Performance Report Document

Submission 4 Specification

Runtime Terror

SWEN90007 SM2 2021 Project

Release Tag:

https://github.com/SWEN900072021/Covid19BookingSystem/releases/tag/SWEN90007_2021_Part4_RuntimeTerror

Team Members:

Thomas Chen | 916183 | thomasc3@student.unimelb.edu.au

Kah Chun Lee | 860318 | kahl2@student.unimelb.edu.au

Jay Parikh | 864675 | jparikh@student.unimelb.edu.au

Jad Saliba | 1014680 | jads@student.unimelb.edu.au

Table of Contents

Design Principles

Bell's Principle

The team had to find a balance between implementing simple design patterns and more complex ones. The team realized that there is a tradeoff between performance and maintainability in adding extra layers of complexity and thus the aim was to implement complex patterns only when strictly needed or when required by the project specification.

It should also be noted that there were some occurrences where implementing a slightly more complex pattern would result in a much higher extensibility for our project such as the case with data mappers. Even when that implementation negatively affected performance, the team felt that it was justified to have a more complete and scalable solution.

On another note, the team also took the scope of our project into consideration when deciding which patterns to use. Some patterns affect specific properties differently such as data normalization which would have a far bigger performance on write-heavy applications. However, since the system is read-heavy, implementing data normalization resulted in a better overall performance.

Caching

Caching is where data is transparently stored in faster-access locations to reduce the amount of time taken to access or retrieve the data. Generally the faster-access location is temporary and non-persistent so if the server crashes or is shutdown, this information can be lost. This means that generally copies of frequently accessed data are saved in caches to improve performance through reducing retrieval times, as well as having to avoid reading from the persistent storage as much (database) which improves performance there as well.

Caching works based on the two properties that most streams of access exhibit: temporal locality and spatial locality. Temporal locality means that if an item has recently been accessed, it will likely be accessed again soon, which means that keeping a copy of a recently accessed item in the cache will likely yield a positive payoff. Spatial locality is when an item which has been recently accessed, the items or objects near it (related through various properties) are also likely to be accessed, which means that when caching a specific item, it can also be worth it to cache objects that are ‘nearby’ as well. For example, the Covid-19 Booking System were to implement caching, and a timeslot on Saturday at 9am was accessed and cached, temporal locality dictates that this timeslot is probably going to be accessed again soon as it may be a popular timeslot. Caching through spatial locality would mean that possibly all timeslots from all HealthCareProviders at Saturday at 9am would also be cached, as it could be likely that those will be retrieved soon too.

Considerations to take into account are also the strategy to remove ‘stale’ records from the cache. This is needed because caches are generally limited in the amount of storage available, and room needs to be made for more relevant or newer records. The first strategy is a simple time-to-live, where if an item has not been accessed in a certain amount of time, it will be removed from the cache. A use of a queue can work, where there are a limited amount of spots available, and each time a record is accessed, it is pushed to the front of the queue, allowing more recently accessed records to populate the cache, while avoiding any risk of running out of memory. While these strategies (and more that have not been mentioned) do not directly affect the performance of the system, they are important to ensure that the cache works best for the context of the specific system, and often there needs to be some trial and error to find which cache expiration strategy best fits for the system at hand.

Moreover, analysis can be done to figure out the best policies and expiry rules on the cache, which will improve the performance and effectiveness of the cache itself. The use of the expected access time formula can aid in checking if the access time between using the cache or the database is quicker, which can be used to compare different strategies or whether to implement caching at all. The formula where T_{fast} is the time taken to access the cache, and T_{slow} is the time taken to access the database, T_{check} is the time taken to check if the cache already has the information, hit_rate is the probability the cache has the information already, and T_{exp} is the expected access time is:

$$T_{exp} = T_{check} + (hit\_rate \times T_{fast}) + ((1 - hit\_rate) \times T_{slow})$$

$$T_{exp} = T_{check} + T_{fast} + ((1 - hit\_rate) \times (T_{slow} - T_{fast}))$$

Caching is worth it if $T_{exp} < T_{slow}$ and if the cost within the cache’s space is acceptable. The factors within the formula such as hit rate can be increased by allowing for more space in the cache, and time taken to access cache and database can both be improved by better hardware.

