Performance Report

SWEN90007

Music Events System

Team: GGBond

In Charge of

Linjing Bi (1369370)

Jie Zhou (1442449)

Baorui Chen (1320469)

Liuming Teng (1292608)

Revision History

Date	Version	Description	Author
29/10/2023	04.00-D01	Initial draft	Liuming Teng
30/10/2023	04.00-D02	Adjust structure to the report	Linjing Bi
31/10/2023	04.00-D03	Add Description of each Patterns	Linjing Bi Jie Zhou Baorui Chen Liuming Teng
31/10/2023	04.00-D04	Add performance testing	Baorui Chen
01/11/2023	04.00-D05	Add impact of performance description	Linjing Bi Jie Zhou Baorui Chen
02/11/2023	04.00-D06	Finalized document content	Linjing Bi Jie Zhou Baorui Chen Liuming Teng
03/11/2023	04.00-D07	Proofreading	Jie Zhou

Content

1. Introduction **5**

1.1 Proposal 5

1.2 Target Users 5

1.3 Conventions, terms and abbreviations 5

1.4 Actors 5

2. Performance Testing **6**

3. Performance of Patterns & Principles **7**

3.1 Data Mapper 7

3.1.1 Description 7

3.1.2 Impact on Performance 7

3.2 Unit Of Work 8

3.2.1 Description 8

3.2.2 Impact on Performance 8

3.3 Identify Field 10

3.3.1 Description 10

3.3.2 Impact on Performance 10

3.4 Foreign Key Mapping 11

3.4.1 Description 11

3.4.2 Impact on Performance 12

3.5 Association Table Mapping 13

3.5.1 Description 13

3.5.2 Impact on Performance 14

3.6 Concrete Inheritance 15

3.6.1 Description 15

3.6.2 Impact on Performance 16

3.7 Authentication and Authorisation 17

3.7.1 Description 17

3.7.2 Impact on Performance 17

3.8 Concurrency control - Optimistic Lock 17

3.8.1 Description 17

3.8.2 Impact on Performance 18

3.9 Concurrency control - Pessimistic Lock 19

3.9.1 Description 19

3.9.2 Impact on Performance 19

4. Not Implemented Patterns & Principles **20**
4.1 Lazy Load 20
4.1.1 Description 20
4.1.1 Impact on Performance 20
4.2 Caching 21
4.2.1 Description 21
4.2.1 Impact on Performance 21
4.3 Pipelining 22
4.3.1 Description 22
4.3.2 Impact on Performance 22

1. Introduction

This document specifies the system's architecture Music Events System, describing its main standards, module, components, *frameworks* and integrations.

1.1 Proposal

The purpose of this document is to give, in a high-level overview, a technical solution to be followed, emphasizing the components and *frameworks* that will be reused and researched, as well as the interfaces and integration of them.

1.2 Target Users

This document is aimed at the project team, with a consolidated reference to the research and evolution of the system with the main focus on technical solutions to be followed.

1.3 Conventions, terms and abbreviations

This section explains the concept of some important terms that will be used throughout this document. These terms are detailed alphabetically in the following table.

Term	Description
Venue	A place where music events, concerts, or festivals are held. It can range from small clubs to large stadiums.
Venue Capacity	The maximum number of attendees or audience that a particular venue can accommodate for an event.
User Dashboard	A personalized page for registered users that displays their profile, booked events, wish list, and other user-specific details.
Booking Confirmation	A digital or electronic receipt provided to the user upon successful reservation of tickets for a music event.
Event Page	A dedicated page within the system that provides comprehensive details about a specific music event, including date, time, venue, ticket prices, and artist lineup.

1.4 Actors

Actor	Description
Administrator	Manage the system, all users and venues are controlled by the admin.
Event Planner	Users who manage the events' information.
Customer	Users who want to buy or cancel the tickets.

2. Performance Testing

Endpoint	method	Samples	Min(ms)	Max(ms)	Average(ms)	Throughput	Error(%)	Notes
/account	login	1000	97	946	342	52.8/sec	0	
/account	logout	1000	72	692	214	178.3/sec	0	
/customer	list	1000	87	1021	425	47.9/sec	0	
/customer	search	1000	104	1327	471	45.1/sec	0	
/customer	save	1000	62	694	263	92.9/sec	0	
/customer	update	1000	77	643	298	79.0/sec	0	
/customer	delete	1000	48	375	186	97.6/sec	0	
/event	list	1000	74	978	475	43.4/sec	0	
/event	search	1000	88	1307	529	55.3/sec	0	
/event	save	1000	210	564	639	25.0/sec	0	
/event	update	1000	65	980	306	97.2/sec	16.3	
/event	delete	1000	62	562	140	107.2/sec	0	
/order	list	1000	68	712	447	49.7/sec	0	
/order	search	1000	90	974	598	63.8/sec	0	
/order	delete	1000	154	1430	1172	34.4/sec	0	
/planner	list	1000	89	847	402	49.2/sec	0	
/planner	search	1000	79	1185	593	60.2/sec	0	
/planner	save	1000	58	574	293	103.2/sec	0	
/planner	update	1000	81	702	254	90.1/sec	0	
/planner	delete	1000	56	363	165	109.0/sec	0	
/purchase	save	1000	486	2072	1508	12.4/sec	28.7	
/venue	list	1000	93	823	367	63.2/sec	0	
/venue	search	1000	83	1043	704	64.8/sec	0	
/venue	save	1000	107	493	326	56.9/sec	0	
/venue	update	1000	90	429	389	82.3/sec	0	
/venue	delete	1000	49	365	329	79.3/sec	0	

3. Performance of Patterns & Principles

The subsequent segment delves into the discussion of various patterns and principles applied, and their impact on performance.

