

TEAM NAME - **MusicBandTeam**

Team Members	Student ID
Bingqing Zhang	1377017
Xin Xiang	1294725
Dhakshayani Perumal	1184165
Vivien Guo	1173459

• Updated Class Diagram

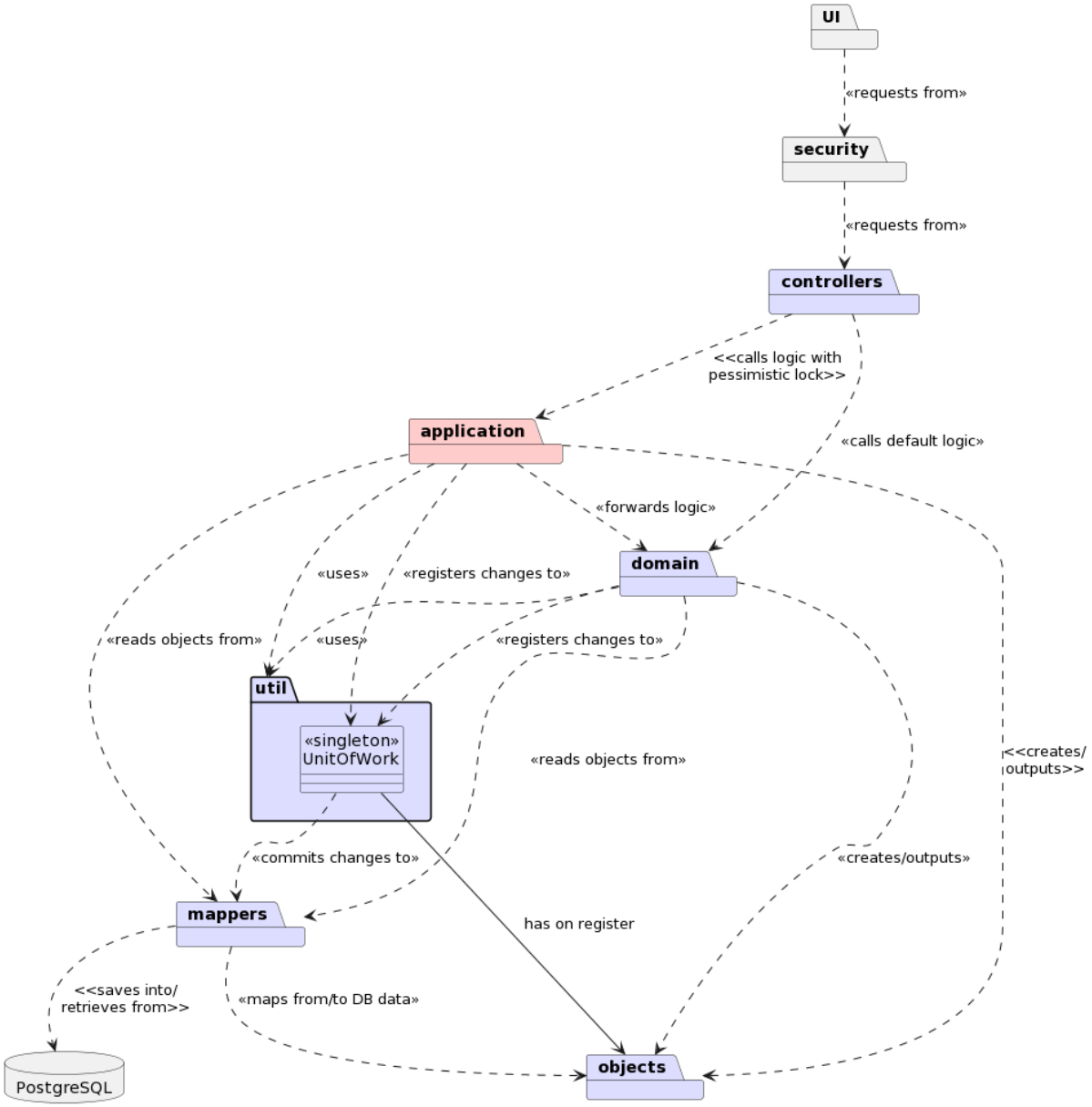
To control the diagram sizes for legibility, the following items are omitted:

- Overridden methods in implementations & subclasses
- Trivial constructors, getters and setters
- **Method updates that do not relate to concurrency patterns**
- Attributes/methods from **certain classes that do not have changes relating to concurrency patterns**
- Custom exceptions and **any unused classes**

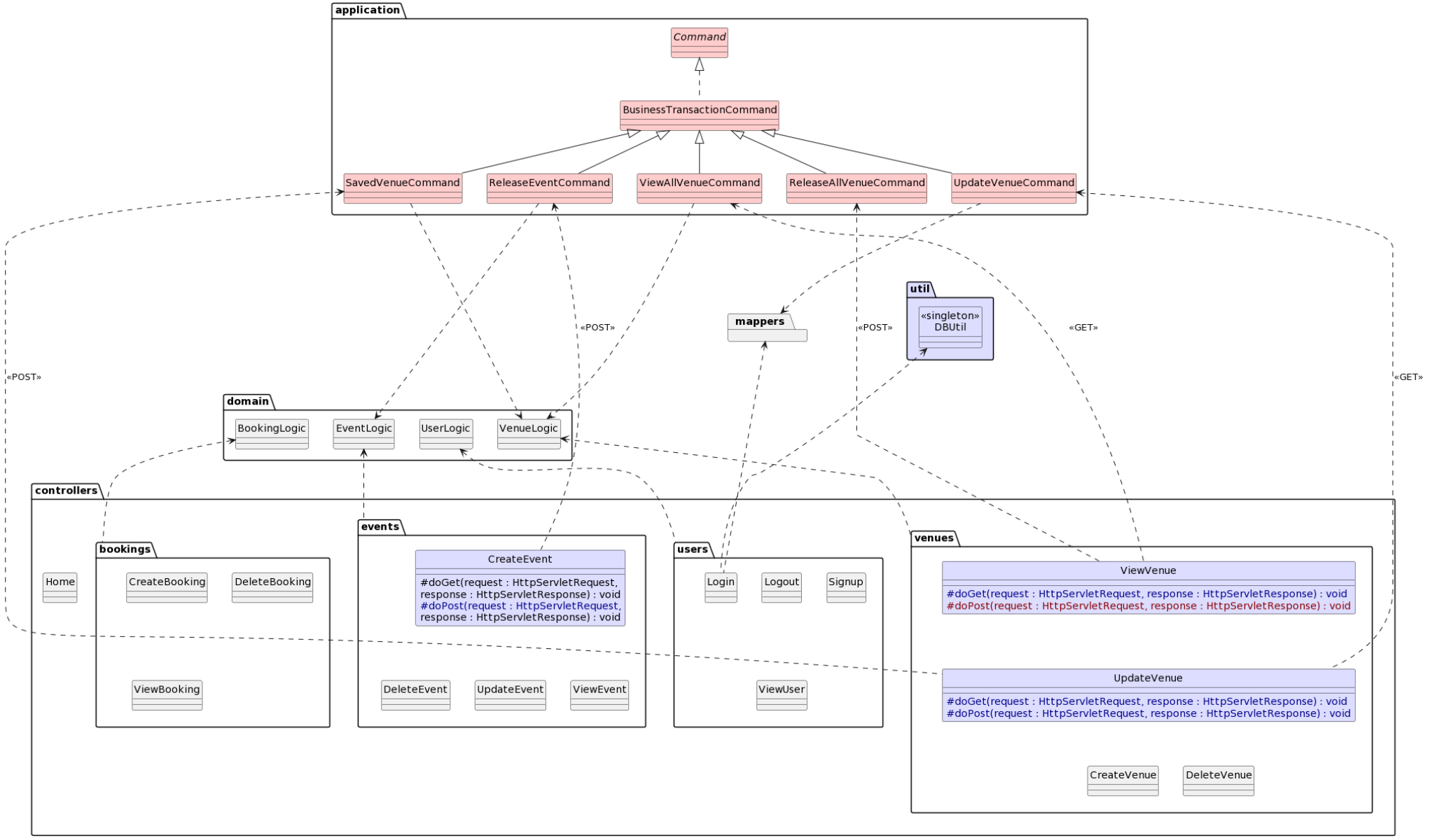
New classes are in red highlight, and updated classes are in blue highlight. New attributes/methods are in red text, and updated attributes/methods are in blue text.

The whole class diagram is too big to be viewed all at once, so we split it up into the following:

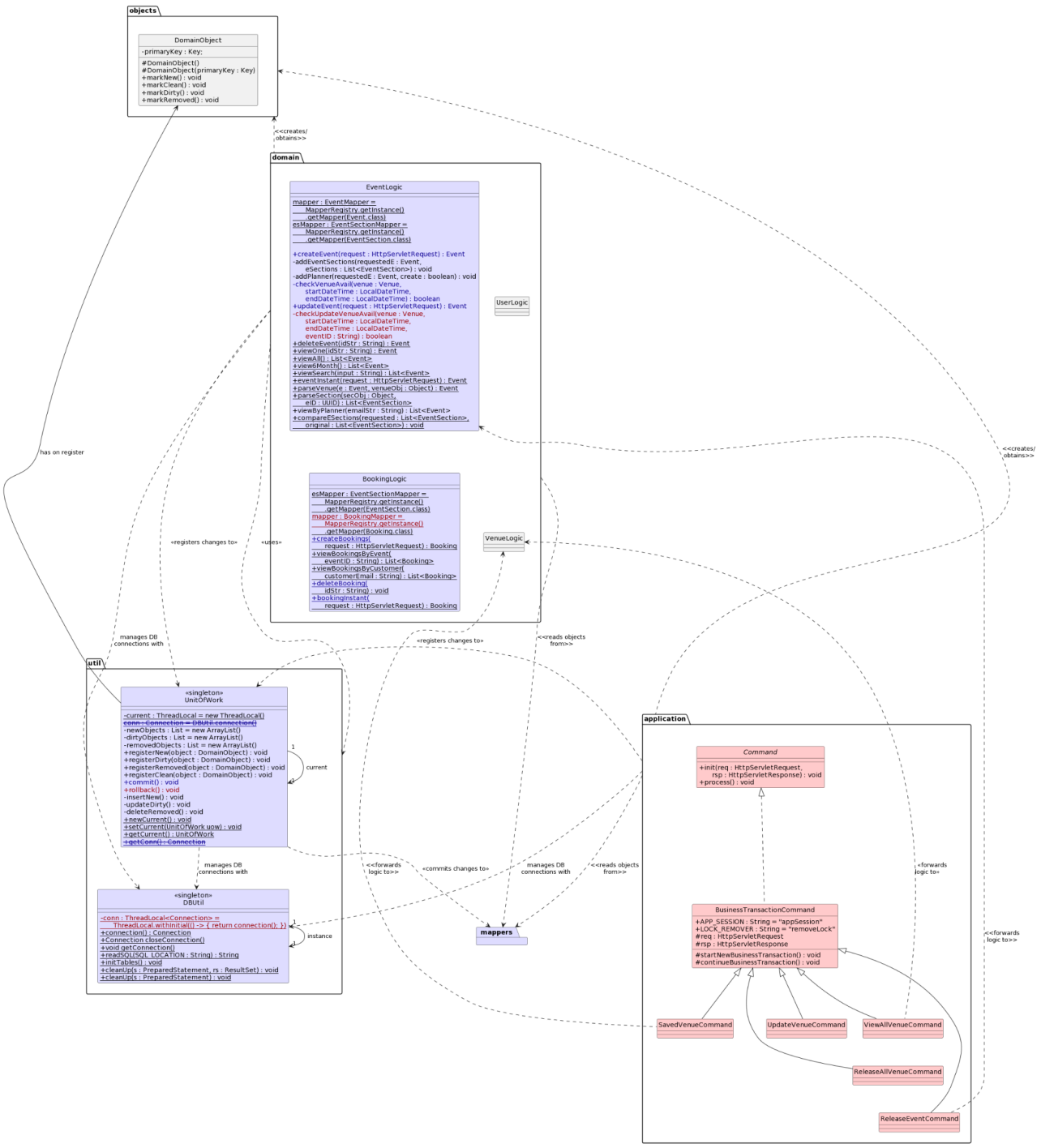
• Broad overview



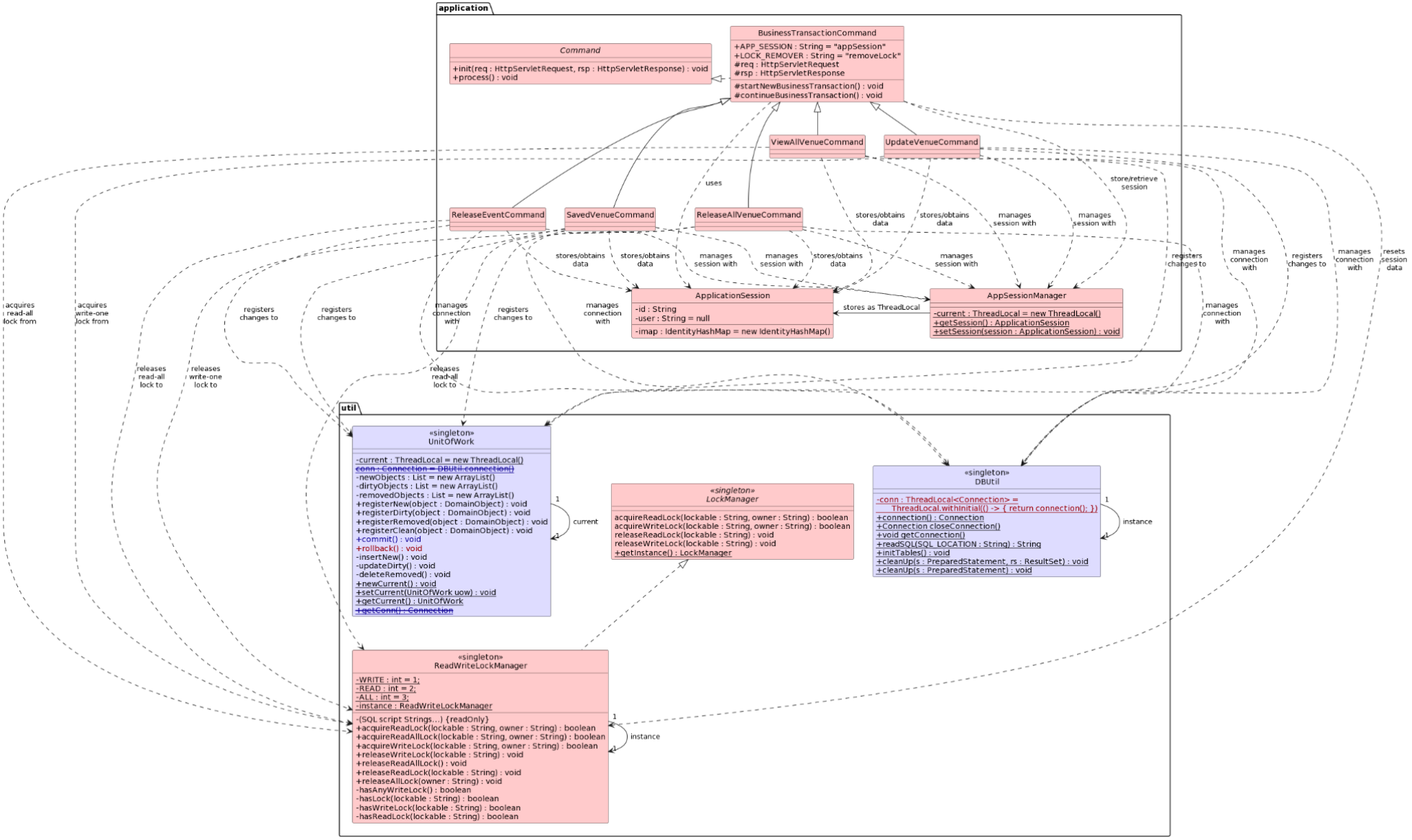
- Controllers -> Domain/Application



- Application logic, domain logic & unit of work

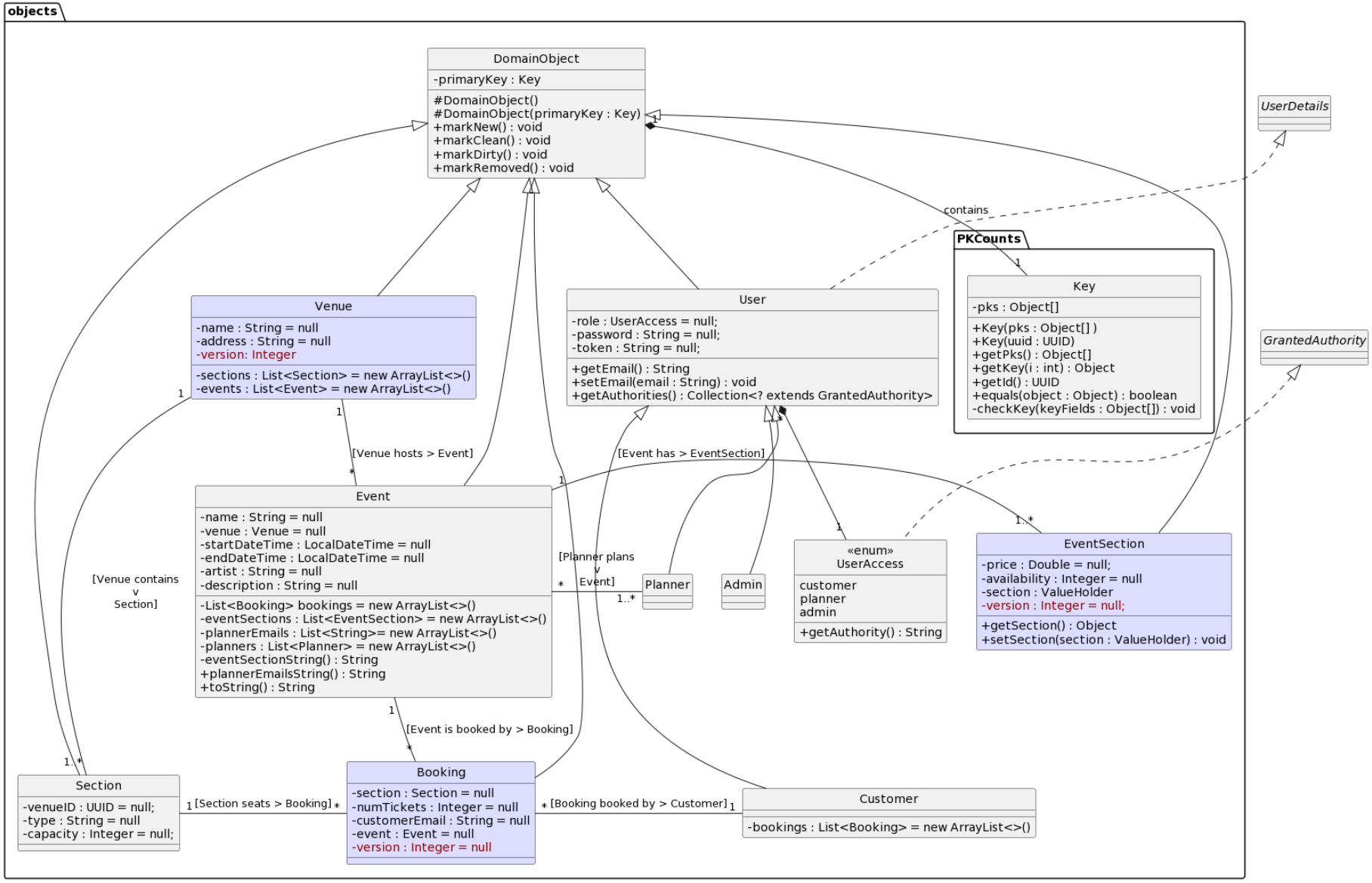


- Application session & pessimistic lock

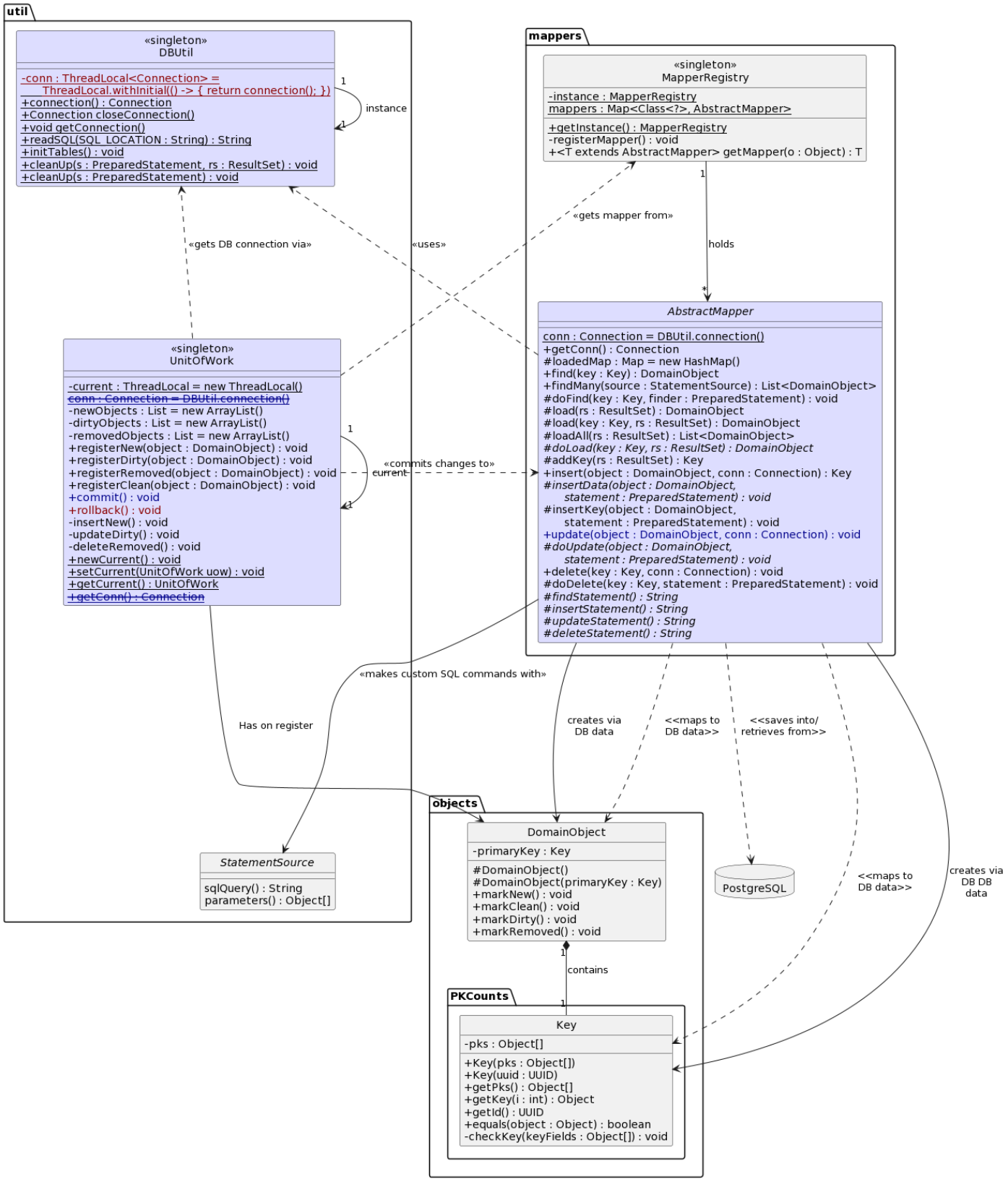


- Domain objects

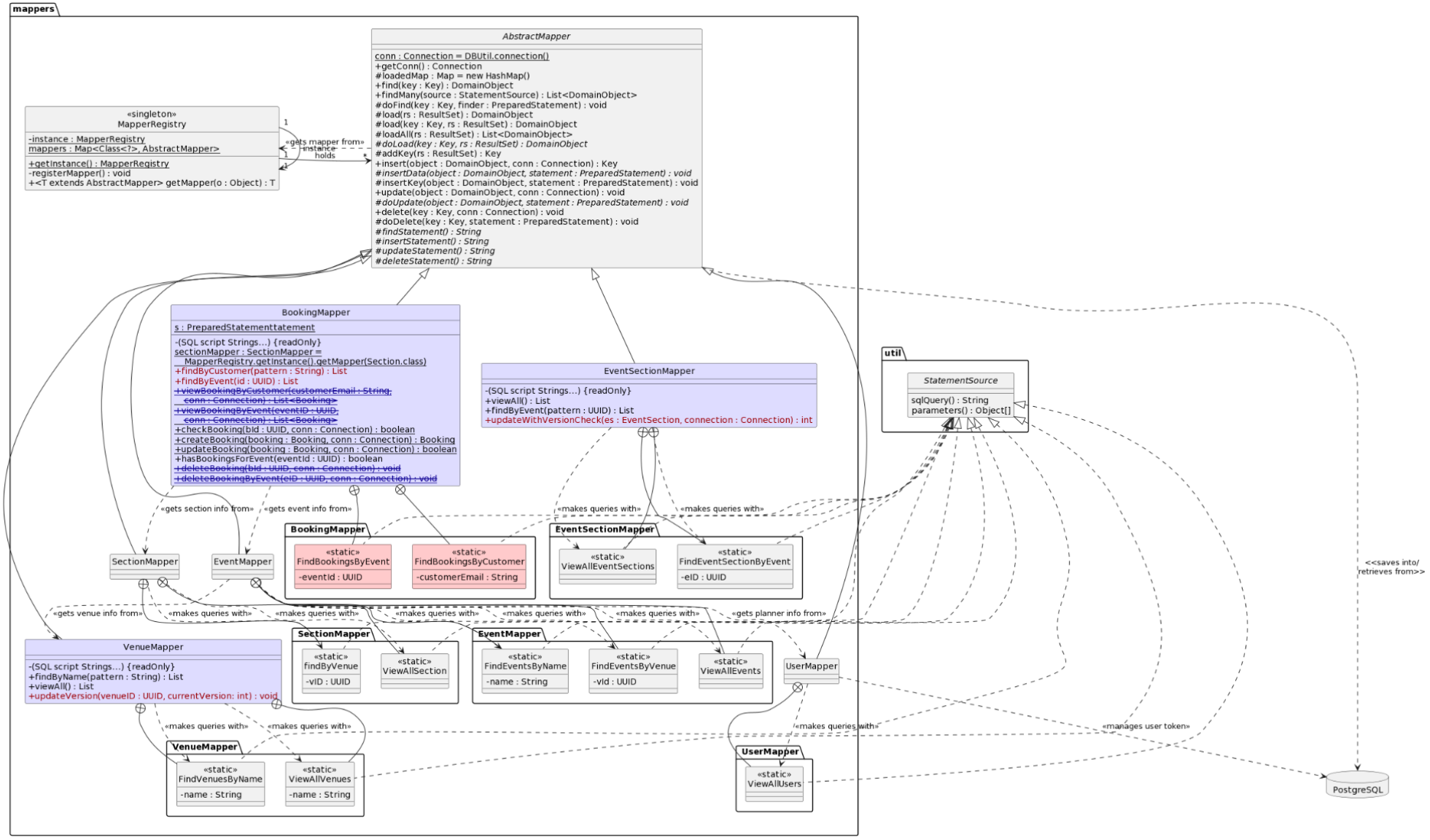
Note that UserDetails and GrantedAuthority refer to the Spring Security classes. Note that `version` on Booking is a temporary field that refers to the current EventSection number recorded in the frontend.