Within the context of the Covid-19 Booking System, caching can definitely be implemented to improve the read performance of the system, which will give a huge boost to the overall performance as well, due to the read-heavy nature of the system. The obvious candidate for caching would be timeslots, as not every timeslot will be booked, and if a timeslot is retrieved, it will be likely that someone else will want to retrieve that timeslot as well. The system could also implement a caching rule where timeslots at the same time, timeslots from the same VaccineType, or timeslots from the same HealthCareProvider are also cached, based on the principle of spatial locality.

Furthermore, if profile pictures or icons are implemented for HealthCareProviders, the use of content delivery networks (CDN), which are a distributed network of servers that allow ‘local’ caching for users by hosting them close to them geographically. This would also work if urls to the HealthCareProviders websites were introduced as a part of a HCP’s description/information page, and caching could be implemented on clicks of links to certain pages.

Overall, caching would be a highly effective way for the system to increase its performance, as it reduces the amount of time required for users to access resources, and while the policies and strategies in the caching themselves may require more in depth analysis of use cases and context, mixed with some trial and error, caching itself is relatively simple to implement, and for quite low risk, can yield prosperous results quickly.

Pipelining

Pipelining is a technique that executes requests in parallel rather than sequentially to turn the performance bottleneck from latency to throughput. In an arbitrary multi-request process, when done sequentially, requests have to essentially wait in line to be executed. The factor that decides how long a request has to wait before being executed is latency (length of time it takes to complete a task). Sequencing of requests is usually applied when there is a dependency between requests, i.e. a request needs information from the previous request.

Pipelining is a technique that aims to reduce the total time of executing multiple requests by executing them in parallel which then changes the bottleneck from latency to bandwidth. This change in bottleneck is due to the change in how the total time of execution changes from execution time of all requests (sequential) to execution time of the slowest requests. The execution time of individual requests might be slower in pipelining as it is limited by the available bandwidth, which is the number of tasks the system can complete in a given amount of time. Since the requests are executed in parallel, it is not a given that the hardware, in particular the network, is able to execute all parallel requests with the same execution time as executing them sequentially. Each request requires a certain amount of resources and as hardware and network are finite resources, the performance of pipelining will be bottlenecked by bandwidth.

In the system, use cases that have multiple requests that are independent of each other are pipelined with the use of the unit of work pattern. Unit of work bundles the requests and executes them in one go which in theory improves the overall throughput of the system as there is sufficient bandwidth to support the larger unit of work request. However there are certain parts of the system where requests are dependent on each other, for example creating a Health Care Provider Account where creating an entry in the parent table Account is needed before creating an entry in the child table Health Care Provider.

Design Patterns

Association Table Mapping

Description

Association Table Mapping is used as a link table that shows the association between two other tables by storing their primary keys. The team decided on using association table mapping to connect the 'vaccine_type' with the 'question' table to link each VaccineType with its respective list of Questions. It is also used to track the associations between VaccineRecipients and VaccineTypes they have received. Association Table Mapping can also boost performance since these tables do not store non-essential information. However it does increase complexity due to the involvement of a new table.

Comparison

- **Foreign Key Mapping:** Foreign Key Mapping is simpler to implement and needs one less join so it would have been suitable to use here. However, Association Table Mapping was included to account for the many-to-many relationships linking the 'vaccine_type' and 'question' tables as well as the 'vaccine_recipient'' and the 'vaccine_type' tables. Association Table Mapping also ensures a simpler association. Using Association Table Mapping was ideal as it avoids creation of a more complicated database schema. Association Table Mapping can also be used for one-to-many relationships which makes it more general and reduces the amount of complexity that the system needs to handle.