3.1 Data Mapper

3.1.1 Description

In our Music Event System application, the implementation of the Data Mapper pattern plays a pivotal role in ensuring a clear separation between the application's business logic and data storage concerns. By introducing a dedicated Data Mapper layer, we facilitate a seamless interaction between the in-memory representations of our entities and the underlying database, all while maintaining their independence.

The Data Mapper layer is comprised of specialized classes, namely ***AdminMapper, CustomerMapper, EventMapper, EventPlannerMapper, OrderMapper, PlannerMapper, TicketMapper, and VenueMapper***. Each of these classes corresponds to a specific entity in our domain and is tasked with executing CRUD operations, ensuring that any changes in the object's state are accurately reflected in the database, and vice versa.

The Data Source Layer acts as the bridge between our Data Mappers and the database, executing the SQL queries generated by the Mappers and returning the results. This layer abstracts the intricacies of database interaction, providing a clean and straightforward interface for the Data Mappers to communicate with the database. On the other hand, the Domain Layer houses our core business entities, encapsulating the business logic and rules of our application. These entities, including Admin, Customer, Event, EventPlanner, Order, Planner, Ticket, and Venue, remain completely focused on the business tasks they need to perform, without needing to worry about how the data is saved or retrieved.

By adhering to the Data Mapper pattern, our application achieves a robust architecture that not only promotes maintainability and scalability but also ensures that our business logic remains untainted by data access concerns. This clear separation of concerns lays the foundation for a flexible and efficient system, capable of adapting to future changes with ease.

3.1.2 Impact on Performance

➢ **Decoupling**: The Data Mapper pattern decouples the domain layer from the data source layer, which means changes in the database schema do not directly affect the business logic and vice versa. This separation can lead to easier maintenance and scalability but might introduce some overhead in data transformation between the layers.

- ➢ **Flexibility in Query Execution**: Since the Data Mappers are responsible for interacting with the database, they can be optimized for specific query operations, potentially improving performance for read or write-intensive operations.
- ➢ **Concurrency Control**: The Data Mapper pattern can help in implementing concurrency control mechanisms, ensuring that multiple users (Customers, Event Planners, or the Administrator) can interact with the system simultaneously without data inconsistencies.

3.2 Unit Of Work

3.2.1 Description

In our application, the `UnitOfWork` class is responsible for tracking changes made to objects during a transaction. It categorizes these objects into three lists: `newObjects`, `dirtyObjects`, and `deletedObjects`, based on the operations that need to be performed on them in the database. For example, when a new event is created in the application, an instance of the `Event` class is added to the `newObjects` list. Similarly, when an existing customer's details are updated, that `Customer` object is placed in the `dirtyObjects` list.

The `UnitOfWork` class also manages the database connection, ensuring it is properly opened and configured for transactions. When the `commit` method is called, it iterates through the objects in the lists, persisting the changes to the database. For instance, new events and customers are inserted, updated customer details are saved, and any deleted objects are removed from the database. All these operations are executed as part of a single database transaction, ensuring atomicity.

If any operation within the transaction fails, the `commit` method catches the exception, rolls back the transaction to maintain the database's previous state, and then rethrows the exception. This ensures that our application remains in a consistent state, even when errors occur.

By using the Unit of Work pattern, our application ensures that related operations are batched together, reducing the number of database round-trips and improving performance. It also maintains data consistency and enhances the reliability of the system, making it a crucial component of our online event booking application.

3.2.2 Impact on Performance

The Unit of Work (UoW) design pattern has been strategically applied to specific endpoints in our application to ensure data integrity during transactions. The primary aim is to consolidate operations into a cohesive unit, enhancing performance while ensuring a uniform database state.

The key endpoints that employ the UoW include:

- event/update
- event/delete
- customer/save
- customer/update
- customer/delete

Upon reviewing the performance results from section 2:

1. ***event/update***: Implementing UoW, this endpoint records an average response time of 306ms with a throughput of 97.2/sec. However, it exhibits an error rate of 16.3%, suggesting possible areas of improvement or potential issues with the UoW implementation. Upon further examination, we identified that two update functions were implemented in eventService. One function is dedicated to the unit of work, while the other addresses concurrency issues. This overlapping functionality is a potential source of the observed errors, and resolving this should be a priority in future development iterations.

2. ***event/delete***: The metrics for this endpoint are promising, showcasing an average response time of 140 ms and a throughput of 107.2/sec. Notably, there were zero errors encountered.