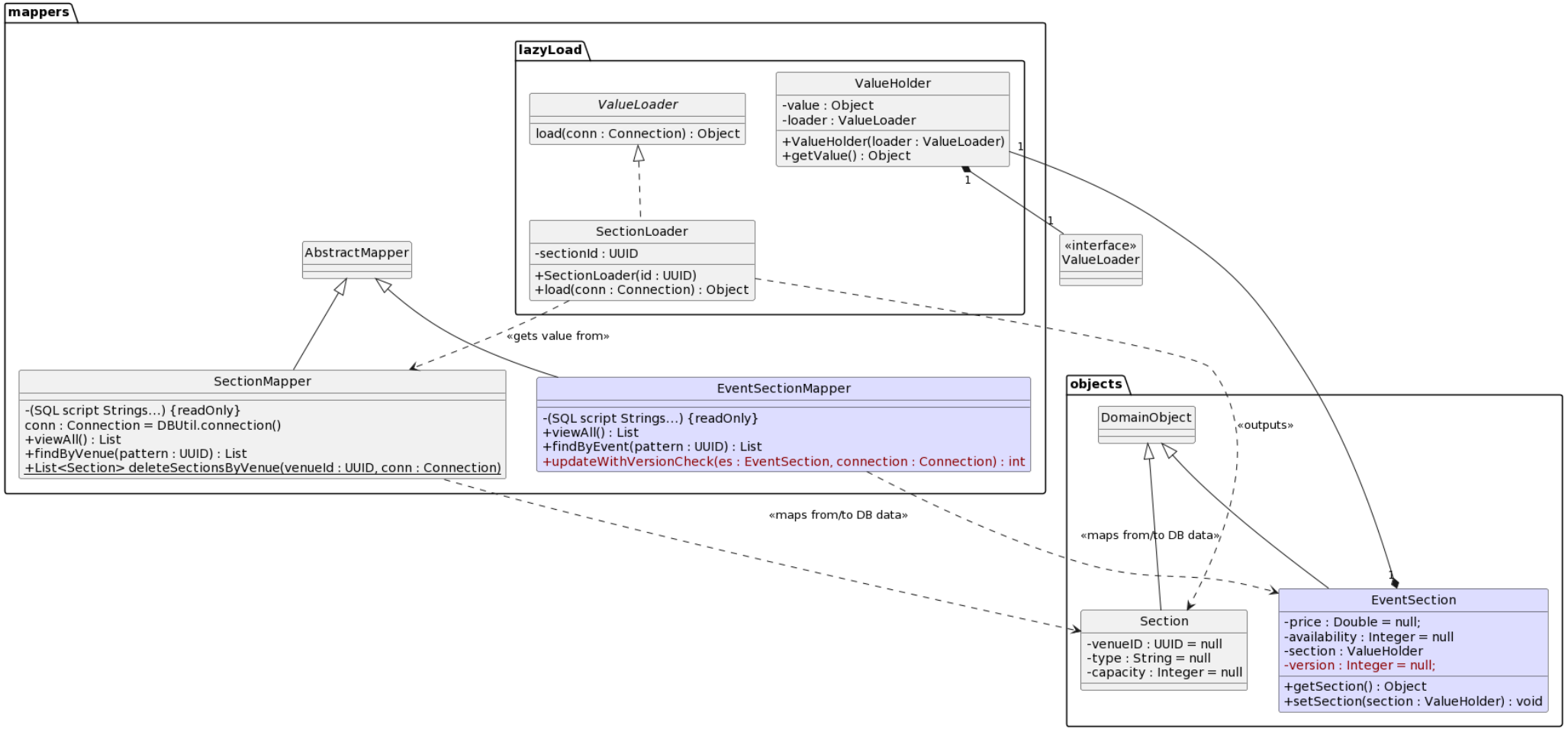
- Unit of work & abstract mapping process



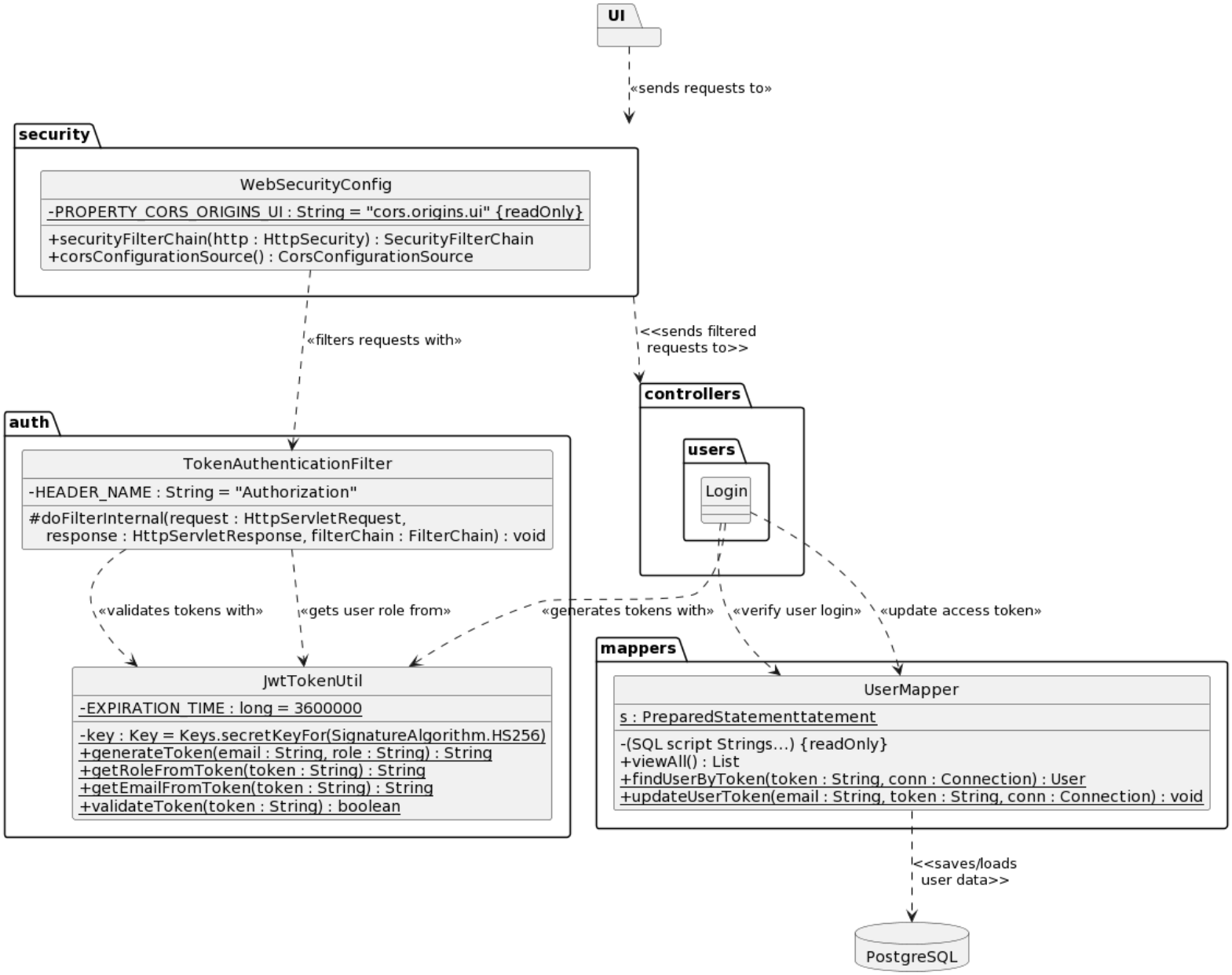
- Specific mappers



• Lazy loading & EventSectionMapper (**Lazy loading logic unchanged**)



- Authentication (**Unchanged**)



● Concurrency Discussion

When our application is used by multiple users at once, we may encounter certain concurrency issues.

We identify and resolve the following concurrency issues:

- Simultaneous customer creations of Bookings at the same EventSection
- Administrator edits Venue (and its associated Sections) during creations of Events
- Planner edits Event (and its associated EventSections), while customers create Bookings.

*: This is not strictly a concurrency issue, but it leads to the same effect as the inconsistent read in the “during Booking creation” case. This will be elaborated on in its issue description.

A list of non-issues is also attached at the end of this report.

• Simultaneous Booking creations

Issue description

The creation of each Booking updates the associated EventSection through reducing its availability by the number of tickets. If two customers simultaneously try to book ticket/s to the same section at the same event (i.e. the EventSection), this leads to the following:

- The second request overwrites the first request’s change, resulting in a lost update of the reduction in the ticket availability.
- The system is thus unable to properly track how many tickets have been sold at the EventSection (without constantly querying from the Bookings table).
- This results in the system overestimating an EventSection’s availabilities and overselling tickets.

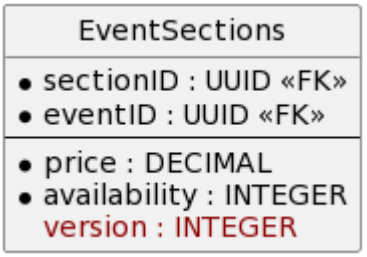
Pattern(s) and/or concurrency mechanisms used

We implement the **Optimistic Offline Lock** to counter this issue for the following reasons:

- Booking creation is extremely simple; the only work done besides the actual Booking creation is checking if the corresponding EventSection has tickets left. Therefore, booking creations are trivial to the application’s functionalities, and lost work on it is both acceptable and easily recoverable.

Implementation details

- In the database we add the “version” field to our EventSection data, (which is defaulted to 1 upon creation).
- Fig 1: addition of the field “version” to EventSections table in the database



- When performing use cases that update EventSections (i.e. Event edits and Booking creation/deletion), we send via the frontend the version(s) for the modified EventSection(s).
- When our BookingLogic parses the Booking information, it temporarily stores the EventSection version in the Booking object. This allows it to do an initial comparison against the current version in the corresponding row of the EventSection database. If the version from the frontend is outdated, we throw an `OptimisticLockingException`.
- Our EventSectionMapper is also modified to accommodate this logic with an extra check:
    - When an EventSection row is being updated, we configure the SQL query string to set the version as (1 + the version) passed from the frontend.
    - When an EventSection row is being updated, we also configure the SQL query string through the WHERE clause, so that it only updates a row that has the same version as the one sent by the frontend.
- When we perform the query, EventSectionMapper tracks the number of rows that have been updated. This number being 0 means that the query is unsuccessful, due to the version on the user's end being lower than the current value in the database.
- If the frontend's version is lower than the current version:
    - This means that the information viewed by the user is outdated, and that a newer version of the EventSection information is stored in the database.
    - We then throw an `OptimisticLockingException` error in the EventSectionMapper, so that no sub-transactions for this request would be committed.
    - The frontend will receive an error message and refresh the page, so that a more recent version of the data is displayed.
- If the current version in the database is the same as the version sent from the frontend:
    - This means that this request is based on the most current data.
    - The whole transaction is committed. A success message is sent to frontend.
- We moved the responsibility of holding onto the database connection out of `UnitOfWork`, and instead into the singleton DBUtil.
    - DBUtil has also been modified to be thread-safe via using `ThreadLocal`.
    - This allows `BookingLogic` to manage database connections outside of `UnitOfWork`, so it can directly rollback on `OptimisticLockingException`s.

Sequence Diagram

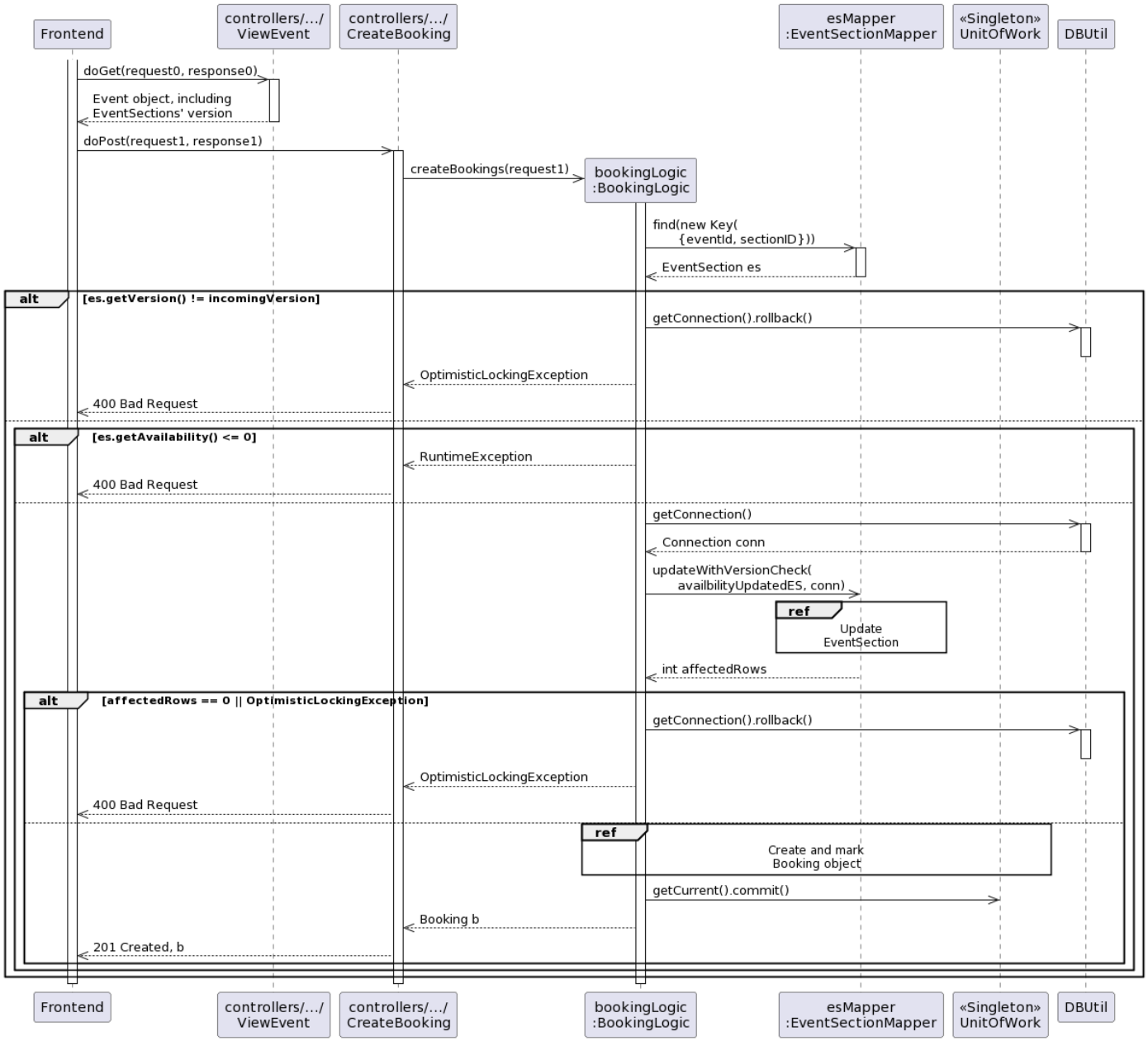


Fig 2: Sequence diagram of booking creation with optimistic lock. Retrieving Event object and other irrelevant logic are omitted.