Conclusion

Association Table Mapping is best used when having many-to-many associations as it is the only pattern suitable for this use case. It provides a simple association between tables which results in a clean database schema. Regardless of the minor complexity issues that emerged due to this pattern, it made sense to implement it for this particular use case as it made the database schema much cleaner and easier to work with.

Data Mapper

Description

A data mapper is a layer which is used to build an independent relationship between the object class and the database. The insertion of the data mapper layer happens between the domain layer and the data source layer. This reduces coupling as the domain layer would operate without being aware of any database actions. This decoupling also allows the system to swap to a better performing database engine in the future.

Using Data Mappers works well with the Domain Model because it ensures that domain objects can be modified without needing to understand how they're actually stored in the database. This is especially useful when using complex mappings. Adding an additional layer between the object class and the database makes the system more extendable but that comes at the cost of slightly lower performance.

Comparison

- **Not using Data Mappers:** No de-coupling which means that changing the persistence logic would have an effect on the business logic. However, not using Data Mappers can have a minor increase in performance due to not having this additional layer.
- **Row Data Getaway:** Since we decided on using a domain model, using Row Data Getaway would not have been suitable here.
- **Active Record:** This could have been used as a different database access pattern as it works well with simple domain models. This would have been a suitable option in our project. However, when taking into consideration potential expansion of the project, the team ultimately decided on using Data Mappers as having a separate layer between the domain layer and the data source layer made sense in that regard. This would secure better scalability for the application going forward.

Conclusion

Data Mappers are the most suitable choice when it comes to more complex domain models and was our pick due to its capability of scaling better with future project expansions. We recognized that Data Mappers add a layer of complexity which can have a negative impact on performance but we felt that this minor decrease in performance was worth having for the sake of extensibility.

Embedded value

Description

Embedded value is used for objects that don't necessarily need a table in the database. It embeds the fields of an object to values inside that object's owner. This pattern is used in the system to map the different fields of a vaccine recipient's address (street, postcode, country, etc) into one address object, while flattening out these details at the database level.

Comparison

- **Not using Embedded Value:** The team would have to add a new table for the Address object for the value that is being embedded. This increases the number of read queries required to obtain a whole domain object. The technique used in embedded values is called data normalization which is the act of grouping data together with the aim of improving read performance. Even though this slightly compromises write performance,

due to the system's read-heavy nature, this results in an overall improvement in performance.

- **Serialized LOB:** When discussing Embedded Value, it makes sense to compare it with the Serialized large object (LOB) pattern. Both these patterns would have achieved similar results within our project but we ultimately decided on using Embedded Value due to the following considerations:
    a. Embedded Value works well with a few simple dependents while Serialized LOB is more suitable to complex structures and subgraphs.
    b. Embedded Value excels at allowing SQL queries to be performed against the values in the dependent object which made it more suitable for the scope of this project.

Conclusion

Embedded value is best used for objects that don't require a full table in the database. Our use case of mapping the different fields of a vaccine recipient's address into a simple structure doesn't require a Serialized LOB. This is what makes using embedded value(s) a good choice here.

Inheritance

Description

In terms of inheritance there are three main patterns to consider: single table inheritance, class table inheritance and concrete table inheritance. While these patterns do not have to be mutually exclusive, to compare the performance and discuss trade-offs, it can be useful to initially weigh them up separately.

Comparison

- **Single table inheritance** is when a class structure that has an inheritance hierarchy is persisted as a single table that flattens out the hierarchy and stores all attributes from the multiple classes within one 'monolithic' table. While it means the table itself is kept simple, and queries are easy to execute and write, with the lack of joins required, there are performance trade-offs in using such a pattern. Storing all information in one table means that additional fields that may not be required on a read will also be stored in-memory, and it also means that since all the columns are in one table, if locking is implemented, it may occur more frequently, which will negatively affect performance and liveness of the system. Furthermore, the lack of joins is actually a detriment to performance when the database grows in size, and more and more rows get added. While joins can increase efficiency in terms of memory and process time, it can also become more inefficient if the tables are not designed and structured properly, which is often the case in single table inheritance, where there is a lot of space and processing overhead. This is because join queries speed up performance as the number of rows increases, as the query planner can utilize joins and indexes to select fewer rows than if

joins were avoided, which as mentioned before, further reduces space complexity in-memory.

