Music Event System

SWEN90007 Software Design and Architecture

Project: Part 4

Software Architecture Report

TEAM NAME - **MusicBandTeam**

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

Table of Contents

1. Design **patterns** used / haven't used

Foreign Key Mapping

The Foreign Key Mapping pattern is essential in object-relational mapping and plays a crucial role in the performance optimization of software applications, by keeping the object model and the database schema in sync. It means that as our software grows, both object model and database can scale harmoniously, ensuring consistent performance improvements with scalability.

We used Foreign Key Mapping both to map our one-to-many relationships and our many-to-many relationships between our objects in the domain. When an object is loaded from the database, its associated objects can be loaded through foreign keys. In our project, for instance, the `Event` object is loaded with its associated `Venue` object using the `Venue`'s ID. Instead of having separate queries managed by the domain logic, we automate this process via the `EventMapper` in a streamlined manner.

Although retrieving by foreign keys negatively affects READ performance since we will have to retrieve from multiple tables, this avoids creating duplicate data in the database. This thus improves performance of CREATE, UPDATE and DELETE by avoiding the need to manage multiple copies of the same data across multiple rows (e.g the same Venue address and name, throughout every Event located at this particular Venue.)

As a result, this also helps us maintain referential integrity, ensuring that the relationships between data are consistent and valid. Although this isn't a direct performance benefit, it does ensure the data's reliability, meaning fewer errors and, consequently, fewer performance-hogging error-handling routines.

Moreover, instead of pulling entire objects with all their data from the database, Foreign Key Mapping allows us to pull only relevant object identifiers. Combining with lazy load, it can significantly reduce memory consumption. For example, our EventSections table only contains the relevant Sections' IDs rather than entire Sections. This way, it enables the `EventSection` domain object to store the `Section` as a `ValueHolder` and only load it when necessary.

The Foreign Key Mapping also helped us handle changes in association more cleanly. Our Events and our Planners (in the User table) have a many-to-many relationship, which we used an association table to model. When changes occur in their relationship (e.g. the planners responsible for an event changes), only the association table needs an update rather than the main tables, leading to faster and more efficient updates.

Optimistic lock

In our system, we have implemented the Optimistic Offline Lock pattern to manage concurrency during booking creation and other operations that update the EventSection entity. This pattern was chosen because:

- Booking creation is a straightforward process, primarily involving the creation of a new booking and checking available tickets for the corresponding EventSection.
- The simplicity of the operation means that any lost work due to conflicts is minimal and easily recoverable.

Positive Impacts on Performance:

- Minimised Lock Overhead:

  Unlike pessimistic locking, optimistic locking doesn't require locking records during the read phase. This means less overhead in managing locks and fewer chances of encountering lock contention, which can significantly improve performance, especially in read-heavy systems like ours.

- Increased Throughput:

  By allowing multiple transactions to proceed in parallel without acquiring locks, optimistic locking enables higher throughput. Users can create bookings and read EventSection data concurrently without being blocked, leading to a smoother and faster user experience.

- Reduced Rollback Scenarios:

  Conflicts in our system are expected to be infrequent given the simplicity of the booking creation process. This means that the likelihood of rollbacks, which can be costly in terms of performance, is relatively low. Optimistic locking, thus, strikes an efficient balance by only rolling back when necessary, preserving system resources.

Challenges and Mitigations:

- Conflict Resolution:

  When conflicts do occur, they can lead to some performance overhead due to the need to rollback and retry transactions. However, given the simplicity of the booking process in our system, the recovery from these conflicts is quick and has a minimal impact on overall performance.

- Stale Data Handling:

  There's a chance that users might be working with stale data, which could lead to an increase in conflicts. Our system mitigates this by promptly informing the frontend to refresh the data upon detecting a version mismatch, thus minimising the window for potential conflicts.

System-Specific Performance Considerations:

- In our system, the booking creation is a critical yet relatively simple operation. By employing optimistic locking, we've optimised for this use case, ensuring that performance isn't hampered by unnecessary locking mechanisms.
- Our careful integration of the version check into the frontend and backend minimises the performance impact of data conflicts. By swiftly detecting and responding to version mismatches, we maintain a high level of system responsiveness.
- The thread-safe design of our DBUtil class ensures that database connections are efficiently managed, further contributing to the system's performance by reducing potential bottlenecks associated with connection handling.

In summary, the adoption of optimistic locking in our system enhances performance by reducing lock contention, increasing throughput, and ensuring efficient conflict resolution. While there are challenges associated with managing stale data and resolving conflicts, our system's design effectively mitigates these issues, leading to a responsive and reliable user experience.

Pessimistic lock

In our application, we implemented a pessimistic lock on the Admin Edit Venue and the Planner Create Event use cases through the following components:

- Session Setup:
    - Browser sessions are identified via `request.getSession()` to ensure user-specific lock matching.
    - An `ApplicationSession` stores each session's ID, user identifiers, and a cache for transactional data, enabling efficient lock management and data retrieval.
    - `AppSessionManager` and `Command` interfaces facilitate session management and lock acquisition/release, respectively.
- Lock Management:
    - Lock management is integrated with user flow to avoid deadlocks and ensure atomicity in resource acquisition.
    - A PostgreSQL database schema is designed to persist locks across requests, capturing the lockable item, owner, and lock type.
    - `ReadWriteLockManager`, a singleton, manages database locks and ensures thread safety.
    - Once a read-all lock is acquired on the venues, no other transaction can acquire a write lock on any venues until the lock is released.
    - Once a write-one lock is acquired on a venue, no other transaction can acquire read-all locks or write-one locks on the same venue until the lock is released.

Concurrency control is crucial for our system; venue edits, albeit infrequent, have a cascading effect on multiple events. Our lock acts as a guardian against concurrent modifications during transactions. For instance, when an event planner is organising an event, the venue data remains locked until the transaction is completed (and vice versa).

This eliminates data anomalies such as dirty reads, non-repeatable reads and phantom reads during transactions involving venue edits or event creation, and thus fortifies our system's operational integrity during these critical operations.

Concurrency control does require the additional steps of session management as well as calling the `ReadWriteLockManager` to check for locks from the database or to delete existing locks from the database, which is additional overhead and increases our response time. However, we designed our lock mechanism with ***performance*** in mind:

- We use a pessimistic lock for these use cases, which reduces response time through providing the following:
    - Minimised rollback scenarios: By adopting pessimistic locking, our system preemptively avoids situations that would necessitate rolling back transactions due to late conflict detection, a scenario often encountered with optimistic locking especially in systems with high write volumes. Thus, when concurrent sessions are making transactions, requests are rejected earlier on in the process rather than at the end.

    - Optimised system resources: By decreasing the frequency of conflict resolutions, our system conserves computational resources, leading to better overall performance. This optimization is particularly beneficial during peak usage times when multiple users are interacting with the system simultaneously.