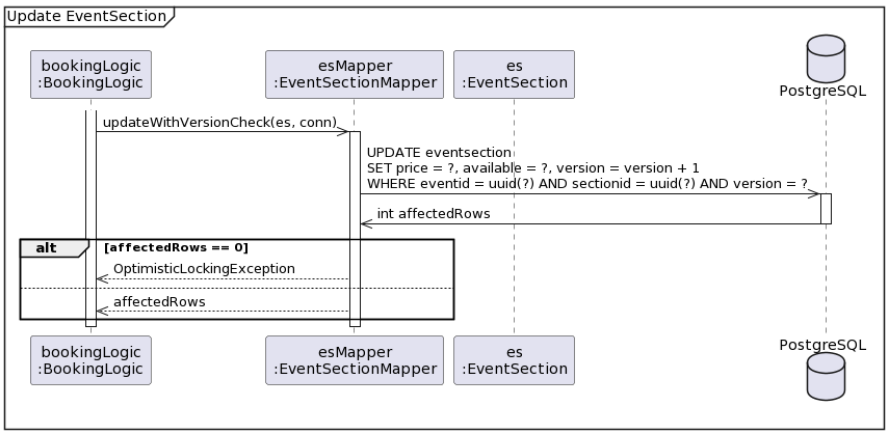


Fig 3: Sequence diagram of the subprocess Update EventSection, during booking creation with optimistic lock. Retrieving Event object and other logic irrelevant to the optimistic lock is omitted

Testing strategy and outcome

TC001	Simultaneous booking creation
TestType	Functional
Execution Type	Manual
TestCase Description	If two customers simultaneously try to book ticket/s to the same section at the same event (i.e. the EventSection).
Steps to reproduce.	1. Customer1 login to the system. 2. Customer2 login to the system. 3. Customer1 view the event1 and books 2 ticket to section1. 4. Customer2 concurrently views the event1 and books 2 tickets to section1.
Preconditions and Sample Input Data	Customer1 - Name say John who is already registered in the system. Customer2 - Name say Alice, who is already registered in the system. EventName - StartNite ,which is already created in the system SectionName - VIP
List of scenarios.	1. John view the “StarNite” event and select 2 tickets . 2. Alice view the “StartNite” event and select 2 tickets. 3. John click “Book tickets” and Alice click “Book tickets”.
Test Output	1. John booking successful. 2. Alice if the event has remaining tickets, then alice will get the tickets or if it is last ticket , then error message is showed to Alice.

• Editing Venue+Sections & creating Event

Issue description

In our business domain, Venue and its associated Sections are depended upon (referenced) by all other objects besides Users; thus, when performing Event-related use cases, the user will have to base their business decision on a read of the Venue’s and its Sections’ data.

Thus, updating them will lead to the information originally read being inconsistent, thus causing the following problems:

- An Event may be set at a venue with an undesired location (e.g. planner creates Event at a Venue that they initially read to be at location A, but has changed to be at location B since the read.)

- Or, an Event's EventSection data may be inconsistent with their corresponding Section data (e.g. EventSection allowing for more tickets than what is available after the Section update).

Pattern(s) and/or concurrency mechanisms used

We implement the **Pessimistic Offline Lock**, in particular the **read-write lock**, to prevent this issue. This is done so for the following reasons:

- Pessimistic:

    - Creation and Edits of events involve retrieving all Events at the same Venue, in order to avoid overlaps in event running time. This is a huge amount of work that can be avoided by using a pessimistic lock instead of an optimistic one.

    - Moreover, although Venue edits are relatively rare (both because of the business logic and because there is only one admin account), many Events will be affected by a concurrent Venue/Section update, since changes in any Venue affects their ability to make business decisions. This makes the operation even more costly.

- Read-write lock

    - Updates (i.e. writes) to Venue and Section must lock the data, as otherwise the inconsistent read issues outlined previously will still occur.

    - Since Venue and Section information will be viewed by a lot of users (both customers and planners), it is essential that concurrent reads are allowed to keep the liveness of the system.

    - Therefore, a read-write lock is our only option.

Implementation details

This report splits up the implementation logic into two sections: Session Setup, and Lock Management.

Note that there are no use cases where Sections are viewed or edited separately from their Venue/s, so we treat Venues and Sections together as an aggregate and call them Venue for readability.

Session Setup

- The flow of certain use cases may involve multiple requests to the backend, and thus each lock will need to be matched to a user. We identify each user flow by its browser session, which we generate by calling `request.getSession()` command on a `HttpServletRequest`, via the `Command` interface described later.
    - The raw information from the browser is processed as an `ApplicationSession`, which stores the session ID, user's identifier, and an IdentityHashMap for caching objects loaded or created during the transaction. This allows the locking process to identify each user's browser session, on top of helping different classes easily store and retrieve data (e.g. The controller `ViewVenue` (on a GET request) retrieving

Venue data stored within and outputting it to the frontend, after a `ViewAllVenuesCommand` has put data in it).
    - The `AppSessionManager` class is created to allow easy access to the `ApplicationSession`. The `ApplicationSession`s are kept thread-safe by having each `current` instance stored as a `ThreadLocal`.
- We create the `Command` interface to abstractify and facilitate the acquisition and release of locks. It comes with the following methods:
    - `init(HttpServletRequest req, HttpServletResponse rsp)`, which stores the HTTP request and response objects within the class itself, so as to support processing
    - `process()`, which runs the actual transaction by either:
        - Acquiring a lock for a session, performing business logic OR
        - Performing business logic for a session, then releasing the lock
- The `Command`s we use extend from the `BusinessTransactionCommand`, a class created to handle the transactions we make in our application. In addition to the base `Command` functionalities, it also has the following:
    - Constant key names for storage and retrieval of session information
    - `startNewBusinessTransaction()`, which clears any leftover locks from a previous session and initiates a new session
    - `continueBusinessTransaction()`, which retrieves an existing session
- We moved the responsibility of holding onto the database connection out of `UnitOfWork`, and instead into the singleton DBUtil.
    - DBUtil has also been modified to be thread-safe via using `ThreadLocal`.
    - This allows our `Command`s to rollback database changes when they detect `ConcurrencyException`s, which is done from outside of `UnitOfWork`.

Lock Management

- We consider the lock acquisition/release process for these two use cases:
    - Admin edits Venue
        1. Admin browses through Venues: Acquires read lock for all Venue rows
        2. Admin entering an edit page for a particular Venue: No longer needs all Venue data, releases read lock for all rows
        3. Admin editing a Venue: Acquires write lock for this particular Venue row
        4. Admin submitting a Venue edit: Releases write lock for this Venue row
    - Planner creates Event
        1. Planner choosing venue for event: Acquires read lock for all Venue rows
        2. Planner submitting event creation: Having already selected a Venue, releases read lock for all Venue rows
- To avoid deadlock caused by circular hold-and-waits, we keep the acquisition of resources completely atomic, by having the controllers requesting for lock management via `Command`s upon the very start of a request, instead of doing so in the middle of processing business logic:
    - **Acquire Venue read-all lock**: This is called via the doGet method of our ViewVenue controller. It creates a `ViewAllVenueCommand` object.
    - **Release Venue read-all lock (in Admin edits Venue)**: This is called via the doPost method of our ViewVenue controller. It creates a `ReleaseAllVenueCommand`.

    - **Release Venue read-all lock (in Planner creates Event)**: This is called via the doPost method of our CreateEvent controller, which creates a `ReleaseEventCommand`.
    - **Acquire Venue write-one lock**: This is called via the doGet method of our UpdateVenue controller, which creates a `UpdateVenueCommand`.
    - **Release Venue write-one lock**: This is called via the doPost method of our UpdateVenue controller, which creates a `SavedVenueCommand`.
- To make our locks persist across different requests, we store them in our PostgreSQL database with the following schema:
    - `Lockableid`: Primary key and the ID of the item to be locked. For our use cases:
        - In a read-all lock, this field is not *really* in use; we store a meaningless UUID in it so this row would exist.
        - In a write-one lock, this field is the selected VenueID.
    - `ownerid`: ID of the lock's owner. For our app we store the owner's SessionID here.
    - `version`: Not to be confused with "version" from an optimistic lock. We store the lock type (**read-one** (not in use) / **write-one** / **read-all**) in this field as an integer.
- We create the singleton interface `LockManager`, which processes requests to acquire and release locks. Then, we implement this interface with a customised `ReadWriteLockManager`.
    - `ReadWriteLockManager` can manage locks from the database by:
        - Storing read-one/write-one/read-all locks for a session (and lockable).
        - Releasing the above.
        - Releasing all locks for a session.
        - Checking if a lockable has a read/write/either lock
        - Checking if the table has any write lock
    - This thus allows us to perform read-write lock management.
    - The acquisition methods throw `ConcurrencyException`, which allows the app to detect when the request is rejected due to being unable to acquire a lock.
    - The `ReadWriteLockManager` is also kept thread-safe by making its instance creation process `synchronized`.
- Our `Command`s facilitate business transactions by doing the following:
    - Initialise a UnitOfWork (if applicable for the use case).
    - Lock-acquiring requests:
        1. Runs `startNewBusinessTransaction()` to begin the session.
        2. Calls `ReadWriteLockManager` to acquire a lock.
            a. If this succeeds, continue onwards
            b. If this fails, throw a `LockFailureException`.
        3. Run logic required to retrieve Venue information that the user needs
        4. Store this information in the session, where the controller will be able to retrieve the information from so it can output it to the frontend
    - Lock-releasing requests:
        1. Runs `continueBusinessTransaction()` to continue the session.
        2. Runs any business logic applicable. (e.g. creating Event, updating Venue).
        3. Calls `ReadWriteLockManager` to release a lock.
    - Committing the UnitOfWork, or rollback its changes if an Exception is thrown.
- Note that the following classes are **not** used in the running of our app: ViewVenueCommand, ReleaseVenueCommand, LockRemover, ImplicitLock

Sequence Diagram

Fig. 4~8 depict the controllers relevant to this pessimistic lock. Fig 9~12 depict the ReadWriteLockManager handling requests to acquire/release the relevant locks. Fig 13~14 depict the session setup process done by the BusinessTransactionCommand, which allows each lock's owner (session) to be identified.