- **Class table inheritance** is where each class in an inheritance hierarchy is assigned to a separate table, with each class mapping directly to its own individual table within the database. The benefit of class table inheritance is that only the relevant information is retrieved on queries, which saves on space in memory and results in faster queries, especially as the database increases in size. In contrast with the single table inheritance pattern, joins are commonly used to retrieve information on child classes, as the server will often need both the parent and child classes attributes (however, this is not always the case as will be mentioned soon). Having to utilise joins on most child requests is not ideal, and in this specific case, will often result in lower performance, as queries will need to utilize joins which could be avoided if the information was stored in one location. Furthermore, there may also be bottlenecks occurring in superclass tables, which are used in each query.
- **Concrete table inheritance** is where each inheritance hierarchy of classes is given one table per concrete class within the hierarchy. The benefit in performance is that there is no need for joins as all columns in a table are relevant, which also means that there is no wasted space when loading into memory (due to lack of irrelevant or non-related fields being loaded like in single table inheritance). It takes the best of both single table inheritance and class table inheritance as it eliminates the need for joins, as well as keeping all the relevant columns into one table, which means doing a query has no wasted memory. It also spreads the load, which means there is no single bottleneck for the database, as all the tables are required to be accessed for each specific concrete class. However, there are certain disadvantages of concrete table inheritance, which include heavy performance implications in scenarios where the superclass/parent class is updated, which could mean having to do multiple writes to different tables, updating the same information, if that information is relevant to multiple tables. There can also be a concern about data inconsistency, as with multiple writes, if any of the writes/updates fail, there needs to be a rollback in the other table updates, which also further adds to server load. Furthermore, all primary keys must be unique across all tables in the hierarchy, which could have an implication on performance as tables need to be aware and check the primary keys of other tables to ensure there is no overlap (can be solved potentially through a global sequence). Lastly, if the type is not known, all tables that are relevant to the hierarchy must be checked, which can require large joins or multiple queries, all which have a negative impact on performance.

Conclusion

Within the Covid-19 Booking System implemented by the team, the team chose to go with class table inheritance with respect to accounts, health care providers and vaccine recipients. Comparing the different inheritance patterns as mentioned above, it becomes apparent that the two best options would be class table inheritance or concrete class inheritance, as vaccine recipient and health care provider attributes have nothing in common, except for the username and password in relation to their accounts. As the implementation of the system was carried out,

it also became apparent that class table inheritance was a better fit for the system at hand, as accounts are attributes to a user of the system, but in terms of relevance and use, are never really needed except during authentication of the user. By utilising class table inheritance, it means that accounts, vaccine recipients and health care providers each have their own separate table, and there is no need for joins when loading information about a user, as the account information is not necessary. This avoids one of the main disadvantages with class table inheritance, which due to the context of the system, makes it an appropriate choice over concrete table inheritance, where storing account details in the same table as both vaccine recipients and health care providers is slightly more inefficient in space during loading into memory, as well as slightly slower writes, as the vaccine recipient and health care provider tables would need to become aware of each other's primary keys, which is avoided using class table inheritance. Overall, the difference between class and concrete table inheritance probably would have been negligible, but to gain slight improvements in performance, the team chose to go with class table inheritance, which looking back, felt like the right choice.

Foreign Key Mapping

Description

Foreign key mapping is the association between tables referenced by a key. Using a relational database, there is no real alternative to using a foreign key, as that is how a relational database store one-to-many relationships. If a many-to-many relationship were to occur, an association table would have to be used, which utilizes two or more foreign keys. The performance implications of using foreign keys are that there is extra processing required for certain updates, as if updates occur to multiple tables through foreign keys, logic needs to be implemented to handle pre-update checks, and rollback scenarios when one or more of the writes to the tables fails. It also is extra work on all write, update and delete operations, as the database has to check for foreign key consistency, which with big data, can be a heavy cost.