- Streamlined Workflow: Users, particularly event planners and administrators, experience a streamlined workflow when booking venues or updating event details. The system ensures that once they begin a transaction, they have exclusive access to the necessary records, thereby providing an uninterrupted and consistent interaction with the system. The predictability of data behaviour due to pessimistic locking also translates into a more reliable system. Users can trust that the data they are interacting with won't change unexpectedly due to concurrent operations, leading to more confident decision-making.

- When implementing the pessimistic lock, we chose the read-write lock. Given the high volume of read operations from various users, it was imperative to allow concurrent reads to maintain system responsiveness and liveness.

Online SQL lock

The online SQL lock is the locking mechanism that is implemented within the SQL Database system in order to regulate the concurrent access to data.There are various type of SQL lock that is implemented within the SQL database namely Read-Lock, Write-Lock,Row-level locks, Table -level locks.These types of locks are called “online”, because the locks are used while the application is running and database is continuously accessed and updated. (to be continued)

Tradeoffs between table data gateway and whatever we have (mapper -> mixed logic with domain, quicker, hardcode? hard to maintain)

There is always a tradeoff between maintaining the data integrity and consistency of data in the multi user data environment and System performance.Generally implementing the concurrency control will decrease the overall performance of the system , which will have better performance if no locks are implemented . The performance of the system will be reduced because of the following scenarios.

1. When multiple users are competing for the same resource, only one user will allocate the lock, which makes the other users to wait till the lock is released , this reduces the overall throughput of the system.
2. Also sometimes if the lock is not correctly implemented in the system, then it may lead to deadlock, where user1 is waiting for another user to release the lock, whereas the other user is waiting for the user1 to release the lock . This deadlock situation may sometimes make the system unresponsive.
3. Parallelism will not be achieved because of the implementation of the locks.
4. In the future,if the application grows dramatically then performance can be improved by moving from SQL database to NoSQL Database like MongoDB which is designed to have improved scalability and allows sharding that involves distributing data across multiple machines enabling it to handle high volumes of data and high write loads.

Unit of work

The Unit of Work pattern keeps track of all the changes (insert, update, delete) made to the objects during a single business transaction. Once this transaction is complete, UoW determines all the changes that need to be persisted to the database. This ensures that the database is updated in one go, rather than with multiple, smaller calls.

It maintains separate lists (`newObjects`, `dirtyObjects`, and `removedObjects`) to categorise and track objects based on their modifications—whether newly created, updated, or deleted. To correctly classify these objects, UoW provides registration methods. The commit method, central to the pattern, batches and consolidates all changes, ensuring they are persisted to the database in a streamlined fashion. Additionally, the rollback method offers a safeguard, allowing for the reversion of changes to uphold data integrity in the face of inconsistencies or errors.

Performance Improvement

Instead of updating the database each time an object changes, UoW batches these operations. So, if an object is modified multiple times during a transaction, the database is updated only once for this particular object. This reduces the number of database operations, leading to less time cost, decreased response times, and thus an increase in the throughput.

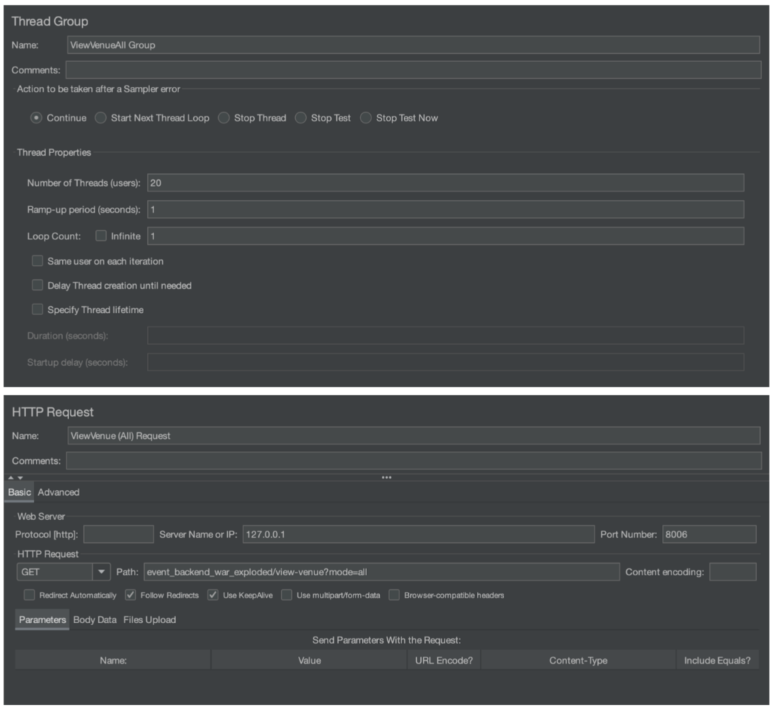
Furthermore, by consolidating changes and applying them in a single transaction, the time the database spends in a transactional state is minimised. This can significantly reduce lock contention and improve the overall transactional throughput of the database.

Moreover, the UoW pattern centralises the tracking of all object changes. This makes it easier to manage and optimise data persistence operations. If a problem arises during a business transaction, the UoW can rollback all the changes, ensuring that the database remains in a consistent state. This avoids partial updates which can potentially lead to data corruption.

2. Room for improvements

Lazy Loading

Currently, our application does not make use of lazy loading. It is only available on our EventSections’ Section type and capacity data, and due to the implementation of our controllers, when Events are requested all this information is also retrieved and outputted to the frontend without utilising lazy load. Similarly, whenever Venues are requested for, for every Venue our system makes another call to the database for its related Section data. This significantly increases our latency.



Thread Group

Name: ViewVenueAll Group

Comments:

Action to be taken after a Sampler error

[✓ Continue] Start Next Thread Loop Stop Thread Stop Test Stop Test Now

Thread Properties

Number of Threads (users): 20

Ramp-up period (seconds): 1

Loop Count: Infinite 1

Same user on each iteration

Delay Thread creation until needed

Specify Thread lifetime

Duration (seconds):

Startup delay (seconds):

HTTP Request

Name: ViewVenue (All) Request

Comments:

Basic Advanced

Web Server

Protocol [http]: Server Name or IP: 127.0.0.1 Port Number: 8006

HTTP Request

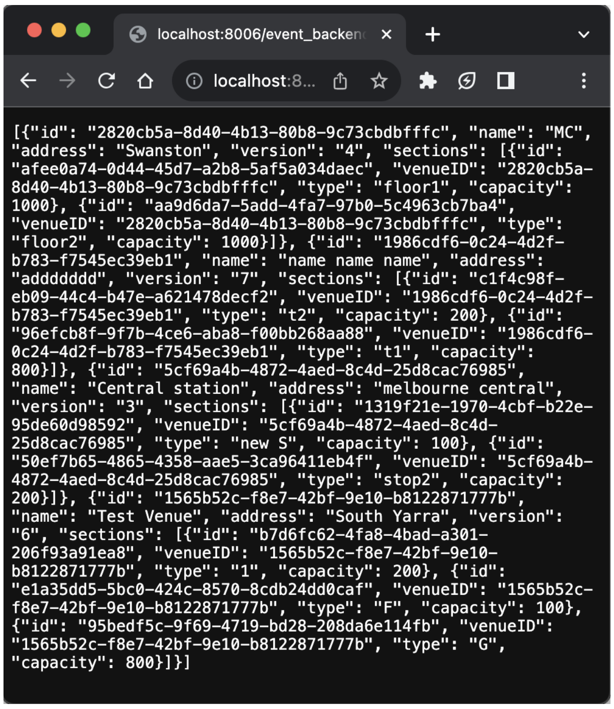
GET Path: event_backend_war_exploded/view-venue?mode=all Content encoding:

Redirect Automatically [✓ Follow Redirects] [✓ Use KeepAlive] Use multipart/form-data Browser-compatible headers

Parameters Body Data Files Upload

Send Parameters With the Request:

Name:	Value	URL Encode?	Content-Type	Include Equals?



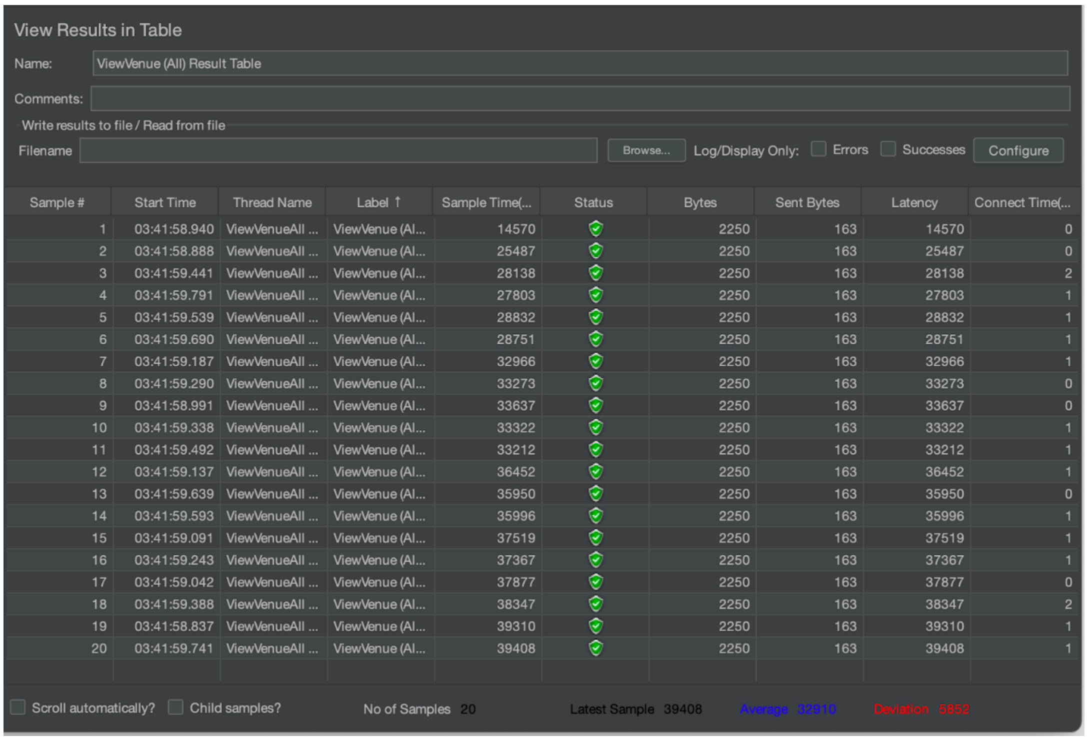
localhost:8006/event_backend

localhost:8...

```
[{"id": "2820cb5a-8d40-4b13-80b8-9c73cbdbfffc", "name": "MC",
"address": "Swanston", "version": "4", "sections": [{"id":
"afee0a74-0d44-45d7-a2b8-5af5a034daec", "venueID": "2820cb5a-
8d40-4b13-80b8-9c73cbdbfffc", "type": "floor1", "capacity":
1000}, {"id": "aa9d6da7-5add-4fa7-97b0-5c4963cb7ba4",
"venueID": "2820cb5a-8d40-4b13-80b8-9c73cbdbfffc", "type":
"floor2", "capacity": 1000}]}, {"id": "1986cdf6-0c24-4d2f-
b783-f7545ec39eb1", "name": "name name name", "address":
"addddddd", "version": "7", "sections": [{"id": "c1f4c98f-
eb09-44c4-b47e-a621478decf2", "venueID": "1986cdf6-0c24-4d2f-
b783-f7545ec39eb1", "type": "t2", "capacity": 200}, {"id":
"96efcb8f-9f7b-4ce6-aba8-f00bb268aa88", "venueID": "1986cdf6-
0c24-4d2f-b783-f7545ec39eb1", "type": "t1", "capacity":
800}]}, {"id": "5cf69a4b-4872-4aed-8c4d-25d8cac76985",
"name": "Central station", "address": "melbourne central",
"version": "3", "sections": [{"id": "1319f21e-1970-4cbf-b22e-
95de60d98592", "venueID": "5cf69a4b-4872-4aed-8c4d-
25d8cac76985", "type": "new S", "capacity": 100}, {"id":
"50ef7b65-4865-4358-aae5-3ca96411eb4f", "venueID": "5cf69a4b-
4872-4aed-8c4d-25d8cac76985", "type": "stop2", "capacity":
200}]}, {"id": "1565b52c-f8e7-42bf-9e10-b8122871777b",
"name": "Test Venue", "address": "South Yarra", "version":
"6", "sections": [{"id": "b7d6fc62-4fa8-4bad-a301-
206f93a91ea8", "venueID": "1565b52c-f8e7-42bf-9e10-
b8122871777b", "type": "1", "capacity": 200}, {"id":
"e1a35dd5-5bc0-424c-8570-8cdb24dd0caf", "venueID": "1565b52c-
f8e7-42bf-9e10-b8122871777b", "type": "F", "capacity": 100},
{"id": "95bedf5c-9f69-4719-bd28-208da6e114fb", "venueID":
"1565b52c-f8e7-42bf-9e10-b8122871777b", "type": "G",
"capacity": 800}]}]
```

Left: Load test of 20 threads on our implementation of the <u>Admin View All Venues</u> use case (which is temporarily modified to bypass the authentication filter).

Right: The data that would have been retrieved by one thread for the above use case.