3. ***customer/save***: This endpoint, post the UoW integration, demonstrates an average response time of 263ms coupled with a throughput of 92.9/sec, with no errors to report.

4. ***customer/update***: With a steady performance, this endpoint presents an average response time of 298ms and a throughput of 79.0/sec. The error-free execution is a positive indicator.

5. ***customer/delete***: Among the tested endpoints, this particular one stands out with an impressive average response time of 186ms and a throughput of 97.6/sec, alongside a clean slate of zero errors.

The alternative to the UoW approach would involve allocating the task of updating objects to either controllers or service layer functions. Such a strategy can result in numerous minor database calls, potentially introducing latency into the transaction. Moreover, persisting a single database connection for the duration of the transaction might present performance bottlenecks, especially if other transactions are queued awaiting connection availability.

To conclude, the UoW pattern provides tangible benefits to several endpoints by safeguarding data integrity and potentially augmenting performance. Nevertheless, there remains significant room for improvement in our implementation of the Unit of Work pattern. Specifically, when it comes to order processing and the current approach to managing database operations. Besides, instead of using instanceOf in our UoW, we could refine our design by adopting interfaces or inheritance. This would allow us to invoke save, update, or delete methods more efficiently and in a more structured manner.

3.3 Identify Field

3.3.1 Description

The Identity Field design pattern focuses on assigning a unique identifier to each object in a system. This identifier, often termed as an "Identity Field," aids in tracking the object throughout its lifecycle. In databases, this pattern is particularly advantageous, where a unique identifier is crucial for distinguishing individual records, even if other attributes are identical. By leveraging the database's capability to generate unique values, applications can simplify many tasks, from basic CRUD operations to complex queries and joins.

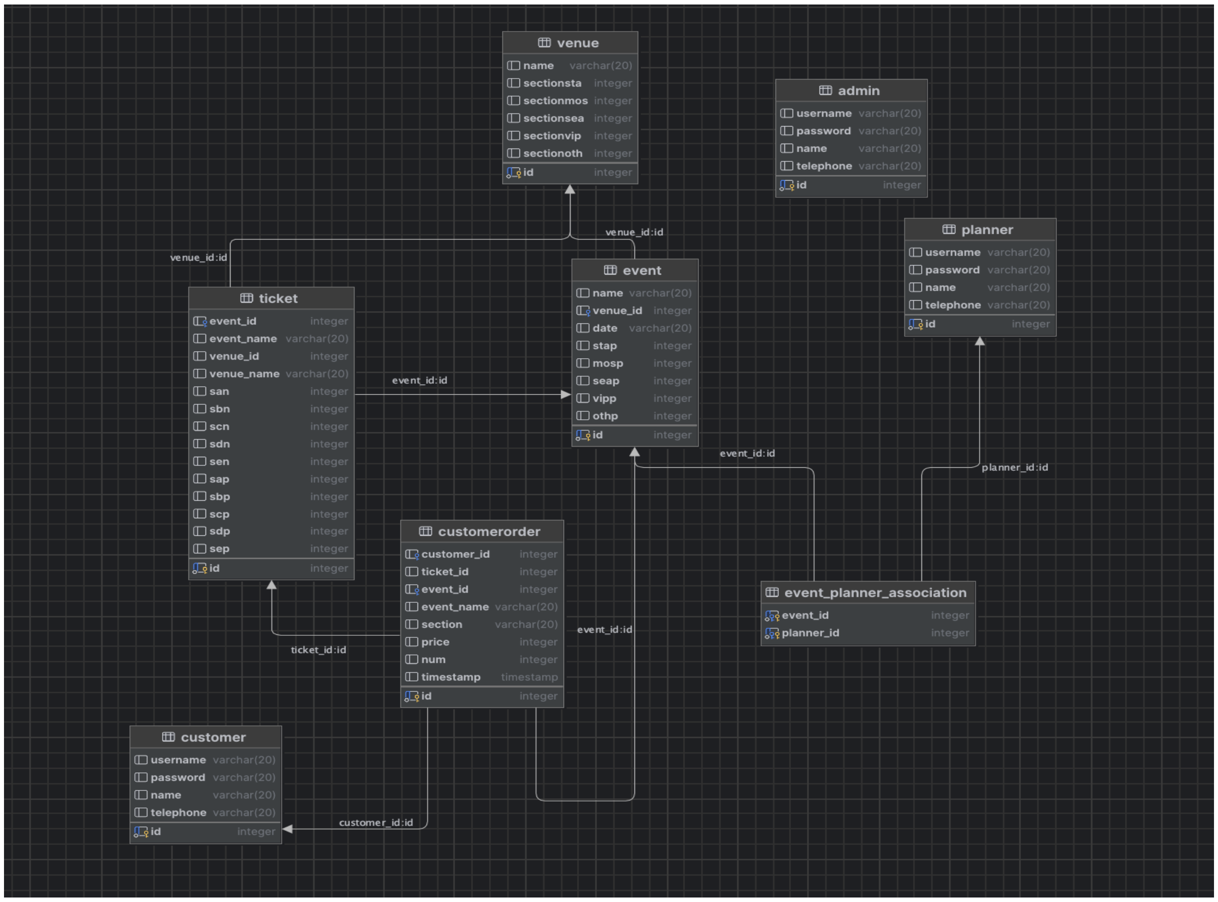


Figure 3.3.1 Identity Field Implementation

3.3.2 Impact on Performance

➢ **Optimized Record Retrieval**: By using the Identity Field pattern, the database can utilize indexed searches on the ID field, dramatically accelerating retrieval times. Indexes on unique ID fields are usually highly optimized.