Conclusion

The team chose to use foreign key mapping for any one-to-many or many-to-many (association table) scenarios that occurred, as the specifications required a relational database. Due to the relational structure of the database, there were no alternatives to this, other than avoiding the use of foreign keys, by potentially changing the data structure (potentially flattening things out or avoiding one-to-many relationships, which can make the user experience rigid) or by avoiding a relational database such as a document or graph database. While changing to a noSQL database would get rid of foreign keys, and come with some other benefits, it also has its own drawbacks, and depends on the data model and context of the application, which the specifics were outside the scope of this course. The team felt the use of foreign keys was appropriate within the context of the course and the specifications, and avoidance of foreign keys (for minimal performance gains) would likely result in a messy data model that could not fully support the functional requirements of the system.

Authentication and Authorisation

Description

There are four main patterns in relation to authentication and authorisation: authentication enforcer, authorisation enforcer, intercepting validator, and secure pipe. While these can be used in conjunction with one another, for comparison, it can be useful to have a look at each pattern in isolation.

Comparison

- **Authentication enforcer** is where the authentication of users is done through a centralised authenticator, which encapsulates the details of the authentication mechanism. While there are multiple ways to implement the pattern, the general performance implications of the patterns are largely the same. While it can improve security to have a centralised authenticator as it reduces the number of places within the presentation layer that the mechanism is accessed, it can also create a bottleneck in the system, as all authentication requests need to be funnelled into one location, meaning in a system with high volume of users trying to log in, there may potentially be issues.
- **Authorisation enforcer** is the use of a central access controller that performs authorisation based on a user's role. The benefit of using an authorisation enforcer is that it separates out the logic for authentication and authorisation, insulating from each other, but much like the authentication enforcer pattern, it can potentially become a bottleneck for the system, where there are lots of permissions required to access certain functions, and high volume of users trying to do so.
- **Intercepting validator** is a component of the system that intercepts requests and then cleanses and validates the data before the system has to use dynamically loadable validation logic. The main performance advantage of intercepting validators is that while the security validations are centralised, multiple instances can be used to ensure there are no bottlenecks, allowing a standardised process for validations. However, there are still performance implications, as intercepting validators may apply unnecessary filters to incoming requests that do not require them, which adds overhead to the processing. Validators may also be required to read lockable session data, which means that there is the potential for concurrency issues of the system such as low liveness (long wait times) or deadlocks.
- A **secure pipe** is a standardised and generic solution to ensure the privacy and integrity of information being sent over a network. While this provides a secure pipe for increased security of data being sent over a network, which reduces the likelihood of a successful attack from a malicious entity, it does come with performance issues. Encrypting and decrypting requests is an expensive process that can have negative impacts on performance of the system. Delegating cryptographic operations to hardware can mitigate this but does not eliminate the problem.

Conclusion

The team chose to use Spring Security to handle authentication and authorisation. Comparing it to one of the alternatives covered by the course, Apache Shiro, there does not seem to be any performance differences. The main benefits of Spring Security is that it is under active development, it has much more community support and has extensions providing support for both Oauth and kerberos and SAML. The main benefit people claim about Shiro is the simplicity. All the pros and cons mentioned do not really relate to performance. Security is one of the aspects of the system that is really about the level of security required, each layer of security provided will have performance implications, and it is about providing an adequate level of security required to the context of the application. Looking at the Covid-19 Booking System, the team realises that if this were to be a real world application, more effort would have to have been made towards security, as the system is dealing with highly confidential data such as user's medical records, government ids and other potentially confidential information. In this aspect of the system, security cannot be compromised, and the performance impacts that arise need to be managed and mitigated, but cannot take priority over the user's privacy.