View Results in Table

Name: ViewVenue (All) Result Table

Comments:

Write results to file / Read from file

Filename Browse... Log/Display Only: Errors Successes Configure

Sample #	Start Time	Thread Name	Label ↑	Sample Time(...	Status	Bytes	Sent Bytes	Latency	Connect Time(...
1	03:41:58.940	ViewVenueAll ...	ViewVenue (Al...	14570	✓	2250	163	14570	0
2	03:41:58.888	ViewVenueAll ...	ViewVenue (Al...	25487	✓	2250	163	25487	0
3	03:41:59.441	ViewVenueAll ...	ViewVenue (Al...	28138	✓	2250	163	28138	2
4	03:41:59.791	ViewVenueAll ...	ViewVenue (Al...	27803	✓	2250	163	27803	1
5	03:41:59.539	ViewVenueAll ...	ViewVenue (Al...	28832	✓	2250	163	28832	1
6	03:41:59.690	ViewVenueAll ...	ViewVenue (Al...	28751	✓	2250	163	28751	1
7	03:41:59.187	ViewVenueAll ...	ViewVenue (Al...	32966	✓	2250	163	32966	1
8	03:41:59.290	ViewVenueAll ...	ViewVenue (Al...	33273	✓	2250	163	33273	0
9	03:41:58.991	ViewVenueAll ...	ViewVenue (Al...	33637	✓	2250	163	33637	0
10	03:41:59.338	ViewVenueAll ...	ViewVenue (Al...	33322	✓	2250	163	33322	1
11	03:41:59.492	ViewVenueAll ...	ViewVenue (Al...	33212	✓	2250	163	33212	1
12	03:41:59.137	ViewVenueAll ...	ViewVenue (Al...	36452	✓	2250	163	36452	1
13	03:41:59.639	ViewVenueAll ...	ViewVenue (Al...	35950	✓	2250	163	35950	0
14	03:41:59.593	ViewVenueAll ...	ViewVenue (Al...	35996	✓	2250	163	35996	1
15	03:41:59.091	ViewVenueAll ...	ViewVenue (Al...	37519	✓	2250	163	37519	1
16	03:41:59.243	ViewVenueAll ...	ViewVenue (Al...	37367	✓	2250	163	37367	1
17	03:41:59.042	ViewVenueAll ...	ViewVenue (Al...	37877	✓	2250	163	37877	0
18	03:41:59.388	ViewVenueAll ...	ViewVenue (Al...	38347	✓	2250	163	38347	2
19	03:41:58.837	ViewVenueAll ...	ViewVenue (Al...	39310	✓	2250	163	39310	1
20	03:41:59.741	ViewVenueAll ...	ViewVenue (Al...	39408	✓	2250	163	39408	1

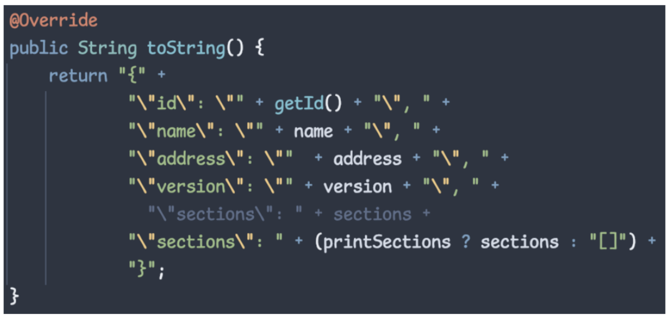
Scroll automatically? Child samples? No of Samples 20 Latest Sample 39408 Average 32910 Deviation 5852

Results from the above load test, showing an extremely slow average latency of 32910 ms.

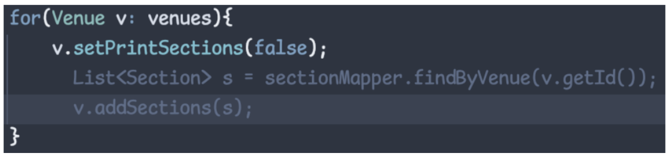
For our Event- and Venue-related use cases, the user does not always need the relevant EventSection and Section information; thus, lazy loading can be utilised. The system can first save the Events/Venues in its cache, and only load any relevant EventSections/Sections when their information is requested for.

- Client/Planner viewing events:

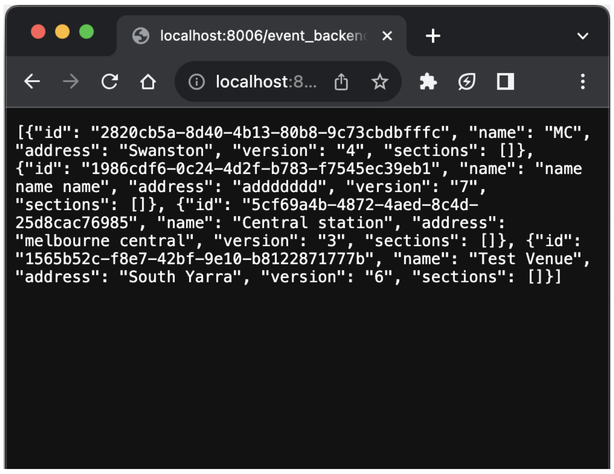
        - The user first only needs the event's general information to make a decision on whether they would like to perform a use case that requires EventSection information:
            - The client does not need EventSection when only browsing for Events; often an Event's name, location and Venue information is enough basis for them to reject the Event.
            - Similarly, an Event's name is enough for the planner to decide whether they would like to manage its EventSection information or not.
- Planner editing/creating event, Admin viewing venues:
    - Like the above, the user would not need any Section information when viewing Venues:
        - The planner may choose not to register an Event at a Venue entirely because of its name and location.
        - The admin may decide they do not need to view and manage a Venue's Section information.



```java
@Override
public String toString() {
    return "{" +
            "\"id\": \"" + getId() + "\", " +
            "\"name\": \"" + name + "\", " +
            "\"address\": \""  + address + "\", " +
            "\"version\": \"" + version + "\", " +
              "\"sections\": " + sections +
            "\"sections\": " + (printSections ? sections : "[]") +
            "}";
}
```



```java
for(Venue v: venues){
    v.setPrintSections(false);
      List<Section> s = sectionMapper.findByVenue(v.getId());
      v.addSections(s);
}
```