- ➢ **Concurrency Benefits**: In multi-user environments, the Identity Field can reduce conflicts. When multiple users interact with the database, unique identifiers ensure that operations like updates or deletes target the correct record, reducing the risk of race conditions.
- ➢ **Reduced Overhead**: Automatically incrementing IDs eliminates the need for manual or application-level logic to generate unique identifiers. This reduces overhead and potential points of failure.

3.4 Foreign Key Mapping

3.4.1 Description

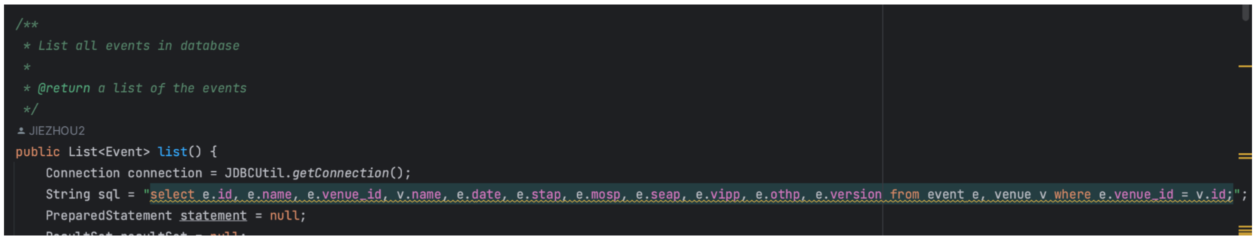
Foreign Key Mapping is a pivotal database design pattern that ensures the establishment and maintenance of relationships between tables in a relational database. A foreign key is a column or set of columns in one table that references the primary key columns of another table. The table containing the foreign key is called the referencing table or child table, while the table containing the primary key is referred to as the referenced or parent table. Foreign keys play a dual role:

- **Integrity Assurance**: They safeguard the integrity of the data in the database by making sure that the relationship between the tables is upheld.

- **Relational Linking**: They act as bridges, linking tables together, and facilitating complex queries that involve multiple tables.

The detailed implementation of the Music Event System shown in he following table

Table	Foreign Key	Reason
event	venue_ id	event can use venue_id to get the information of the venue
event_planner_association	event_id planner_id	It can save the information about “an event can be held by many planners”
ticket	event_id venue_id	ticket can get the information of the event and venue by their ids
custommer_order	event_id ticket_id customer_id	the oder table can save the information of the event, ticket and customer information by their ids


```
/**
 * List all events in database
 *
 * @return a list of the events
 */
JIEZHOU2
public List<Event> list() {
    Connection connection = JDBCUtil.getConnection();
    String sql = "select e.id, e.name, e.venue_id, v.name, e.date, e.stap, e.mosp, e.seap, e.vipp, e.othp, e.version from event e, venue v where e.venue_id = v.id;";
    PreparedStatement statement = null;
```
Figure 3.4.1 Foreign Key - Event contains venue id Example Implementation

3.4.2 Impact on Performance

Foreign keys play a pivotal role in maintaining referential integrity in databases. They ensure that relationships between tables remain consistent, even as data changes over time. Yet, when mismanaged or poorly indexed, they can also introduce performance overhead.

The following endpoints create new domain objects and hence evoke this key creation method:

- event/list
- event/save
- purchase/save
- order/save

Upon reviewing the performance results from section 2:

1. ***event/list***: This endpoint showcases an average response time of 475ms with a throughput of 43.4/sec. Given the 'list' functionality, it's possible this endpoint might be joining multiple tables, with foreign keys aiding in making the correct associations. However, this can potentially slow down the process, especially if there's an abundance of data to be retrieved.

2. ***event/save***: With an average response time of 639ms and a throughput of 25.0/sec, it appears slightly slower compared to some of the other endpoints. Implementing foreign keys might require checks against other tables to ensure data integrity when saving, which can introduce latency.

3. ***order/save***: This endpoint's performance metrics show a response time of 1172ms and a throughput of 34.4/sec. The elevated response time might be due to checks against related tables where the foreign key references are present, ensuring that orders are saved with valid references.

4. ***purchase/save***: This particular endpoint manifests a response time of 1508ms alongside a throughput of 12.4/sec. Such metrics might hint at extensive foreign key validations, especially if the purchase data correlates to several other tables. The interplay with foreign keys might be the reason behind its slightly sluggish performance.