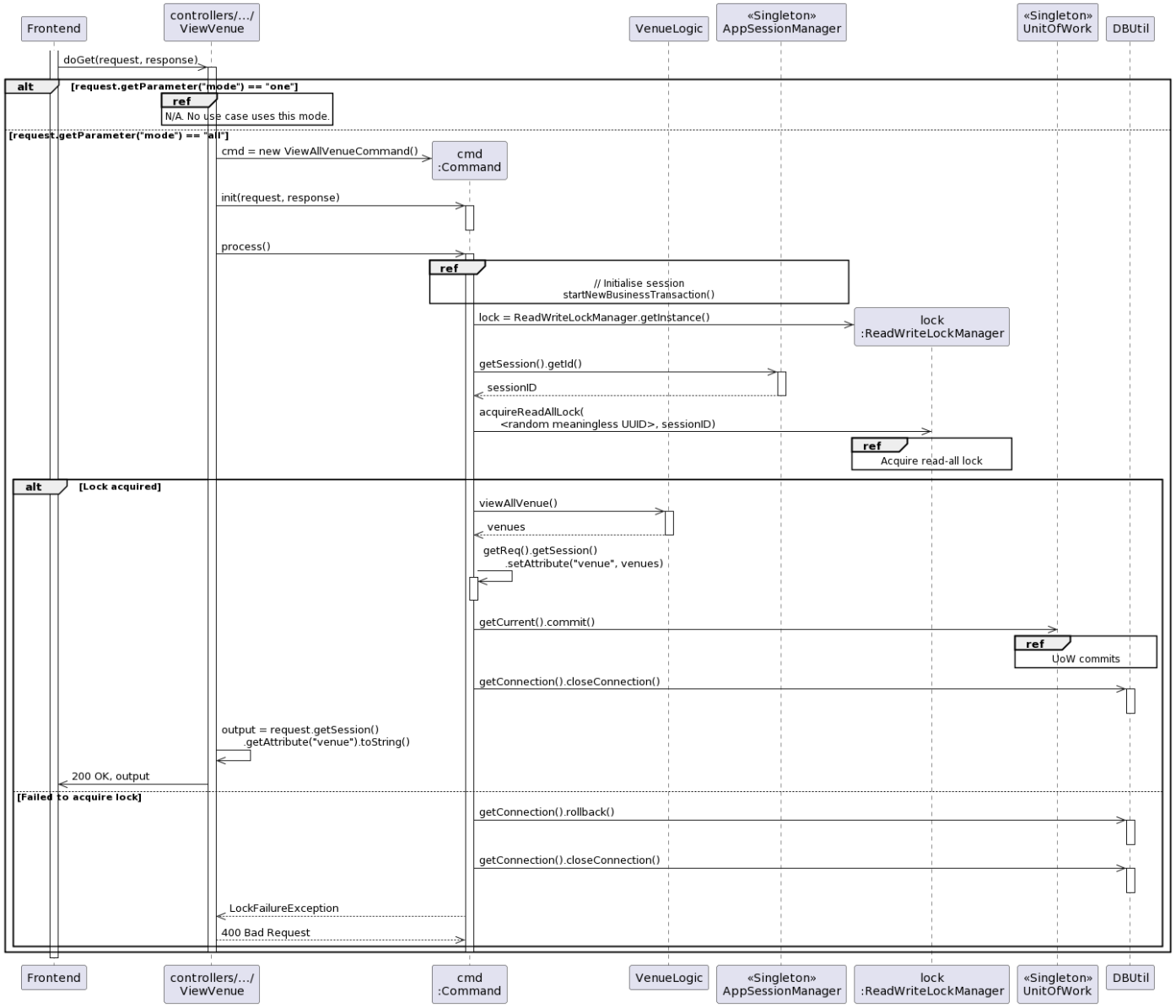


Fig 4: The /view-venue GET path acquiring a read-all Venue lock for a browser session

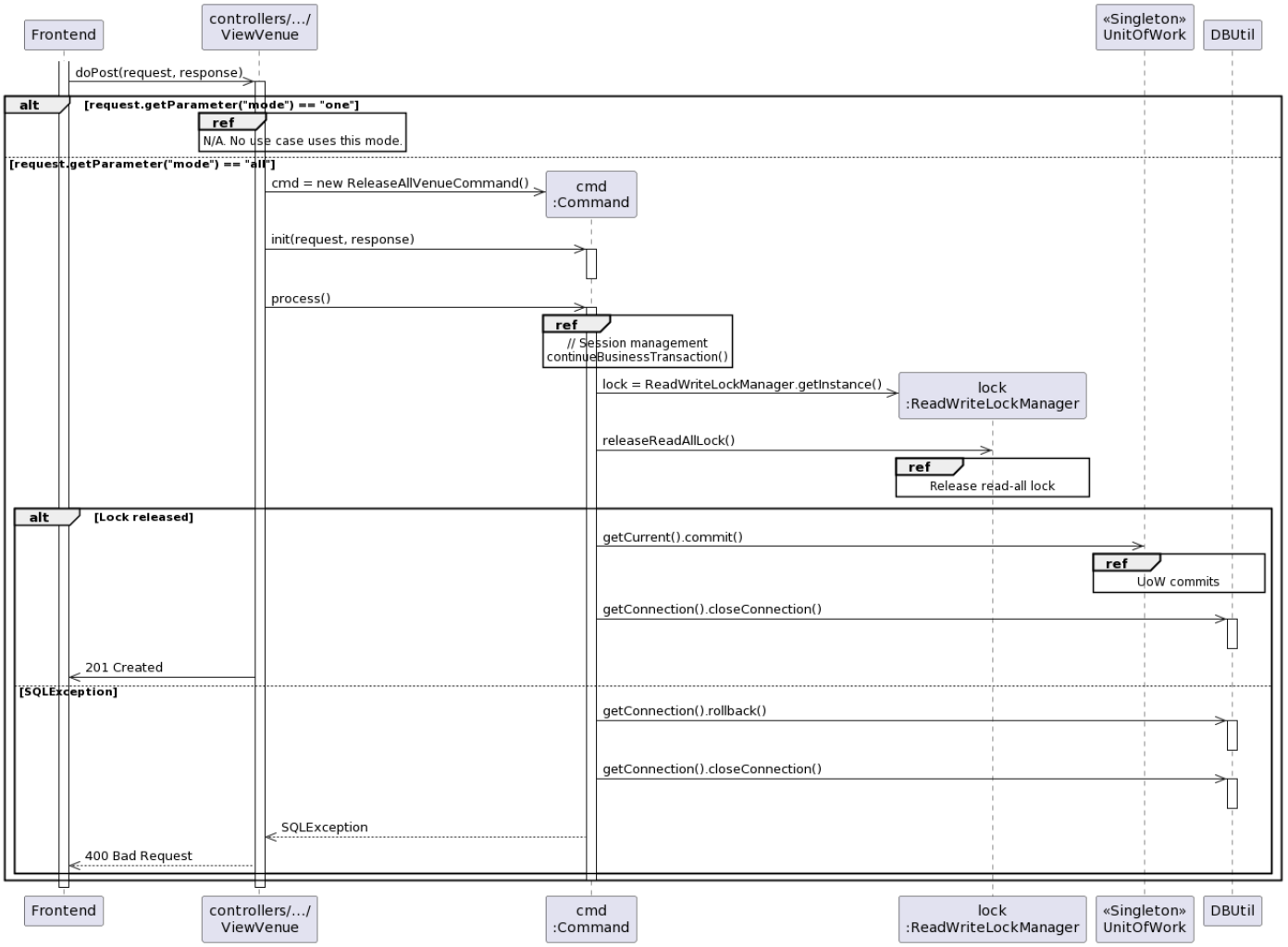


Fig 5: The /view-venue POST path releasing a read-all Venue lock for a browser session

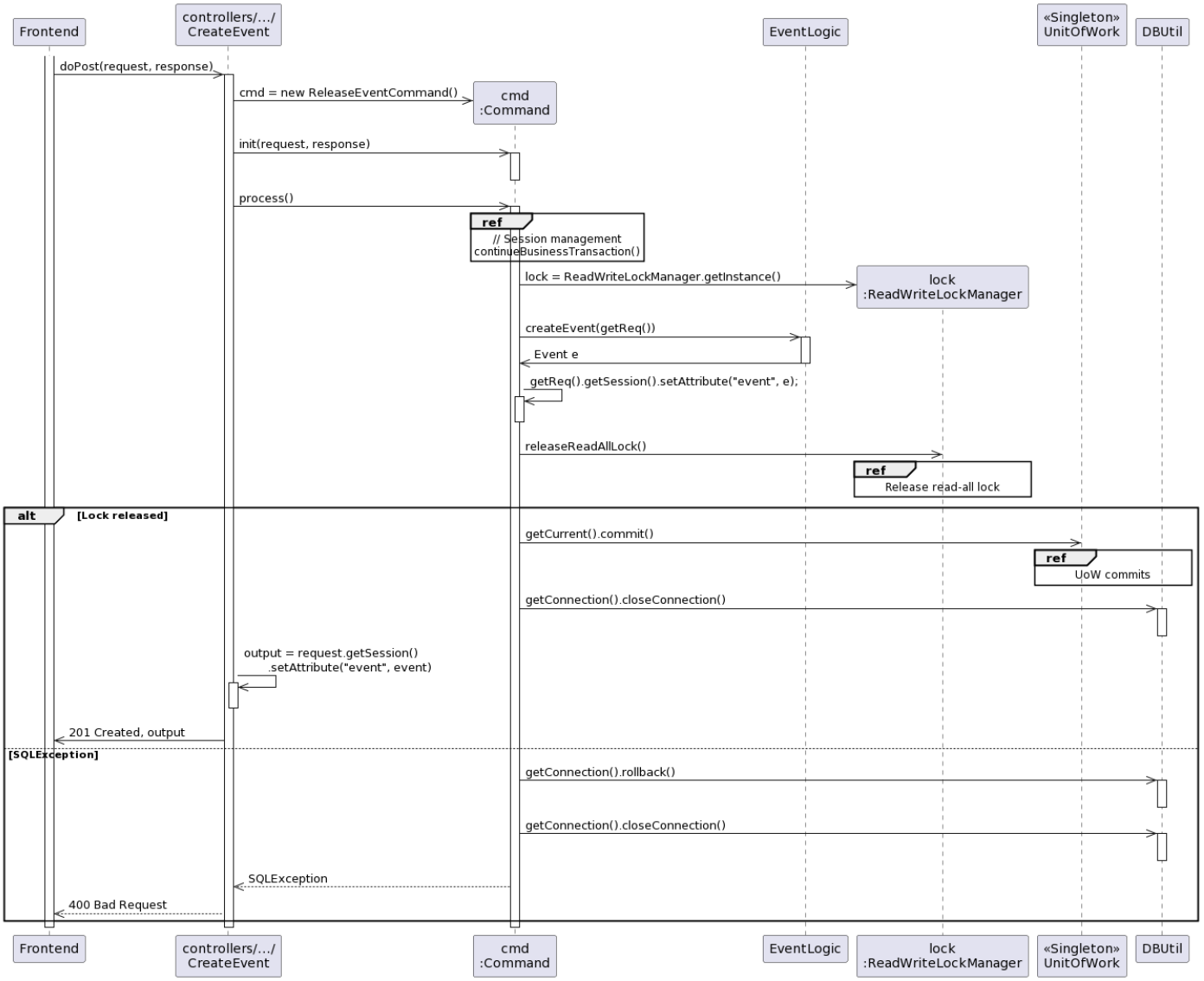


Fig 6: The /create-event POST path releasing a read-all Venue lock for a browser session

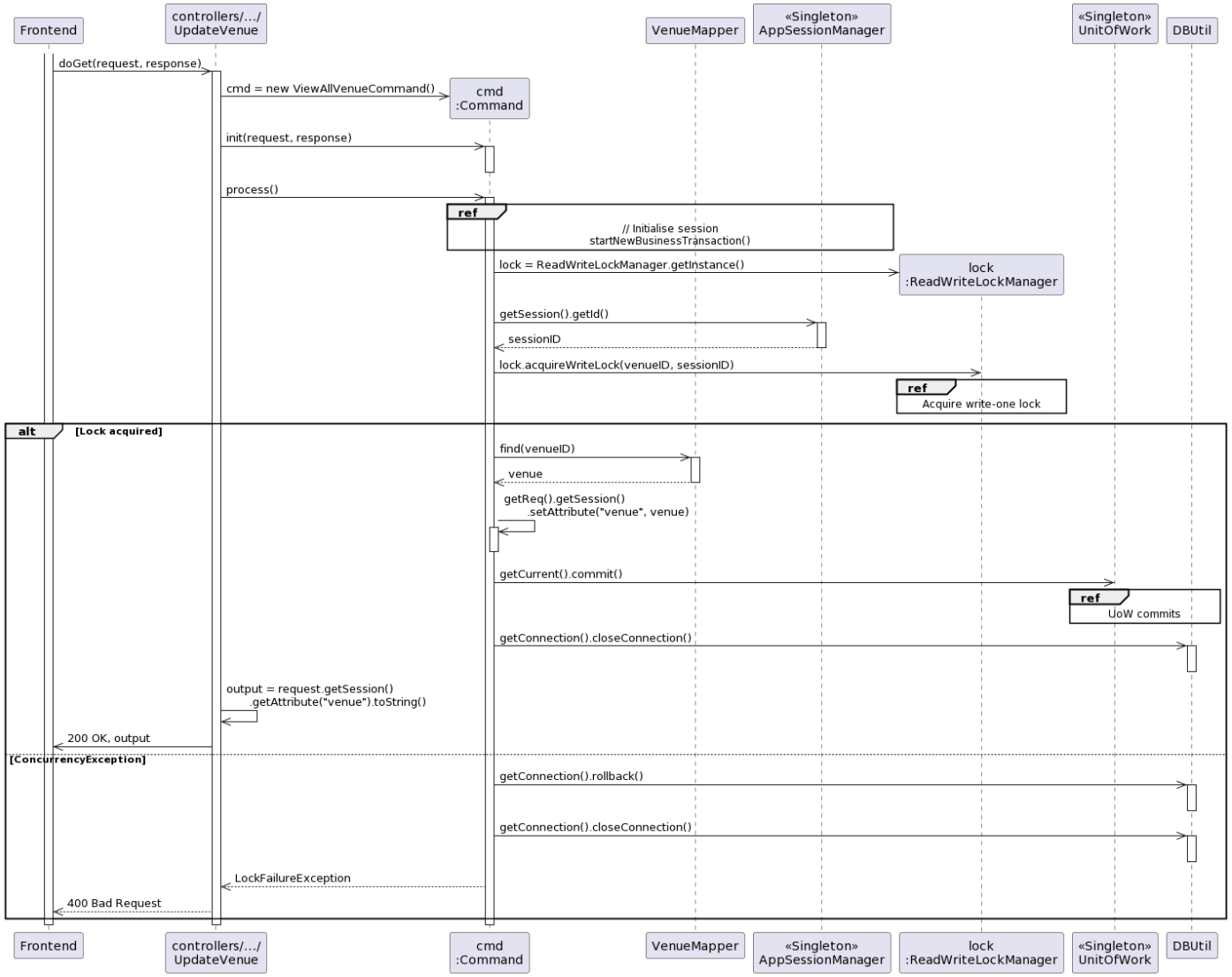


Fig 7: The /update-venue GET path acquiring a write-one Venue lock for a browser session