Optimistic Offline Locking

Description

Optimistic locking is a concurrency pattern that handles concurrent business transactions by identifying a conflict and rolling back that given transaction. The system conducts optimistic locking by doing a version check on incoming insert/update queries to ensure that the version stored in-memory in the domain layer matches the one currently stored in the database layer. In the case of a version mismatch, the request will then be rejected. Optimistic Locking is used in use cases: Book Timeslot & Record Completed Vaccination.

Comparison with similar patterns

- **Pessimistic Locking:** Pessimistic Locking is a concurrency pattern that requires the system to first acquire a lock before it can conduct any actions. This pattern will decrease the liveness of the system as users would need to wait until the lock becomes available before being able to conduct the business transaction.

Conclusion

With respect to performance, optimistic locking would be the preferable pattern as it has a lower overhead and better liveness compared to pessimistic locking. Pessimistic lock increases the resources used and another layer of complexity as a Lock manager is introduced to the system to manage client access to the locks. It has to be acknowledged however that from a usability perspective, pessimistic locking for use cases where the user is required to do work (i.e. fill in a form) for a business transaction would be more suitable over optimistic locking. In the use cases of Book Timeslot & Record Completed Vaccination where the user has very little work lost, Optimistic locking would be the preferable pattern to best optimise the performance of the system without compromising the user experience.

Unique Key Constraint

Description

Unique Key constraint is a pattern that is used by the system to handle the concurrency issue that arises when two or more users attempt to concurrently create an account with the same username. This is done by defining the “username” key as a unique field. Hence, the database would not allow any data entry queries if the username key already exists in the database.

Comparison with similar patterns

- **Pessimistic Locking**: Using pessimistic for account creation will provide an unpleasant user experience as it would lock access to account creation and only allow one user to create an account at a given time. This results in poor liveness and is not suitable for account creation where it would be necessary to have maximum liveness so users can create accounts on demand.
- **Optimistic Locking**: Optimistic locking and Unique Key Constraint are almost identical apart from the fact that in Unique Key Constraint doesn’t roll back a transaction but simply denies the transaction. Both patterns prevent concurrency issues by handling it at the database level. Optimistic locking in the system is done by version checking and rolling back transactions that have a version mismatch. Unique Key Constraint is done by the database rejecting the query if the unique key constraint of username is violated. Not having the rollback step makes it marginally less computationally taxing.

Conclusion

Unique Key Constraint is a simple pattern to prevent duplicate entries for certain fields that need to be unique which is also suitable for handling certain concurrency issues. Due to the simplicity of the pattern, it has a low computational cost and resource overhead. When compared to another concurrency pattern that is relatively lightweight, optimistic locking, it performs marginally better as it has fewer steps to be executed.

Unit of Work

Description

Unit of Work is essentially a software ledger that keeps track of changes on domain objects of the system and only commits the changes to the database once at the end of the multi-step business transaction. The types of changes it keeps track of are creation of new objects (new), objects with changes to their variables (dirty), objects that have not been changed since they were last read from the database (clean) and objects that have been deleted (deleted). In the system, unit of work is used to handle the use case of Health Care Providers adding one or more timeslots. This allows the system to minimise the number transactions the client commits to the database by combining multiple queries into one transaction.

Comparison with similar patterns

The alternative to Unit of Work would be not using the pattern. In the use case of Health Care Providers adding timeslots, not using a Unit of Work to pipeline the requests would increase the total time it takes to execute every individual request to add a timeslot on the database. This increased runtime for the business transaction leads to a reduced throughput. The benefit of not using a Unit of Work however is the reduced usage of resources to complete a business transaction and thus would improved performance on the computational side of the business transaction. For simpler use cases, for example: Creating a Vaccine Recipient Account, using a Unit of Work to create one account would not increase the throughput of the system as there is only one query to be executed. Moreover, it would increase the computational overhead and resulting in an overall decrease in performance.