Inherent to the integration of foreign keys is the series of checks and validations the database undergoes, especially during data insertions or updates. While this might extend certain response times, the overarching benefit of data consistency and integrity often compensates for such minor performance challenges. Furthermore, periodic optimization and indexing of these foreign keys is non-negotiable. Efficiently indexed foreign keys can drastically curtail overheads during database operations, resulting in a swifter query execution.

In conclusion, the presence of foreign keys undeniably leaves its footprint on the performance metrics of our endpoints. Their indispensable role in preserving data integrity may at times coincide with minor performance delays. However, consistent performance assessments and diligent database optimizations, particularly around indexing, can significantly mitigate the challenges posed by foreign keys, enabling a judicious balance between performance and data accuracy.

3.5 Association Table Mapping

3.5.1 Description

Association Table Mapping is a design pattern employed primarily to bridge many-to-many relationships in a relational database. Such relationships can't be directly represented using traditional table structures due to their inherent complexity. Instead, an intermediate, or "association", table is introduced. This table holds foreign keys to the tables that are being associated, thereby mapping the relationships. Each record in the association table typically represents a single relationship between the entities from the two associated tables.

We have strategically deployed the association table concept in the design of the event_planner_association table, as depicted in the subsequent figure. As a result, the intricacies tied to this table are seamlessly integrated into the EventPlanner class, further illustrated in the subsequent diagram.

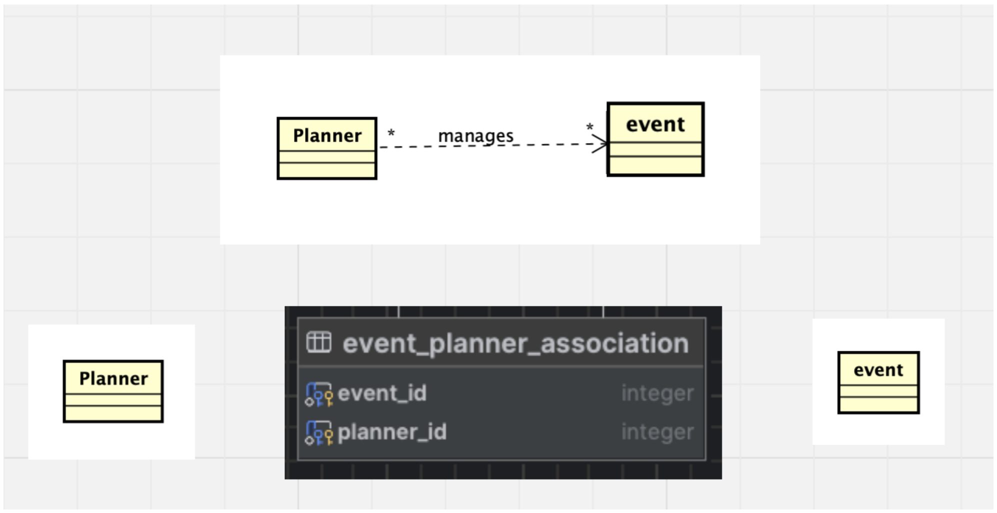
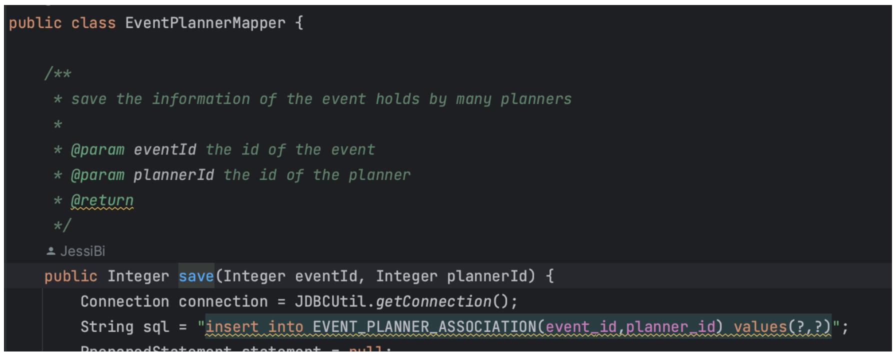


Figure 3.5.1 Associations table



```
public class EventPlannerMapper {

    /**
     * save the information of the event holds by many planners
     *
     * @param eventId the id of the event
     * @param plannerId the id of the planner
     * @return
     */
    JessiBi
    public Integer save(Integer eventId, Integer plannerId) {
        Connection connection = JDBCUtil.getConnection();
        String sql = "insert into EVENT_PLANNER_ASSOCIATION(event_id,planner_id) values(?,?)";
```

Figure 3.5.1 EventPlanner Class using the associated table

3.5.2 Impact on Performance

The incorporation of an association table in our database design warrants a crucial evaluation of its impact on the performance of key endpoints, especially within the domain of events. One of the primary motivators behind employing association table mapping was to seamlessly represent and handle the scenario where an event can be managed by multiple planners. This offers our application enhanced flexibility and accuracy. Below, we provide a detailed breakdown of the performance metrics for the following endpoints.