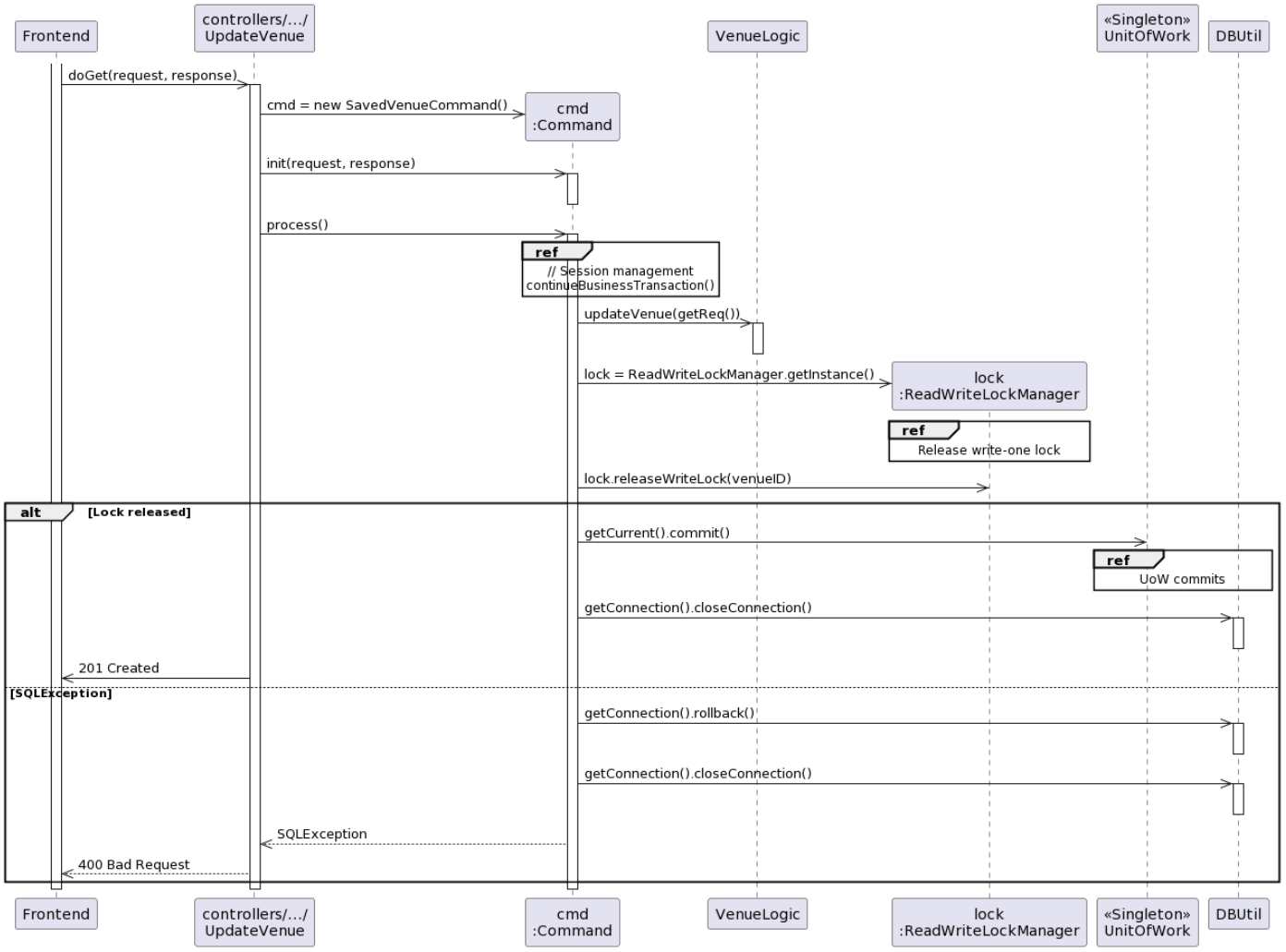


Fig 8: The /update-venue POST path releasing a write-one Venue lock for a browser session

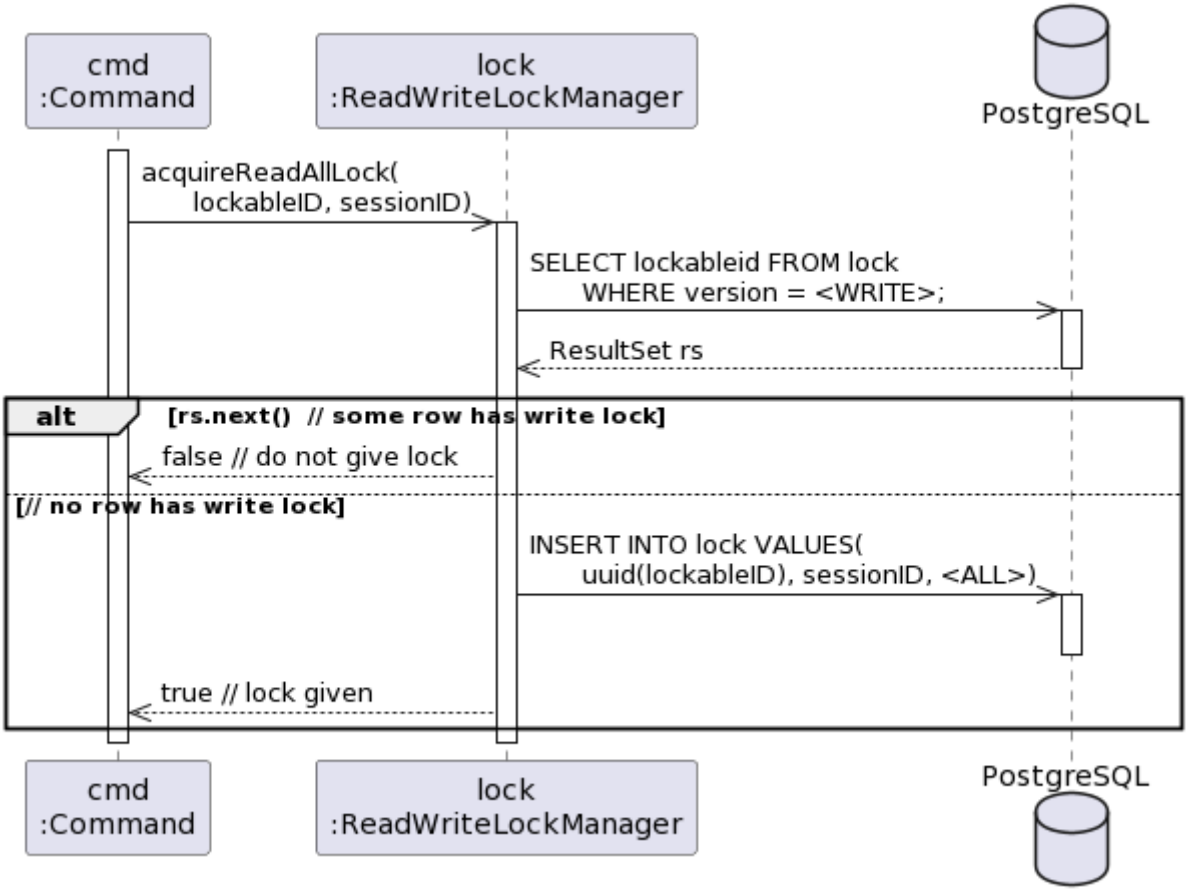


Fig 9: `ReadWriteLockManager` letting a session acquire a read-all Venue lock. Note that <WRITE> and <ALL> refers to the Integer constants stored in the class to identify lock type