Conclusion

Utilising the unit of work pattern for a complex business transaction improves the system's performance by pipelining multiple synchronous requests and significantly reducing the throughput of the entire business transaction. However, using a unit of work unnecessarily will lead to system overheads as it introduces unnecessary resource usage for simple business transactions. Therefore a unit of work should only be used for use cases where multiple database queries are expected to be executed to ensure the throughput increase is more significant than the computational overhead it introduces.

Identity Field

Description

The identity field pattern involves storing the primary key of a database table in the corresponding domain object for each row. Reading data from the database can be easily done and data can be displayed with ease through JSX files. However, the identifiers (primary key) stored in the in-memory objects using the identity field pattern provide a method of correctly identifying and updating (writing back) specific rows to the database.

In the team's implementation of the covid19 vaccine booking system, there were several use cases that required writing data back into the database after an initial read. In other words, there were use cases that required updating specific rows in the database. For example, a vaccine recipient in the system can book a timeslot which involves updating the chosen timeslot's row in the database with a vaccine recipient id and 'Booked' status. Further, there are use cases for healthcare providers in which they can record vaccination appointments (timeslots) as completed and they can edit the details of any timeslot. All of these use cases require updating the timeslot table in the database.

However, each of these update queries require a reference from the database table for the row that needs to be updated. In the database, the timeslot table has a column called 'id' which is also the primary key of the table, hence unique. When the timeslots are read by the system, they are also stored in-memory as Timeslot objects. These objects have several attributes such as vaccineType, dateTime, duration, etc. The id corresponding to each timeslot row in the

database is also stored in the respective object. Hence, the identity field pattern is used here so that when data is already read by the client and needs to be updated, the corresponding id acts as a reference for the row in the database.

Comparison with similar patterns

- **Not using Identity Field Pattern**: The alternative to using the identity field pattern is simply not using it at all. This would mean that the id field would not be stored as an attribute in the in-memory objects. If the id field was not stored as an attribute in the Timeslot objects, updating an individual row would require reading the data from the database again based on another unique attribute. This would reduce the performance of the system significantly as the data would have to be read twice before updating. However, using the identify field pattern can also have a negative impact on overall coupling of the system. Introduction of a database table primary key as an attribute in the in-memory objects would imply higher coupling between the domain and the database layer.
- **Using meaningful and compound keys**: There are several methods of choosing and generating keys for the identify field pattern. The team chose to have meaningless, simple, table-unique and auto-generated keys. For example, the id in the timeslot table was of type ‘serial’ which meant that it auto-generates unique integers starting from 1 and has no meaning in relation to domain objects. The other options include generating meaningful and compound keys. This would add another layer of complexity to the system which could hinder performance. Moreover, since auto-generating keys is done at the database layer, the main system is not burdened with more compound calculations reducing the complexity of the system.

Conclusion

Using the Identity field pattern can instigate several performance improvements and makes it easier to update individual rows in the database. Even though it increased the complexity and overall coupling of the system, it also made sense to use it for this system as it provided a way for the domain layer to communicate with the database layer and vise-versa. Moreover, it reduced additional reads from the database, hence increasing performance.

Lazy Loading

Description

Lazy loading is specifically used to reduce the number of reads made to the database by only reading what is needed at a given point in time. Dummy data is loaded on the client side until the actual data from the database is required. This increases performance significantly as reads from the database are minimalized to those instances when data needs to be loaded/shown on the client side.

From the list of all available lazy loading patterns, the team chose to implement the ghost pattern. In this pattern, each attribute of an object is assigned to null upon initialization. When any field is requested/loaded, all the fields are loaded at the same time. In the covid19 vaccine booking system implementation, when timeslots are returned after the user searches for an available timeslot, the healthCareProvider field is set to null as this information is not displayed to the client yet upon the first request. When a vaccine recipient wants to view a particular timeslot, all the fields of the corresponding timeslot's health care provider are loaded at the same time and displayed on the client side.

If the user does not view any particular timeslot from the list of timeslots, then the respective Health Care Provider data is never loaded/read from the database hence reducing processing time and increasing fetching speeds. This in turn reduces the performance overheads of unnecessary reads.