- event/save
- event/update
- event/list

Upon reviewing the performance results from section 2:

1. ***event/save***: With the deployment of the association table, this endpoint showcases an average response time of 639ms and boasts a throughput of 25.0/sec. Impressively, the error rate stands at zero, demonstrating stable and robust performance even when dealing with intricate data relationships.

2. ***event/update***: The endpoint produces an average response time of 306ms and a commendable throughput of 97.2/sec. Despite its performance, a significant error rate of 16.3% was observed. However, upon further inspection, it was determined that this error rate might not directly correlate to the association table mapping. Because we have two update functions that may cause errors.

3. ***event/list***: Crafted for retrieving the event list, this endpoint reveals an average response time of 475ms and a throughput of 43.4/sec. Mirroring the /event/save endpoint, the absence of errors underlines the robustness of the association table implementation in managing lists.

In conclusion, the introduction of association table mapping in the event_planner_association table has made a substantive impact on our performance metrics for the event-related endpoints. The zero error rate in two out of the three endpoints is particularly commendable. However, the error rate in /event/update underscores that while the association table amplifies clarity and flexibility in data relationships, regular performance reviews and optimization are pivotal to ensuring continued efficiency in the system.

3.6 Concrete Inheritance

3.6.1 Description

The Concrete Inheritance Pattern revolves around crafting a distinct database table for every concrete subclass. This design offers a solution to the complications of data redundancy, leading to enhanced query performance. When contrasted with other inheritance patterns such as single-table inheritance or class-table inheritance, this pattern stands out as it delineates the unique attributes and behaviors of each subclass.

By harnessing the Concrete Inheritance Pattern, each subclass has the liberty to possess its own independent data structure and business logic. This independence not only boosts the clarity and upkeep of the code but also facilitates the individual evolution of each subclass. This is attributed to the fact that every subclass has its dedicated database table, making it easier to manage over time.

For instance, in the provided diagram, the transition from the Webuser superclass to individual tables for Admin, Planner, and Customer epitomizes the essence of the Concrete Inheritance Pattern. It elucidates how this pattern simplifies data retrieval and storage by negating the necessity to navigate multiple tables to accumulate complete user data.

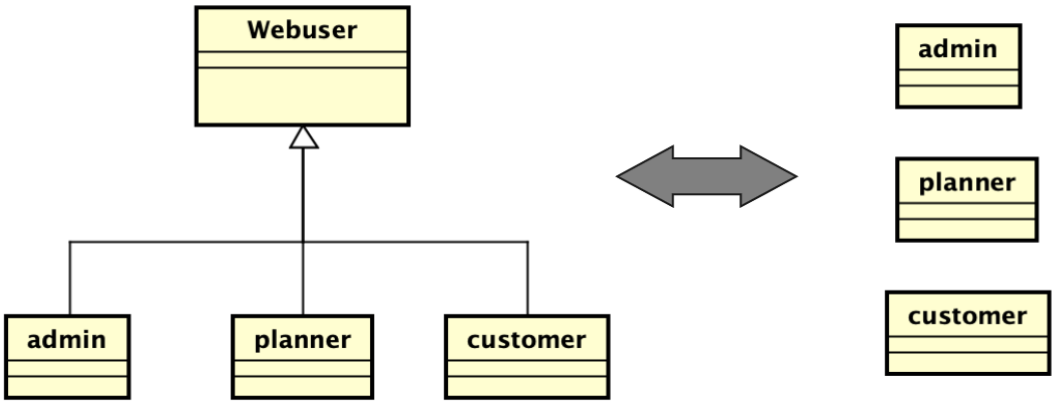


Figure 3.6.1 *Concrete Table Inheritance*

3.6.2 Impact on Performance

The concrete table inheritance pattern is a database design approach that involves creating individual tables for each concrete class within an inheritance hierarchy. In this schema, every table encompasses columns representing all attributes associated with the mapped classes, including those inherited from parent classes. This pattern shares similarities with class table inheritance, as it effectively mitigates issues related to wasted space and eliminates potential bottlenecks caused by superclass tables. However, it is not without its trade-offs.

One notable advantage of the concrete table inheritance pattern is its efficient use of storage space. By allocating a dedicated table for each concrete class, there is no redundant storage of attributes, ensuring optimal space utilization. This addresses a common concern in database design where storage efficiency is a crucial consideration.

Despite these benefits, there are inherent challenges associated with the concrete table inheritance pattern. Loading collections of objects of different types necessitates table joins or multiple calls, introducing a potential performance overhead. Retrieving instances whose type is not known upfront requires checking all the tables in the inheritance hierarchy, adding complexity to queries and potentially impacting response times.

3.7 Authentication and Authorisation

3.7.1 Description

In our Music Event System application, the implementation of Authentication and Authorisation modules takes place during user login, where the user's username and password are authenticated, and a permissions check is conducted when users access different pages.

Authentication and Authorisation are implemented by two distinct classes. The authentication module is responsible for verifying the login credentials, i.e., username and password, and returns one of three possible states: user does not exist, incorrect password, or correct. The authorisation module categorizes system users into three types: admin, planner, and customer. These three categories of users have different levels of access rights and maintain separate permissions pools. Admins have access to all pages, while planners and customers have different permissions according to the system's requirements.