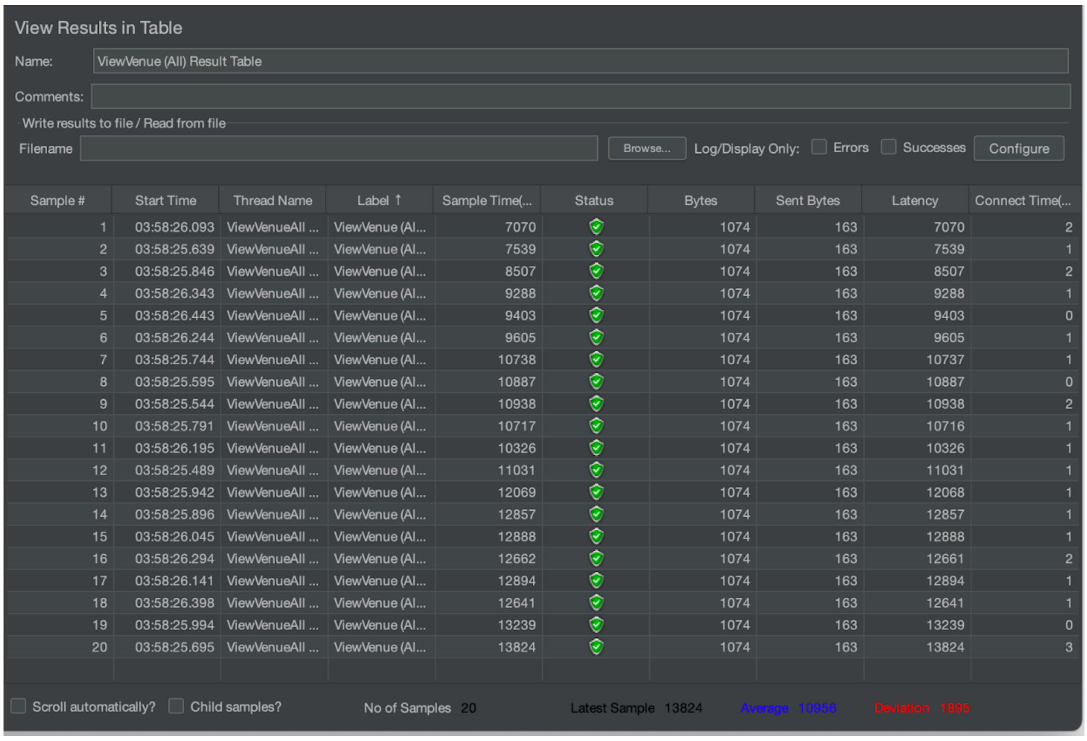


```
[{"id": "2820cb5a-8d40-4b13-80b8-9c73cbdbfffc", "name": "MC",
"address": "Swanston", "version": "4", "sections": []},
{"id": "1986cdf6-0c24-4d2f-b783-f7545ec39eb1", "name": "name
name name", "address": "addddddd", "version": "7",
"sections": []}, {"id": "5cf69a4b-4872-4aed-8c4d-
25d8cac76985", "name": "Central station", "address":
"melbourne central", "version": "3", "sections": []}, {"id":
"1565b52c-f8e7-42bf-9e10-b8122871777b", "name": "Test Venue",
"address": "South Yarra", "version": "6", "sections": []}]
```

Top left: Simple change to `objects.Venue.toString()` *that allows for a Venue's Section data to be omitted. This allows for lazy loading to be used on this data.*

Bottom left: Simple change to `VenueLogic.ViewAllVenues()` *that ensures Section data is not retrieved. This would make use of a lazy load implementation on a Venue's Sections.*

Right: The data that would have been retrieved by one thread for the aforementioned Admin View All Venues use case, after these modifications.


View Results in Table

Name: ViewVenue (All) Result Table

Comments:

Write results to file / Read from file

Filename

Sample #	Start Time	Thread Name	Label ↑	Sample Time(...	Status	Bytes	Sent Bytes	Latency	Connect Time(...
1	03:58:26.093	ViewVenueAll ...	ViewVenue (Al...	7070	✓	1074	163	7070	2
2	03:58:25.639	ViewVenueAll ...	ViewVenue (Al...	7539	✓	1074	163	7539	1
3	03:58:25.846	ViewVenueAll ...	ViewVenue (Al...	8507	✓	1074	163	8507	2
4	03:58:26.343	ViewVenueAll ...	ViewVenue (Al...	9288	✓	1074	163	9288	1
5	03:58:26.443	ViewVenueAll ...	ViewVenue (Al...	9403	✓	1074	163	9403	0
6	03:58:26.244	ViewVenueAll ...	ViewVenue (Al...	9605	✓	1074	163	9605	1
7	03:58:25.744	ViewVenueAll ...	ViewVenue (Al...	10738	✓	1074	163	10737	1
8	03:58:25.595	ViewVenueAll ...	ViewVenue (Al...	10887	✓	1074	163	10887	0
9	03:58:25.544	ViewVenueAll ...	ViewVenue (Al...	10938	✓	1074	163	10938	2
10	03:58:25.791	ViewVenueAll ...	ViewVenue (Al...	10717	✓	1074	163	10716	1
11	03:58:26.195	ViewVenueAll ...	ViewVenue (Al...	10326	✓	1074	163	10326	1
12	03:58:25.489	ViewVenueAll ...	ViewVenue (Al...	11031	✓	1074	163	11031	1
13	03:58:25.942	ViewVenueAll ...	ViewVenue (Al...	12069	✓	1074	163	12068	1
14	03:58:25.896	ViewVenueAll ...	ViewVenue (Al...	12857	✓	1074	163	12857	1
15	03:58:26.045	ViewVenueAll ...	ViewVenue (Al...	12888	✓	1074	163	12888	1
16	03:58:26.294	ViewVenueAll ...	ViewVenue (Al...	12662	✓	1074	163	12661	2
17	03:58:26.141	ViewVenueAll ...	ViewVenue (Al...	12894	✓	1074	163	12894	1
18	03:58:26.398	ViewVenueAll ...	ViewVenue (Al...	12641	✓	1074	163	12641	1
19	03:58:25.994	ViewVenueAll ...	ViewVenue (Al...	13239	✓	1074	163	13239	0
20	03:58:25.695	ViewVenueAll ...	ViewVenue (Al...	13824	✓	1074	163	13824	3

Scroll automatically? Child samples? No of Samples 20 Latest Sample 13824 Average 10956 Deviation 1895

Load test results of the same use case, after the above modifications (only without the step of implementing the Sections list as a value holder). Note the much faster latency of 10956 ms.

Single Connection for Transaction

With some of our use cases, our domain logic is implemented such that multiple `Connection`s are opened for each request. (e.g. Each EventSection's Section data, which is implemented as a `ValueHolder`, uses a different `Connection` instance. Retrieving multiple EventSection data at once such as in Customer View Events, would create many `Connection`s.)

This not only reduces the atomicity of our application and allows dirty reads on our database (which is a concurrency issue), this also decreases our system's performance. As opening and committing/rolling back each `Connection` takes time, doing so multiple times increases the latency. By making full use of our singletons like `UnitOfWork` and `DBUtil` to maintain a single `Connection` for all database operations, we can improve the latency of our app.

Pagination

Often the user would not be viewing all data at once; they would only be viewing a small subset of it at each point in time. When there is a large amount of data, we can both improve the system performance and increase usability by retrieving smaller, more manageable subsets of it rather than outputting everything to the frontend at once.

The frontend would track the currently-loaded "page number" and a size limit to send to the backend each time it requests for more data, and the backend would implement a route to output the corresponding portion of data from the database.