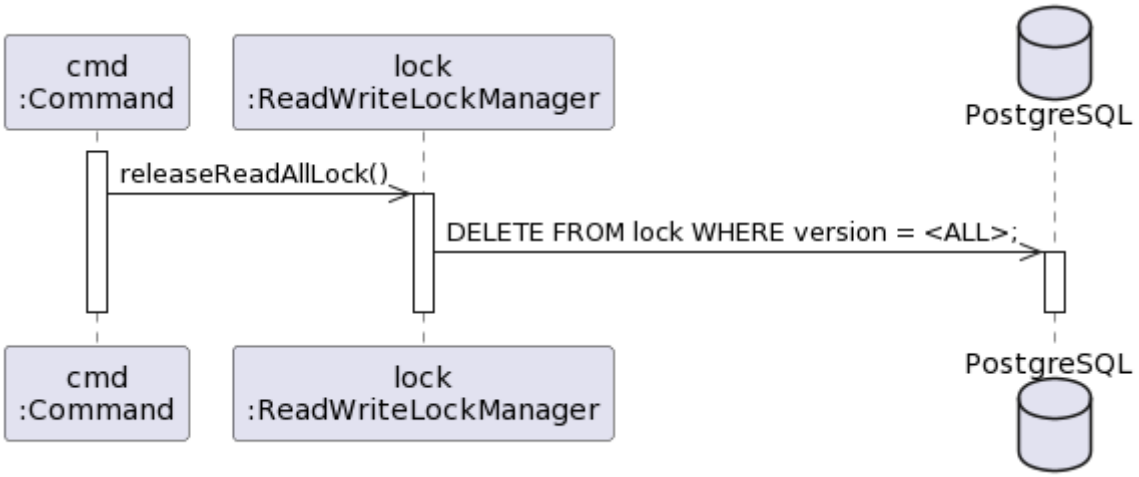


Fig 10: `ReadWriteLockManager` letting a session release a read-all Venue lock

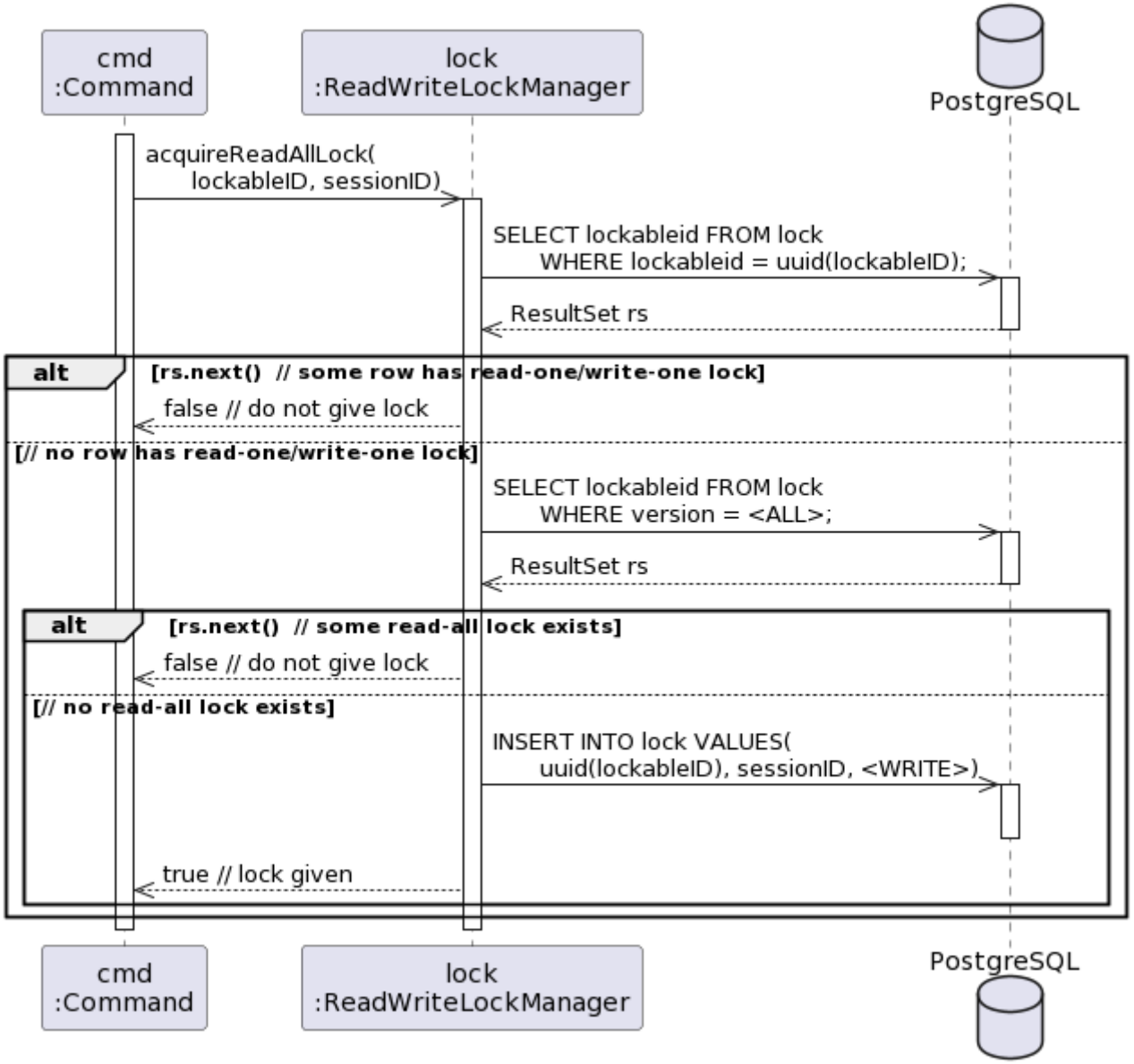


Fig 11: `ReadWriteLockManager` letting a session acquire a write-one Venue lock

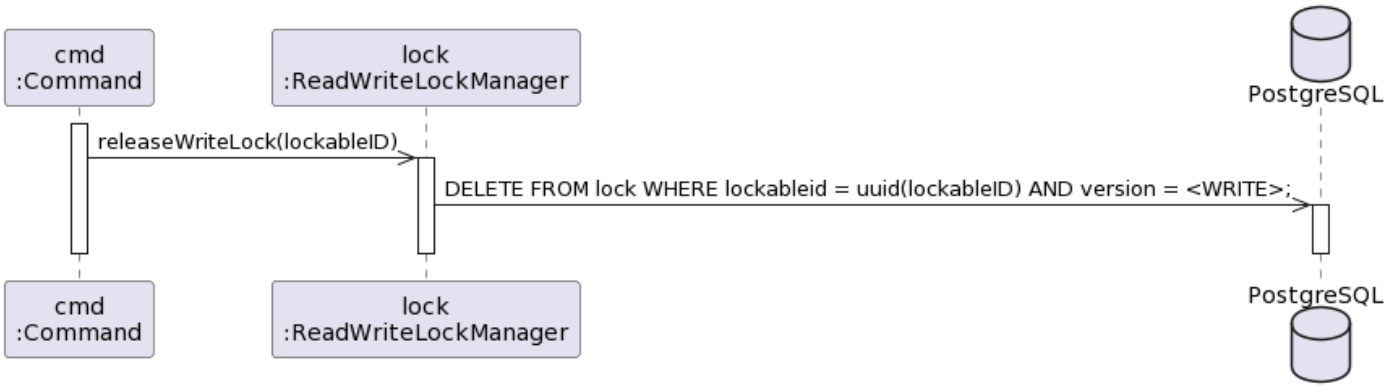


Fig 12: `ReadWriteLockManager` letting a session release a write-one Venue lock

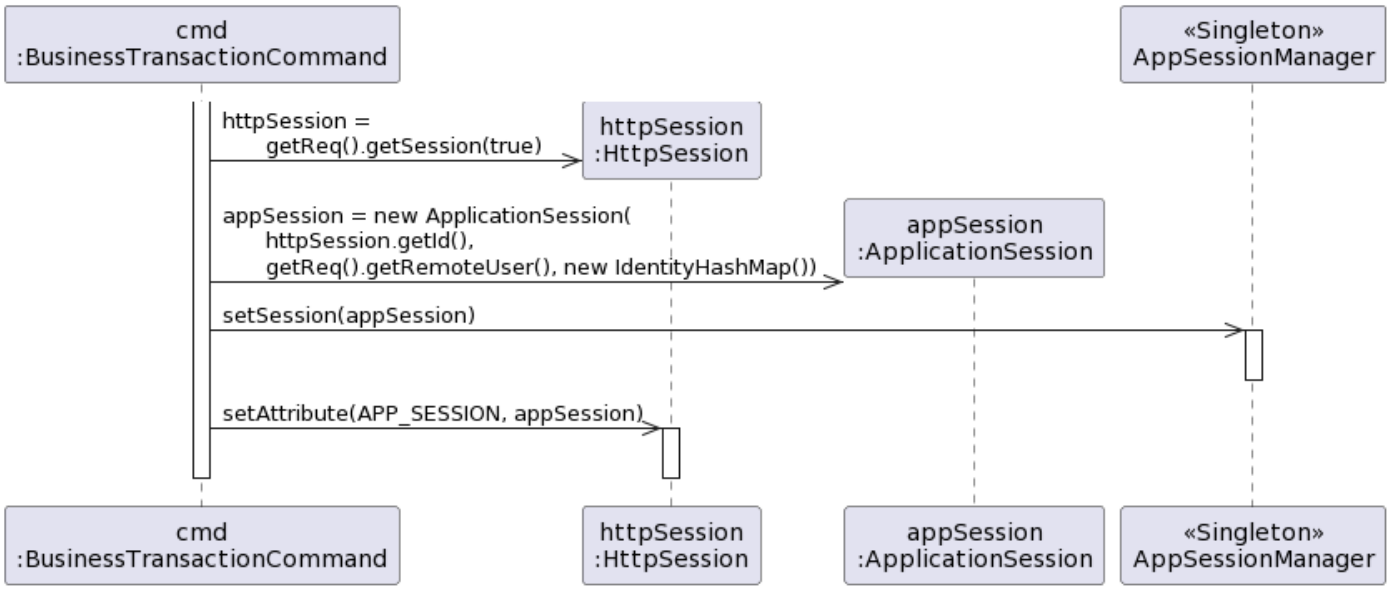


Fig 13: `BusinessTransactionCommand` initialising a session by storing an `ApplicationSession` object within the browser session and within `AppSessionManager`

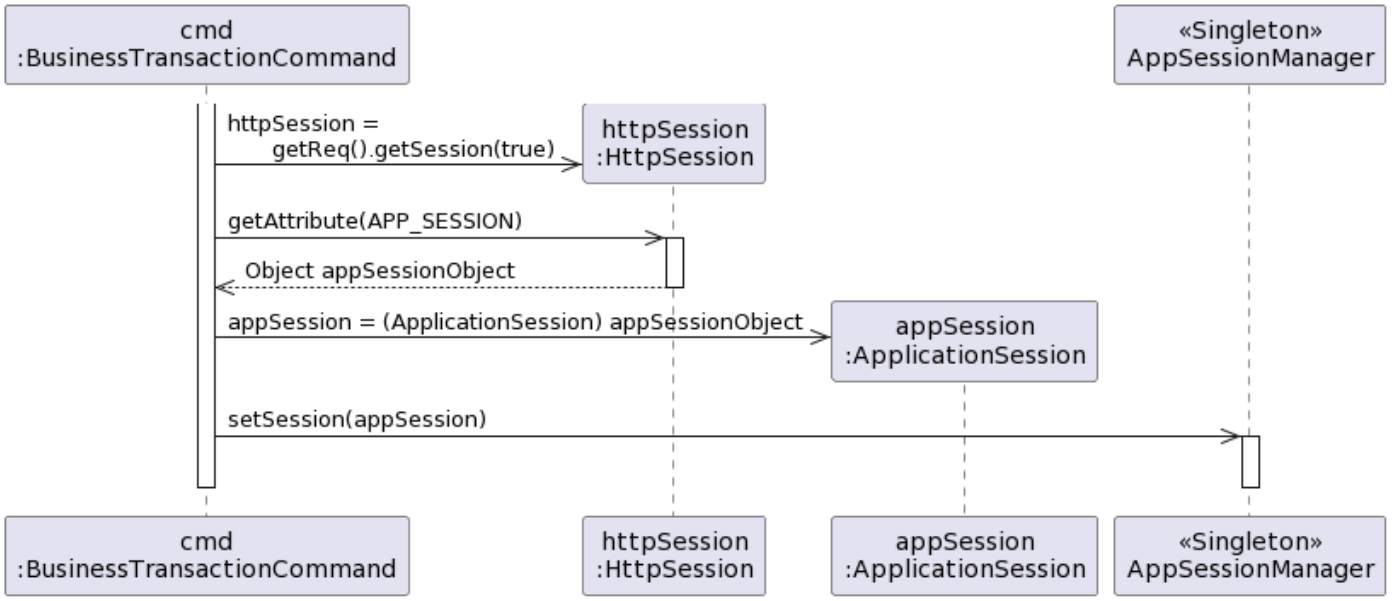


Fig 14: `BusinessTransactionCommand` retrieving an ongoing `ApplicationSession` from the browser and storing it within `AppSessionManager`

Testing strategy and outcome

TC002	Editing venue
TestType	Functional
Execution Type	Manual
TestCase Description	Editing the venue -section data.
Preconditions	Venue and section is already created in the system.

Test sample data	Venue - Richmond Section1- Standing Section2 - VIP User1 - John Eventplanner - Jackson Administrator - Mary
Steps to reproduce	1. Eventplanner namely Jackson creates a event on venue Richmond. 2. Customer namely John books an event which is at venue Richmond. 3. Administrator namely Mary edits the venue Richmond, by removing the VIP section.
Test Output.	Administrator Mary should be able to edit the venue. And notification has to be sent to Eventplanner Jackson and Customer John.

• Editing Event & creating Booking

Issue description

This is an inconsistent read issue, in that customers are creating Bookings based on an outdated version of the Event (and the prices of its associated EventSections, which are updated together with the Event in this use case).

This leads the following issues:

- The customer may book for an event that they could actually not attend (e.g. due to the updated time/Venue location being unaccessible to them)
- The customer may book for an EventSection that is pricier than they expected.

Pattern(s) and/or concurrency mechanisms used

We use an **optimistic offline lock** to counter this issue for the following reasons:

- Event edits are very rare, especially since planners will avoid making changes to a published Event in the first place (in hopes of staying reputable amongst real-life customers)
- We already have implemented such a lock on EventSections for the Simultaneous Booking scenario (described in a previous section), so we can reuse this lock for this.

Implementation details

- We reuse the lock in the Simultaneous Booking scenario.
- Our “Edit Event” use case is implemented such that all EventSections are always updated after an edit. Hence, this use case also updates and checks against their versions.

- When the versions sent from the frontend turns out to be outdated, EventSectionMapper throws an exception, which prompts the frontend to show an error.
- Although we moved the responsibility of holding onto the database connection out of `UnitOfWork`, and instead into the singleton DBUtil, our `UnitOfWork` is still able to manage database connections via DBUtil.
    - In this case, `UnitOfWork` automatically handles EventSectionMapper's `OptimisticLockingException` as part of the standard Event update process.

Sequence Diagram

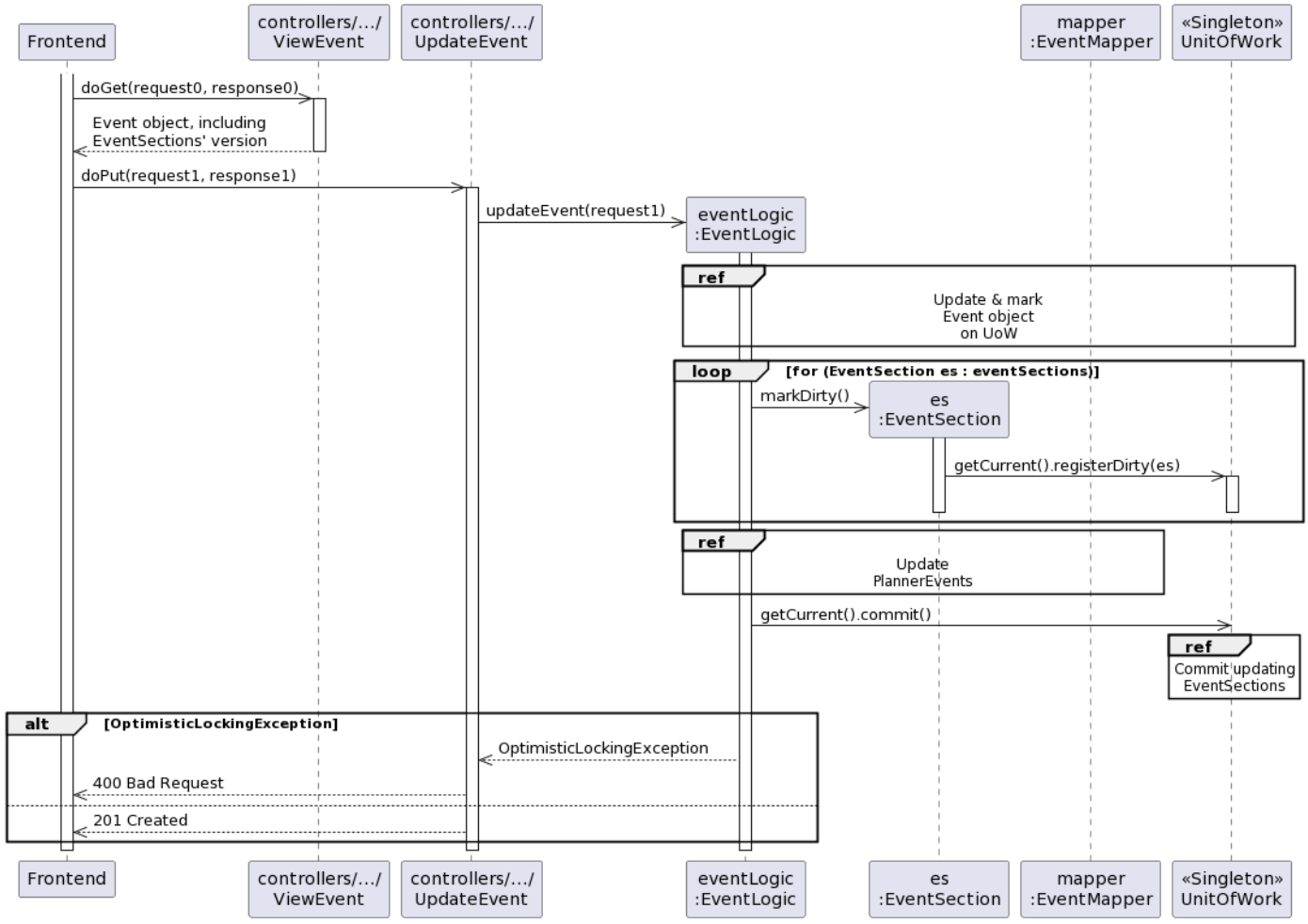


Fig 15: Sequence diagram of an Event update, showing the scenario where the "Commit updating EventSections" subprocess throws an `OptimisticLockingException`.