Permission checks are carried out within the control module. Once a user is successfully authenticated and logged in, the user's type is recorded in the session. Subsequently, when the user calls a relevant method, the authorisation module's checkPermission function is invoked to conduct a permissions check. For example, if a customer user attempts to call the event/delete method without having the requisite permissions, the action is denied, and the user is redirected to a page indicating they lack the necessary permissions.

3.7.2 Impact on Performance

The impact of Authentication and Authorisation on system performance is minimal. As shown in Section 2, the authentication module requires access to the database to verify the correctness of the username and password, which means the average time taken for account/login is longer than for account/logout.

The impact of the authorisation module on system performance is not significant because the authorisation process does not require database access but only necessitates a lookup in the permission pool. Given that the current system functions are relatively straightforward, the permissions pool is quite small, so it does not significantly affect the system's performance.

3.8 Concurrency control - Optimistic Lock

3.8.1 Description

The privilege to modify ticket prices should be restricted exclusively to Event Planners associated with the specific event, ensuring the accuracy and reliability of pricing information. Additionally, when adjusting ticket prices, the system should ensure that the modification does not create conflicts or discrepancies with ticket prices or promotions of other events scheduled at the same venue and time, thereby preventing potential pricing differences or confusion. Concurrency occurs when multiple planners are working on the same event at the same time in the event management interface. However, the amount of concurrency is not high, and there are more reads than writes. That's why optimistic locking is appropriate. Optimistic lock is very optimistic when operating data, and it is believed that others will not modify the data at the same time, so optimistic lock will not lock the data before updating the data. Only when the data is updated, judge whether the data is modified. If the data is modified, it will give up the current modification operation. The optimistic lock was implemented to combat any conflicts in event details that can arise when the same event is being modified by a different planner. The system allows for multiple planners to be assigned as co-planner for a single event, and there can be situations where both co-planners are making changes to the event at the same time. In order to ensure that the co-planner is modifying the most recent event details, the optimistic lock was chosen so that when there are inconsistencies between event versions on commit, the co-planner is forced to fetch the latest version and start again.

3.8.2 Impact on Performance

In analyzing the performance test results, a notable observation is the relatively high throughput achieved with the use of optimistic locks. This efficiency can be attributed to the characteristic of optimistic locks that eliminates the need for queuing mechanisms, opting for a more direct approach by throwing exceptions when conflicts arise. This deviation from traditional locking mechanisms contributes to a streamlined process, enhancing system responsiveness and resource utilization.

Optimistic locking stands in contrast to pessimistic locking, where a lock is acquired on a resource before any modifications are made. In an optimistic locking strategy, there is an inherent trust that conflicting modifications will be infrequent. Consequently, transactions are permitted to proceed without acquiring locks during the read phase, fostering a higher degree of concurrency and reducing contention for resources.

One key aspect of optimistic locking that directly impacts performance is its exception-based conflict resolution mechanism. Instead of maintaining queues and waiting for resources to be released, optimistic locking relies on detecting conflicts at the time of update. When a transaction attempts to modify a resource that has been concurrently altered by another transaction, the system throws an exception. This exception serves as a signal to the user that a conflict has occurred, prompting them to roll back the operation and retry.

3.9 Concurrency control - Pessimistic Lock

3.9.1 Description

In the music event system, we categorize concurrent situations into two types. One is high concurrency, requiring strong consistency, such as ticket purchasing. Since ticketing operations are not atomic, when users are trying to buy tickets for popular music events, there might be cases of duplicate purchases of the same ticket or buying a quantity that exceeds the available stock. This situation involves monetary transactions, so strong consistency is necessary. Therefore, for ticket purchase operations, we use pessimistic locking. As the name suggests, the strategy of pessimistic locking is very pessimistic. Once a thread is ready to make changes to certain data, it assumes that concurrency issues will definitely occur. Therefore, the thread applies pessimistic locking to the data at the beginning of the operation and releases the lock when the operation is complete. When lock acquisition fails, it either waits or exits. Pessimistic locking achieves strong consistency by serializing operations but also comes with a performance trade-off. Because Redis performs well and is easy to implement for distributed locking, we use the Java client Jedi for Redis to implement pessimistic locking.

3.9.2 Impact on Performance

In the realm of concurrency control, pessimistic locking stands as a robust mechanism for preventing conflicting behavior at the cost of introducing significant performance considerations. Unlike its optimistic counterpart, pessimistic locking transforms the execution of transactions from parallel to serial, where each transaction acquires a lock on the data it intends to modify before proceeding. This cautious approach ensures that conflicting modifications are avoided, maintaining data integrity but at the expense of reduced parallelism and potentially increased contention.

The key characteristic of pessimistic locking lies in its proactive acquisition of locks during the read phase of a transaction. When a transaction desires to modify a particular resource, it first requests and obtains a lock on that resource. This lock acts as a gatekeeper, preventing other transactions from concurrently accessing or modifying the same resource until the lock is released. While this methodology effectively prevents conflicting behavior and ensures the consistency of the data, it inherently limits parallelism by enforcing a sequential execution of transactions.