Comparison with similar patterns

- **Not using lazy loading pattern**: If the team decided to not use the lazy loading pattern, it could have affected the performance of the system significantly. This is because, for each timeslot loaded into the system, the healthcare provider table in the database would also be read to load the healthcare provider information corresponding to each timeslot. This would mean that unnecessary reads would take place consistently throughout the life of the session and would have reduced performance.
- **Using the lazy initialization pattern**: The lazy initialization pattern is similar to the ghost pattern. It involves setting all the fields in the object to null at the time of initialization. However, only the field that needs to be displayed on the client side or the field that is requested is loaded from the database at the time of the request. All the other fields remain null until they are requested subsequently. This was not an ideal approach for the team's implementation as when the user views any particular timeslot in detail, all the healthcare provider fields are shown on the client side at the same time. If the team implemented lazy initialization, the system would make multiple calls to the database to get information for the same row one after another. This would be data intensive and would increase processing times, hence reducing the overall performance of the application.
- **Using the value holder and virtual proxy patterns**: These patterns involve postponing object creation to save memory usage. However, the healthCareProvider object contains only a few fields and did not require implementation of this complex pattern as it did not take up much space in the first place.

Conclusion

The ghost pattern seemed the most appropriate to implement for the use case of viewing timeslot details. Unnecessarily loading data from the database could have negatively impacted performance of the system. Even though implementation of a lazy loading pattern increased complexity of the system, it was necessary for this use case.

Pessimistic Offline Locking

Description

The purpose of pessimistic locking is to avoid conflicts occuring by locking the row in the database being edited throughout the life of the update. For example, in the context of the system, when a Health Care Provider wants to edit any of their timeslots in the system, they would need to update the details of the timeslot on the client side. This can be a cumbersome process as the user would be entering details in input boxes rather than just clicking on buttons and navigating.

We used exclusive write locks for the described use case. This means that when a timeslot is being edited, the session holder would first need to acquire a write lock on the chosen row in the timeslot table. This would result in two scenarios:

1. If no other user has already acquired a lock on the chosen timeslot, then the current user would be able to update the details of the timeslot.
2. If another user has already acquired a lock on the chosen timeslot, then the current user would be redirected to an error page.

This approach certainly reduces the overall performance of the system as another user would not be able to edit the timeslot at the same time. Moreover, the use of a locking mechanism adds an extra layer of complexity to the system and could impact the performance of the system in the case that many users attempt to edit the given timeslot at the same time.

Comparison with similar patterns

- **Optimistic Locking**: If the team decided to use optimistic locking for the use case of editing timeslots, multiple users would be able to update the same timeslot at any given time. This would result in conflicts occuring at the time of committing the transaction, resulting in many users being redirected to an error page. This scenario is not ideal for the use case in question as the user would need to refill the details to update the chosen timeslot. It would however improve the performance of the system as the session holder would not waste time attempting to acquire a lock and would also be able to edit the timeslot at any time they want without having to wait for a lock to be available.
- **Exclusive Read Locks**: If the team decided to implement exclusive read locks instead of exclusive write locks, it would mean that multiple users would not be able to view or edit timeslots at the same time. This would however, reduce performance as the users would not even be able to view timeslots at the same time. Another issue with exclusive read locks is inconsistent reads. However, this would only happen if a user would stay on the editing timeslots page (the list view of all timeslots) for the whole duration of another user editing any of those timeslots. It is unlikely that this would ever happen and since it has a major impact on the overall performance of the system, the team decided to use exclusive write locks.

Conclusion

Using pessimistic locking had several benefits as described above, however, it reduced the liveness of the system. This is because no two users would be able to edit the same timeslot at any given time. Even though using pessimistic locking minutely reduced the performance of the system, it was necessary for the chosen use case as it was a better choice for user experience. Moreover, the team understood that the use case of editing timeslots would not often occur concurrently as they are unique to each healthcare organisation.