This would apply to all of our use cases that require READs of the data. Given the nature of our business domain, the amount of data would grow exponentially as time passes and as the user base scales up (e.g. one Venue may host hundreds of Events in a year, and each Event may have tens of thousands of Bookings).

Although this pattern requires making more API calls to the backend, for each request the amount of data sent from the database to the backend and from backend to the frontend would be much smaller. As more users use our system, this is crucial to keep the website more “live” (in appearance), so that a user doesn’t have to wait for all the data to be sent from backend to the frontend before their page loads.

For example, when an event planner views an Event’s Bookings, (of which there may be tens of thousands), they should not have to wait for all of them to load at once. Our app would seem much more responsive and be lower in latency if information is loaded by small batches.

Shared lock

A shared lock (also known as a read lock) is a type of lock that allows multiple transactions to read a database resource like a row or table simultaneously but prevents any transaction from modifying it until all shared locks are released. This ensures data consistency during read operations.

Application to Our System: In our music event management system, shared locks could be employed during operations where multiple users need to view venue availability or event details concurrently. By allowing concurrent reads while preventing writes, we ensure that users see consistent data without hindering the user experience due to unnecessary waiting.

Improved pessimistic read-write mechanism

Pessimistic locking assumes that conflicts between transactions are common and thus locks resources to prevent other transactions from accessing them simultaneously. A read-write lock mechanism includes shared locks for read operations and exclusive locks for write operations, ensuring that no other transaction can read or write to the resource until the exclusive lock is released.

Application to Our System: Our system could utilise an improved pessimistic read-write mechanism for operations that involve critical data changes, such as finalising a booking or modifying an event schedule. For instance, when an admin is updating the details of an event, an exclusive lock could prevent other transactions from reading or writing to that event, ensuring data integrity and preventing conflicts.

More usage of pessimistic lock over optimistic lock

- Due to the frequency of the booking creation and deletion, a pessimistic lock would be more helpful than an optimistic lock in avoiding doing work that will inevitably be lost due to the concurrency control.
- Retrieving all events at the same venue to avoid overlap in event timing is a resource-intensive process. A pessimistic lock mitigates the potential data inconsistency and excessive workload that could arise from concurrent access during such operations.

While optimistic locking is based on the assumption that conflicts are rare and resolves them after they occur, pessimistic locking prevents conflicts from happening in the first place by acquiring locks during the transaction.

Application to Our System: Given the real-time nature of event booking and scheduling, where data consistency is paramount, employing more pessimistic locking could be beneficial. For example, when retrieving all events at the same venue to avoid overlap in event timing, a pessimistic lock can ensure that the data does not change mid-operation due to concurrent access, thus maintaining data integrity and reducing the workload that arises from conflict resolution.

Potential Challenges: However, it's important to note that while pessimistic locking enhances data consistency, it can lead to decreased system performance if not implemented carefully. Excessive locking can cause contention, leading to bottlenecks where transactions are waiting for locks to be released.

Mitigating Lock Contention: To mitigate lock contention, our system could implement more granular locking at the row level instead of broader table-level locks. This approach minimises the locked resource scope, allowing greater concurrency. Additionally, implementing lock timeouts can prevent transactions from holding locks indefinitely, which can be a source of system unresponsiveness.

Conclusion: Incorporating shared locks, an improved pessimistic read-write mechanism, and a preference for pessimistic locks over optimistic ones can significantly enhance the robustness and reliability of our music event management system. By carefully managing lock granularity and duration, we can maintain high data consistency standards while minimising the performance impact, ultimately providing a smooth and reliable experience for users and administrators alike.

3. Design **principles**

Throughout the project, we adhered to the following design principles, all of which had direct or indirect effects on our performance:

- Single Responsibility Principle
- Open Closed Principle
- Liskov Substitution Principle & Dependency Inversion Principle

We did not use the following:

- Interface Segregation Principle
- Bell’s Principle
- Caching
- Pipelining

<u>Single Responsibility Principle (SRP)</u>

- Single Responsibility principle states that each class should have only one representative or purpose of the object. For example in our system, We have designed the classes in such a way that all the classes will hold a single responsibility .
- The Single Responsibility principle implemented in our application is as follows .
- The layering in the application starts from logging to the application, and implementing the business logic in the application server(controllers , unit of work) and communicating to the database(mappers), each class is designed with Single Responsibility principle.
- The security that is implementing while entering the application has class with mostly single responsibility
- The classes in business logic are also designed with single responsibility . For example, in our music system, Booking class holds the details about the booking . Event class holds

the details about the event. Similarly the same thing holds for Venue and Event Section which holds the details about Venue and Event Section respectively.

- For accessing the database, we have created the mapper section with their respective classes namely BookingMapper, Event Mapper, VenueMapper, EventSectionMapper.
- The classes that are created for UnitOfwork and in Controllers also have a single purpose to solve.
- The use of single responsibility in our application has improved our system since each class has its own responsibility so it is easy for maintaining code and to do modification.
- Testing becomes comparatively easier than creating classes which get bloated by adding functions which are not related to that class. Because of single responsibility, testing is more focused and unit testing is targeted for the single purpose of the class. Also debugging is easier while fixing the bugs , as each class holds only single responsibility, we can focus only on the classes that are related to the testing.
- It improves the maintainability of code. As each class is designed for a specific purpose, each class is easier to understand and maintain.
- Loose coupling is achieved by the single responsibility of class , as each class is designed for a specific purpose.

<u>Open Closed Principle</u>

The Open/Closed Principle (OCP) is a foundational tenet in software development, emphasising that classes should remain open for extension but closed for modification. (Bipin Joshi, 2016) This principle is integral for the evolution of software systems, especially those with increasing complexity. It addresses a critical challenge in software maintenance: introducing new functionality without adversely impacting the existing, stable, and tested components of the system.

Importance of OCP

As software systems mature and requirements change, there's a propensity to modify existing classes. However, such direct modifications come with risks, potentially leading to cascading impacts on other parts of the system. Thus, OCP promotes the enhancement of functionalities without altering the core code base. The principle ensures stability by preserving tested classes while still accommodating necessary extensions and the balance is achieved through abstraction mechanisms such as inheritance and interfaces.

The importance of OCP is underscored by its role in enhancing software maintainability and reusability. Statistics indicate that maintenance costs can be double the original development costs.(Xu & Hughes, 2005) Therefore, addressing challenges like rigidity, fragility, and other factors that hinder easy maintenance, is crucial. A well-implemented OCP approach fosters extensibility, flexibility, and pluggability in software design. This, in turn, elevates productivity, quality, and maintainability, the cornerstones of efficient software systems.

Implementation of OCP

1. Domain Object Class

The DomainObject class serves as a foundational piece of the online booking system's architecture. By abstracting common behaviours and attributes into this base class, you can design the system in a way that both adheres to and benefits from the Open/Closed Principle (OCP).