The impact of pessimistic locking on performance is particularly pronounced in scenarios where there is a high degree of contention for resources. The serialization of transactions can lead to bottlenecks, as each transaction must wait for the release of the locks held by others before proceeding. This queuing effect introduces latency and can result in decreased throughput, especially in situations where multiple transactions are vying for the same resources.

4. Not Implemented Patterns & Principles

4.1 Lazy Load

4.1.1 Description

Lazy loading is a technique for optimizing page load times by deferring the loading of certain parts of a webpage until they are actually needed. This approach ensures that entities or objects are only retrieved from the database when necessary, avoiding the need to load all potentially associated data at the outset. It is particularly effective when combined with identity mapping, as it allows for objects to be loaded from the database as needed while keeping track of which objects have already been loaded to prevent redundant operations.

The Music Event System has not implemented lazy loading primarily because integrating this feature would require significant modifications to the existing codebase, a challenge compounded by the time constraints of the project. Consequently, lazy loading was only considered during the design phase and was not actualized in the final product.

4.1.1 Impact on Performance

If implemented, lazy loading could improve the performance of certain methods:

- customer/list
- planner/list
- venue/list
- order/list
- event/list

For these methods, lazy loading could be leveraged to materialize a selective data retrieval paradigm. Take the customer/list method as an example; it could be configured to present only the most rudimentary customer data, such as names and IDs, at first blush. The full spectrum of customer information could be subsequently fetched on demand when detailed scrutiny is invoked by the user. This partial data loading approach stands in stark contrast to the system's prevailing methodology, which indiscriminately pulls and exhibits the entire data set from the get-go, engendering unnecessary data processing and bandwidth consumption. Through the prism of lazy loading, the system would reap the benefits of a more streamlined resource allocation, manifesting in enhanced list-method efficiencies and precipitously reduced latency for initial data presentation.

4.2 Caching

4.2.1 Description

Caching is a strategy of temporarily storing data in a fast access storage system so that on subsequent requests for the same data, it can be served from the cache rather than retrieved from the original data source, like a database, each time. Caching can significantly improve performance, especially where data changes infrequently or read operations far exceed write operations.

While the Music Event System has not implemented a full caching mechanism, we have applied a similar caching approach through technologies like Redis to address synchronization issues. Caching in this project could be used to lessen the demand for database access, particularly for displaying customer and planner list information, which does not change often.

Considering that the performance improvements from caching may not be significant and would require changes to the existing code, we chose not to implement this technique.

4.2.1 Impact on Performance

The impact of caching would vary across different methods. For data that seldom changes:

- /venue
- /customer
- /planner

In the context of the Music Event System, venue, customer, and planner information is relatively static, with infrequent updates required. Implementing caching for these endpoints can dramatically decrease the frequency of database queries. This would not only improve the system's response time but also lower the burden on the database, leading to more efficient resource utilization. By avoiding repetitive and unnecessary data fetches, caching ensures that server resources are freed up to handle more intensive tasks or to serve a higher number of concurrent requests, thereby enhancing the scalability of the system.

For data that changes frequently:

- /order
- /event

Orders and events represent the dynamic segment of the system's data, with frequent changes reflecting new transactions or updates to event details. Here, the caching strategy would need to be more sophisticated, perhaps with a lower time-to-live (TTL) for cache entries or utilizing a smart invalidation scheme that updates the cache only when changes occur. While caching might not reduce the database load as significantly as it would for static data, it can still offer performance improvements by serving a portion of the read requests from the cache, especially if there is a pattern to the reads, such as retrieving the most recent orders or upcoming events more often than older data.

4.3 Pipelining

4.3.1 Description

Pipelining is a performance enhancement technique that involves the parallel processing of multiple requests. It involves asynchronously sending multiple requests without waiting for each one to complete before proceeding to the next. By doing so, pipelining can reduce the waiting time to complete functions by allowing the application to continue operating while waiting for the results of requests (assuming they are not needed immediately), thereby improving performance. However, implementing pipelining adds complexity to the system, requiring multi-threading to handle asynchronous message passing and processing.

Pipelining has not been implemented in the Music Event System. A potential implementation could involve processing multiple requests in methods that require multi-step operations, such as purchase/save, without waiting for parts of the data to be ready and then committing once all data operations are complete. Given the complexity of implementing pipelining and the need for substantial changes to the existing code, we opted not to adopt this technique.

4.3.2 Impact on Performance

The integration of pipelining within the Music Event System's infrastructure could be transformative, particularly when applied to the system's more intricate methods. By curtailing the cumulative wait times associated with individual requests, pipelining could notably reduce the interference these requests might pose to concurrent data operations. This reduction in latency has the capacity to markedly amplify the system's operational efficiency. Through pipelining, the system could witness a significant uptick in throughput and a corresponding descent in response times, resulting in a smoother, more responsive user experience.