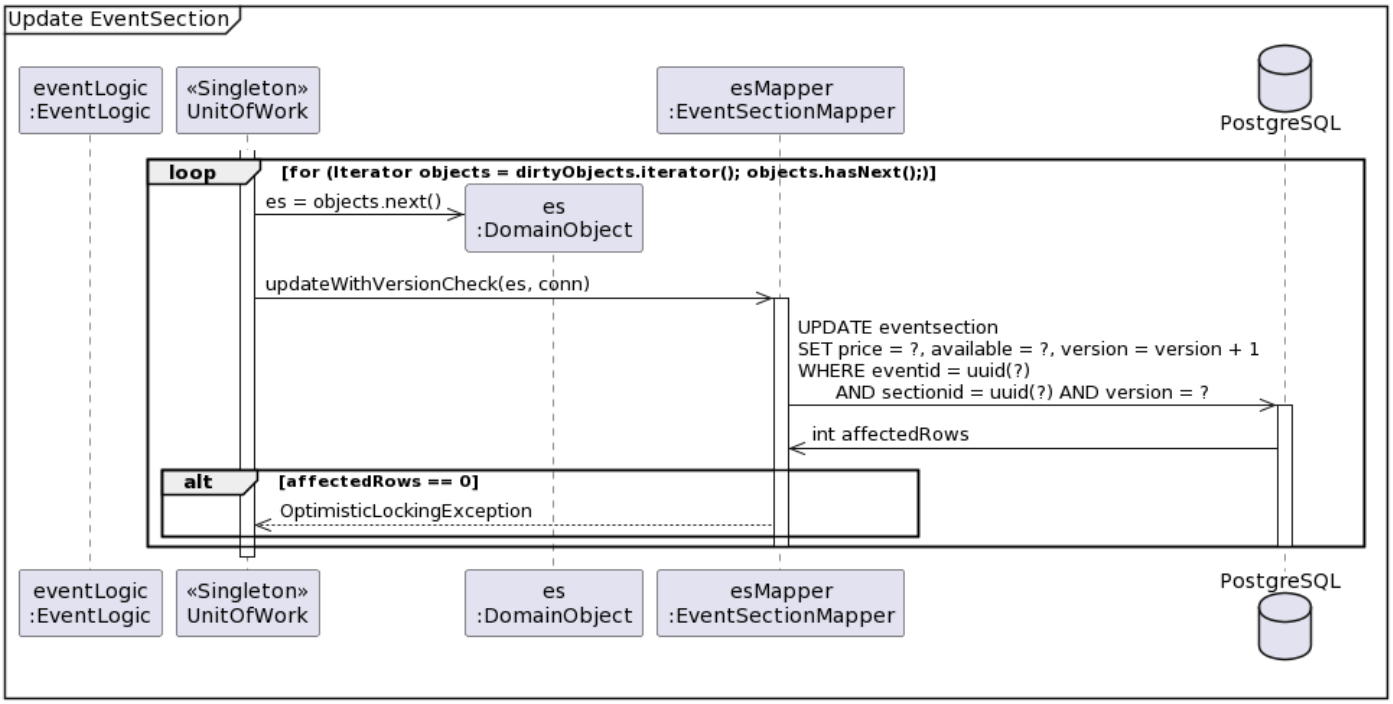


Fig 16: Sequence diagram of the EventSection update during a full Event update, showing how the optimistic lock works

Testing strategy and outcome

TC003	Editing Event & creating Booking
TestType	Functional
Execution Type	Manual
TestCase Description	Event is modified after some customers have already booked for the event.
Preconditions	Customers and events are already created in the system.
Test sample data	Venue - Richmond Eventplanner1 - Jackson Event - Name is “Star Nite” on Venue Richmond on 10 Dec 2023 for time 6pm to 10pm.
Steps to reproduce	1. Jackson created the event “Star Nite” on Venue “Richmond” on 10 Dec 2023 for time 6pm to 10pm. 2. Customer Alice books 10 tickets for event “Star Nite” on Venue “Richmond” 3. Jackson modified the event “Star Nite” on Venue “Richmond” from 10 Dec 2023 to 10 Jan 2024 from 6pm to 10pm.

Test Output.	TBD

• Simultaneously create/edit Events to be at the same Venue and time

Issue description

Although we do not store the occupied times within Venue, this issue is one of an inconsistent read of some sorts, in that the second request would be creating/editing the Event based on a version of the database that does not have another Event running at the selected Venue and time.

This issue can lead to one of two problems:

1. Two planners simultaneously register the same “event” (in the business domain) twice, at the same Venue and time.
2. Two planners simultaneously register two different “events” at the same Venue and time.

Variant 1 is less of an issue; two duplicate “events” would confuse customers, but ultimately they both facilitate the real-life business function of tracking customer entry into the event.

However, Variant 2 leads to the implication that two different events are occurring at the same Venue and time, which is infeasible in the business domain. (Since this cannot occur in real life, it must be the case that either the real-life “venue” has double-booked, or that one planner has entered the Event’s time wrongly.)

Pattern(s) and/or concurrency mechanisms used

We implement the **Optimistic Offline Lock** to counter this issue for the following reasons:

- The frequencies of both variants above are very rare.
    1. If multiple planners are organising the same real-life “event”, they likely would have been able to coordinate and ensure only one user is creating the event.
    2. It is unlikely that either a real life “venue” would be double-booked, or that a planner would mis-enter their data.
- Regardless of time, the chance of two Events simultaneously being registered at a particular Venue is relatively rare; each venue is only able to host one event at a time, and real-life venues also have downtimes for maintenance etc.. Thus, not too many Events will be running at a Venue in the first place, so lost work is acceptable.

Implementation details

- We add a “version” field in the **Venue** table. This field is updated whenever an Event is being created or updated. (This is **not** updated when the Venue itself is created/updated, since this lock is not relevant to those use cases.)
- When a planner browses through Venues during Event creation, they will also receive the version of the Venue. When making an Event creation request, the frontend sends the version of the Venue together with the to-be-created Event.

- Within the create/update methods of EventLogic, we already check for the Venue's occupied times before registering items onto the UnitOfWork. We now add the step of checking that the Venue's version has not been changed since the last time it is viewed by the planner, i.e. that the version sent from the frontend equals the version in the database.
    - We create the `checkVenueAvail` and the `checkUpdateVenueAvail` methods in EventLogic. The former is for Event creation only, while the latter is for Event update; they only differ in their time-checking logic in that they take their own existence into account when calculating overlaps in existing time periods.
- VenueMapper is tasked with updating the Venue. An additional SQL statement is added for version updates, in which its WHERE clause is configured to only update Venue rows with both the right id and the version sent from the frontend.
- VenueMapper counts the number of rows affected by the Venue update. If this is 1, this indicates that the frontend's version is up to date. If this is 0, this indicates that the frontend's information is outdated.
- If the frontend's version is up to date, the request succeeds.
- If the frontend's version is outdated, VenueMapper throws an exception within the system. The frontend will thus be notified of this and rollback all changes.
    - The frontend will receive an error message and refresh the page, so that a more recent version of the data is displayed.

Sequence Diagram

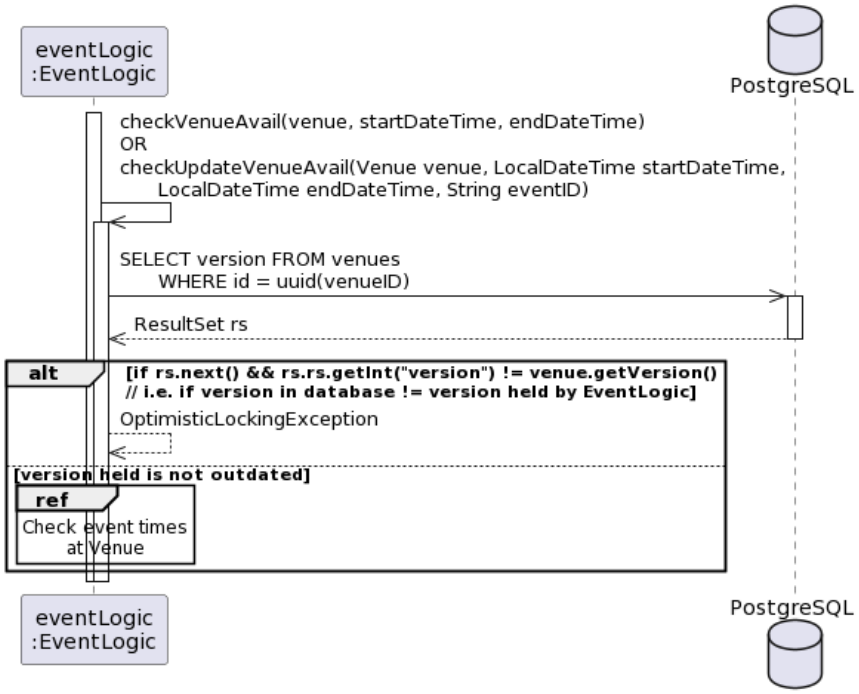


Fig 17: The Venue version check part of the optimistic lock: `checkVenueAvail`/`checkUpdateVenueAvail`

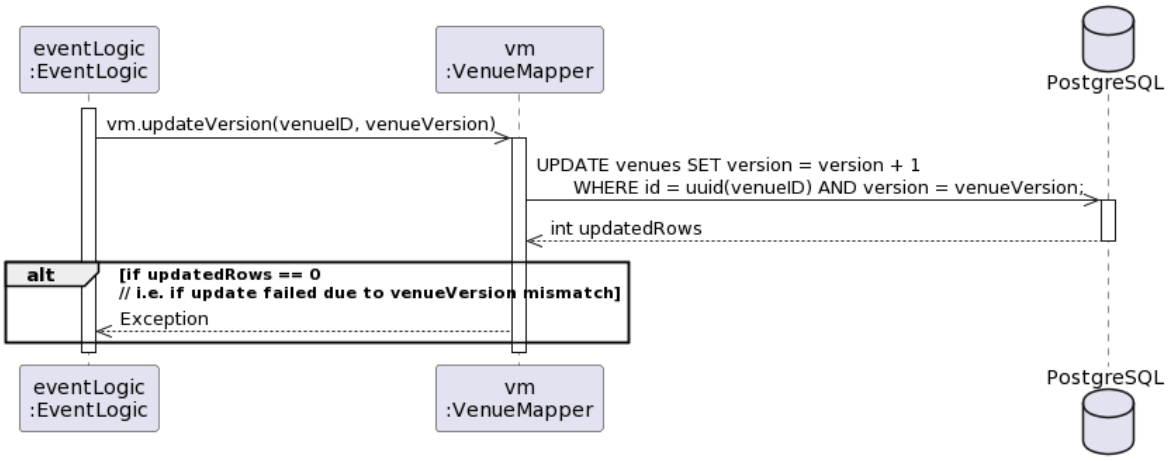


Fig 18: The optimistic lock at work when VenueMapper updates Venue version

Testing strategy and outcome

TC004	Simultaneously create/edit Events to be at the same Venue and time
TestType	Functional
Execution Type	Manual
TestCase Description	Creating events to the same venue and time.
Preconditions	Venue and section is already created in the system.
Test sample data	Venue - Richmond Eventplanner1 - Jackson Eventplanner2 - Paul
Steps to reproduce	1. Jackson created an event namely “Starnite” on Venue Richmond on 10 Dec 2023 for time 6pm to 9pm. 2. Paul created an event namely “RockPeople” on Venue Richmond on 10 Dec 2023 for 6pm to 9pm.
Test Output.	Only one of them is able to create their event. The other one will receive an error alert.

• Concurrency non-issues

A number of other concurrency situations may or may not occur in our system. We do not consider them to be issues:

- Simultaneous reads are not an issue to our system; in fact, due to the nature of our business domain, it is essential that simultaneous reads are allowed so that e.g. many customers are able to read information about the same events.

- Lost updates due to simultaneous editing use cases (Event+EventSections, Venue+Sections) are not an issue:
    - Event planners will not be basing their changes on any of the Event's previous information (e.g. they will not be incrementing counts or anything of the sort). Thus, any lost updates can easily be recovered through outside communication between planners (which they will be able to make since they are already working on the same event).
        - Moreover, with the newly-implemented optimistic lock on EventSection, this will not occur. Due to our current implementation, an Event and its EventSections are treated as an aggregate, and an Event cannot be updated if the EventSections fail to update due to simultaneous edits.
    - Similarly, Venue and Section changes are not based on past data. The admin (and anyone who the admin gives account access to) will already be able to coordinate and recover lost updates.
- Attempting to create/update data based on deleted information (e.g. creating an Event at a deleted Venue) will not be an issue. Since we implemented foreign key constraints within the PostgreSQL database, a new creation will not be able to reference a newly-nonexistent row, and existing objects referencing it will automatically cascade. Such a creation/update will result in lost work, but this is inevitable when the referenced object is being deleted.
- Dirty reads will not occur. Our `UnitOfWork` is implemented such that all changes on a Connection to the database are committed all at once at the end of a transaction, (or rolled back entirely, when an Exception is thrown). Thus, changes to the database will be atomic (since our database's isolation level is set to "READ COMMITTED" by default) and information will not be stored in a dirty state.
- Simultaneous user creation on the same email will not be an issue, since our database is set up so that the email field is the primary key, and hence will be unique. The second user will simply receive a duplicate email error.
- Simultaneous creation/edit of events at the same Venue can be an issue, as their times may overlap; due to having an extra step of having to check for existing Event times at the Venue, the two threads can interleave and end up both determining that one particular time can be booked at. However, the business domain means that this can only occur in one of three ways, all of which have such a small likelihood of happening that it is not an issue:
    1. Multiple planners are organising the same real-life "event", and thus created a duplicate Event: They likely would have been able to coordinate and ensure only one Event is created in the first place.
    2. The real-life "venue" is double-booked, leading to multiple distinct Events: This would be an oversight on the venue staff's scheduling, which is extremely unlikely given the importance of the task and the scheduling tools that the real-life staff would have.
    3. One event planner mis-entered their Event data, leading to multiple distinct Events: Similar to the above, this is unlikely to happen due to the importance of this task.