In our project, we introduced an abstract `DomainObject` class, preventing direct instantiation but allowing entities like `Event`, `Venue`, and `Customer` to inherit its attributes and behaviours. This facilitates the addition of future domain objects without altering the base class. Every new object automatically integrates the primaryKey attribute and the Unit of Work pattern functionality, ensuring consistent behaviour across entities. Despite its extensibility, the core logic of the `DomainObject` remains protected, safeguarding foundational behaviours from unintended disruptions.

By leveraging the `DomainObject` class, all domain entities in the system benefit from consistent interactions, particularly with the Unit of Work. This centralization not only streamlines maintenance by localising common behaviours but also reduces redundancy, leading to a more maintainable codebase. Furthermore, the integration with the Unit of Work pattern optimises database operations, batching transactions and ensuring only essential operations are executed, thereby enhancing system performance.

2. AbstractMapper Class

The `AbstractMapper` class encapsulates core database operations, ensuring adherence to the OCP. The class is designed to be extensible without modification. It's abstract methods like `doLoad`, `insertData`, `doUpdate`, and various SQL statement methods are intended for subclass implementations. This allows new entities or mapping mechanisms to be integrated by simply extending AbstractMapper and defining these methods, without altering the AbstractMapper's core logic. Concurrently, the core functionalities (e.g., `find`, `findMany`, `insert`, `update`, `delete`) are defined within this abstract class and are stable against changes, ensuring that new domain objects or modifications in database interactions don't necessitate changes to the AbstractMapper but are handled in its derived classes.

The AbstractMapper class offers numerous advantages to the application. It centralises common database operations, reducing code duplication and ensuring a consistent approach throughout the system. This not only streamlines maintenance but also facilitates easier and more effective testing, as the design's separation of concerns allows for specific method mocking or overriding in subclasses. Furthermore, the introduction of the loadedMap optimises performance by minimising redundant database fetches. The inherent flexibility of this architecture also positions the system favourably for future enhancements, such as caching or logging, which can be seamlessly integrated at the class level to benefit all associated mappers.

The AbstractMapper design notably enhances application performance through multiple strategies. The `loadedMap` serves as an internal cache, enabling direct retrieval of domain objects, thus circumventing potentially time-consuming database queries. By utilising methods such as `DBUtil.cleanUp`, the design ensures optimal database resource management, which minimises memory leaks and prevents database connection exhaustion. Furthermore, the `loadAll` method provides an efficient mechanism to load multiple domain objects simultaneously from databases, offering a more streamlined and efficient approach compared to executing separate queries for each object.

<u>Liskov Substitution Principle & Dependency Inversion Principle</u>

Liskov Substitution Principle (LSP) dictates that derivations of a class/interface should support the functionality of the classes/interfaces they are based on; Dependency Inversion Principle (DIP) dictates that rather than these specific, concrete subclasses/implementations, the system should make use of their superclasses/interfaces as much as possible.

In our system, we tried to adhere to both of these principles by creating a series of abstract classes and interfaces, which we then create concrete classes to inherit from/implement. We adhere strictly to LSP; although we find that sometimes we cannot follow DIP since our business logic requires specific functionalities from concrete classes (e.g. `BookingLogic` specifically using `BookingMapper` and `EventSectionMapper`), we still followed DIP from time to time:

- `AbstractMapper`
    - We created this abstract class as a superclass for our different mappers. All of our concrete mappers inherit and extend the CRUD operation methods and a table row-to-object `doLoad()` method, which they override to suit their own target objects. This supports LSP.
    - This class is primarily used by our `UnitOfWork` (UoW). Each time the UoW commits changes in its new/dirty/removed object register, it uses a helper singleton `MapperRegistry` to retrieve the object's relevant `AbstractMapper` via the object type. Then, it makes use of the `AbstractMapper`'s CRUD methods to make changes to the database. This supports DIP.
    - Our `Command` classes and our UserService also make use of them to retrieve objects that they need. Rather than calling the specific mappers, they use the `MapperRegistry` to retrieve object data, thus supporting DIP.
- `DomainObject`
    - We created this abstract class to represent our different objects (e.g. `Event`s, `Venue`s etc.). Each DomainObject can manage its primary key/s, as well as register itself onto the UnitOfWork. Our subclasses that extend DomainObject do not implement (through method overrides) these functionalities as the same logic applies to all subclasses; instead, we implement these functionalities on DomainObject as concrete methods. This makes these classes follow LSP by default.
    - Our `UnitOfWork` and `AbstractMapper` (AM) directly make use of this class. UoW stores abstract `DomainObject` in its registers, while AM handles `DomainObject`s in its CRUD operations. This supports DIP.
    - As mentioned above, some other classes retrieve object data from AMs; this data is stored as `DomainObject`s, which they then cast into concrete objects. This also supports DIP.
- `Command`
    - We created the interface `Command` and the abstract class `BusinessTransactionCommand` (which implements `Command`), as well as a series of concrete subclasses extending `BusinessTransactionCommand`. These classes act as a layer for pessimistic lock requests/releases in the <u>Admin Edit Venue</u> and <u>Planner Create Event</u> use cases.
        - `Command` contains the `init()` abstract method, which stores the HTTP request and response objects for processing. This method is implemented as

    a concrete method in `BusinessTransactionCommand`, thus allowing all subclasses to use it.
  - `Command` also has the `process()` method which requests/releases locks and runs business logic. Each concrete subclass implements this method according to their separate business logic.
  - `BusinessTransactionCommand` has the `startNewBusinessTransaction()` and `continueBusinessTransaction()` methods to help users maintain session identity across multiple requests in the same business transaction. These methods are implemented as concrete ones here, so the concrete subclasses are able to use them in their `process()` implementations.
  - Therefore, the concrete `Command` subclasses are able to function as abstract `BusinessTransactionCommand`s and `Command`s, thus following the LSP.
- When our different route controllers manage user requests, they create instances of concrete `Command`s and just store them as `Command`s, rather than the specific subclasses. Then, they simply call the `Command`'s `init()` and `process()` methods, the behaviour of which are determined by the implementation of these subclasses themselves. This adheres to the DIP.

These uses of LSP and DIP increases our response time and thus negatively impact our system performance:

- By calling our interface/superclass's methods, a slight overhead is incurred as the runtime has to locate the specific implementations of these to-be-executed methods.
- Moreover, in the case of the `DomainObject`s outputted by the `AbstractMapper`, having to explicitly cast them also incurs extra work.

However, this loss in performance is a worthy compromise for better extensibility and scalability of the system.

Interface Segregation Principle (Not used)

In the realm of software engineering, the Interface Segregation Principle (ISP) is a pivotal concept within the SOLID design principles, advocating for the creation of concise, client-specific interfaces as opposed to large, all-encompassing ones. The core idea behind ISP is that clients should not be forced to rely on interfaces they do not use.

Our current music event management system, a sophisticated amalgamation of patterns including Domain Model, Data Mapper, and Unit of Work, has not explicitly incorporated ISP. The DomainObject superclass and AbstractMapper in our system, for example, provide an extensive array of functionalities. While this has been functional, it also presents opportunities for refinement.

Rationale for Not Implementing ISP Initially: Our initial architectural strategy centred on robustness and comprehensive functionality. Key considerations included:

- Simplicity in Initial Development: Broad interfaces simplified the initial development phase, enabling a swift market entry—vital in the competitive event management sector.

- Familiarity and Consistency: A unified interface structure eased the learning curve for developers, fostering a consistent coding standard across the system.

- Resource Allocation: In the early stages, our resources were focused on developing core features rather than the granularity ISP requires.
- Anticipated Flexibility: The broad interfaces were expected to accommodate future requirements without significant restructuring.
- System Complexity and Scale: The initial complexity and scale of our system did not warrant the overhead associated with implementing ISP.

Potential Benefits of Implementing ISP: Despite the reasons for not initially adopting ISP, integrating it could markedly enhance our system:

- Reduced Coupling: Tailored interfaces for specific clients like Admins, Planners, and Customers could decrease dependencies, making the system more resilient.
- Enhanced Clarity and Maintainability: Smaller interfaces would render the system more accessible and manageable, particularly beneficial for new team members and during audits.
- Increased Flexibility: ISP could facilitate the addition of new features or the substitution of components with minimal impact on other system parts.
- Improved Performance: While runtime performance might not be significantly affected, compilation and deployment times could improve due to more decoupled components.
- Streamlined Testing: Narrow interfaces enable targeted testing, leading to more robust and precise testing processes.

While our system operates effectively without ISP, it is imperative to reassess our architecture as the system grows. The benefits of reduced coupling, enhanced maintainability, and increased flexibility that ISP offers become more relevant with scale.

In summation, our music event management system, though currently functional and robust, could greatly benefit from the strategic incorporation of ISP. As we look to evolve and scale our platform, embracing ISP should be a prominent aspect of our developmental roadmap, ensuring that the system remains adaptable, efficient, and easy to manage.

Bell’s Principle

Bell’s principle is a principle that is coined by Gordon Bell. His quote is “The cheapest ,fastest and most reliable components of a system are the ones that aren’t there”. He emphasised on having simple designs rather than complex designs since simple designs will have better performance than more complex ones. We should avoid adding more features and complexity until it's needed. So taking   the Bell’s Principle as reference, in the music system, the classes are designed in such a way that these classes are needed for the exact purpose. And we avoided adding any extra features which are not needed in the system. Also we designed the classes so that it will be simple to implement and so it will help if there is any change in the requirement in the future, because of the simple design , the system can accommodate the changes in the requirement. We also designed our classes starting from logging to the system, designing the domain classes , the classes of unit of work, the mapper classes are designed to fit the exact purpose. In this way, the system holds a simple design which is as Bell mentioned simple and efficient.

Pipelining (Not used)

Our backend system is somewhat equipped for pipelining. We have modified all of our static classes to be thread-safe, which allows the system to accommodate many requests at once without making significant errors. However, we did not implement any request pipelining in our frontend client, as we recognise that pipelining would do very little to improve our system's performance.

The way we designed our user flow left little room for asynchronous requests, since most use cases only require one GET call to the backend for all information necessary for the user to make a business decision and advance in the user flow. (For example, the Customer Create Booking use case only needs one call to retrieve Event data (which includes its EventSection pricing and availability.)

With the addition of our pessimistic lock, more requests are indeed being made at once for the relevant use cases. (E.g. Admin redirecting from View Venue page to Edit Venue page involves first calling to release venue lock and then calling to obtain venue write lock.) However, these (very scarce) requests require a strict sequential order to the requests sent; if we are to manage these sequential requests concurrently, it would largely complicate the thread handling behaviour in the system.

Although following this principle would do little to improve our system, if we are to continue improving the system (e.g. to implement lazy load) to the point where more requests are made, there would be value in implementing pipelining. For example, in the Planner Edit Event use case, lazily-loaded Events without EventSection data can be fetched first; then, when the Planner selects an Event to edit, its EventSection data and Venue data (for location change) can be loaded at once.

Caching (Not used)

Caching is a fundamental optimization strategy used in backend systems to improve the performance of software applications. Caching enhances software performance by temporarily storing data, often from databases or in memory systems. On subsequent requests, the system first checks this cache for faster data retrieval. To ensure data relevance, caching involves a time-to-live (TTL) mechanism that clears or updates stale data. Moreover, when the application alters data, related cache entries are invalidated to maintain data accuracy.

Rationale for not using caching

In our application we already made use of session state, both in frontend and in backend. In frontend we made use of the browser's `localStorage`, while in backend we implemented the `ApplicationSession` class for pessimistic locks on Venues during business transactions. Although we did not use the server session for other purposes, we could make use of it to store session-identified identity maps for each session. This would be helpful for all business transactions that involve more than one request to the backend.

For example, in the Admin Edit Venue use case, they would first go through the steps of viewing all venues before requesting for a chosen Venue's information from the backend. During the first step, the backend can store the retrieved venue information within the identity map in the `ApplicationSession` created for their session, so that when a user from the same session then requests for the venue's more specific information, the backend can easily locate it in the map and return it to the frontend right away, without having to query the database nor a cache.

Moreover, although caching offers substantial performance improvements for applications, it brings its own set of challenges. Key among these are the possibility of delivering outdated information unless correctly invalidated, heightened intricacy, especially when integrating distributed caching within microservices, memory usage concerns linked with in-memory caches, and the crucial task of preserving data consistency between the cache and primary data repositories to safeguard data accuracy.

To add caching onto our application would further increase the complexity of our application, which may both violate Bell's Principle and create additional room for human error. However, if our application's business logic is to scale up in complexity, a cache may be worthwhile:

Benefits of using caching

Fetching data from a cache (especially an in-memory cache) is much faster than fetching it from a database or from local storage. By serving data from cache, the application can drastically reduce the time it takes to process and respond to user requests. Meanwhile, frequently accessing data can put strain on databases. By offloading some of this demand to a cache, we can ensure that databases don't get overwhelmed, thereby preventing slowdowns or crashes.

Additionally, If our application fetches data from remote sources (like third-party APIs), caching can minimise the number of these potentially slow and costly network calls. Besides, If there's a sudden spike in traffic, a well-configured cache can absorb a significant portion of the increased demand, helping to ensure that the primary systems don't get overwhelmed.

References

- Bipin Joshi. (2016). *Beginning SOLID Principles and Design Patterns for ASP.NET Developers*. Berkeley, Ca Apress. *https://doi.org/10.1007/978-1-4842-1848-8_2*
- Xu, C.-W., & Hughes, J. (2005). Realizing the open-closed principle. *Fourth Annual ACIS International Conference on Computer and Information Science (ICIS'05)*, (pp.274–279). IEEE.