Performance Evaluation Report

Team0 - Marketplace

Lai Wei Hong 1226091

Sean Wong 885710

Andrew Liu 1084190

Xi Zhao 1184353

Table of Contents

Introduction

The purpose of this document is to evaluate the performance of the Marketplace System developed by Team0. In this document, we will be

1. Outlining the different aspects that we will be measuring our performance against.
2. Discuss about the design patterns implemented by the team and its performance implications
3. Discuss about several design principles / patterns that we believe will have a significant impact on performance

How is performance measured?

To start discussing improving performance, we need to identify areas at which performance can be improved. To do so, we need metrics to be able to gauge the performance of our system. The metrics considered are listed below.

Average response time

Average response time is the average amount of time it takes for the marketplace backend server to the client's request. And It is measured in milliseconds, the timer starts from the second a client sends out a request and stops when the marketplace backend server sends back a response. And we will conduct many such response time tests, and then average the data we get to get the average response time. The average response time can help us to measure the performance of our marketplace server, and it is an essential metric to track and monitor as it can give us a clear understanding of marketplace server performance.

Error rates

Error rates refers to the percentage of requests issues we meet in relation to the total number of requests which can help us to measure the performance of our marketplace server, because it is an important metric to track our server, and we should avoid the peak situation, because it may mean that our system has a serious failure and does not perform well. On the contrary, if our system can keep it a relatively low value, it shows that our server has good performance.

Memory leaks

Memory leaks refers to a type of resource leak which may happen in our system when our marketplace server incorrectly manages memory allocations. And it may lead to our marketplace system performance degradation because the system will consume increasingly memory resources due to the memory leaks, and even crashes. Besides, Memory leaks tests can help us to measure the performance of our system which is also an important metric to track and monitor our server which can confirm whether our system has memory leaks and how many memory leaks do we have. Memory leaks tests are based on the theory that some API processes will start to consume more and more memory without releasing it under heavy load. Finally, the API becomes completely unusable and needs to be restarted. Therefore, regularly sample the memory consumption of API processes and put it on a line chart. If the line of the line chart is relatively straight, it means that there is no memory leak or very little memory leak, on the contrary, there are many memory leaks which will seriously affect the performance of the system.

Which design pattern will affect performance

Identity Maps

The main purpose of an identity map is to ensure that the object used is consistent throughout a business transaction. To achieve this, the instance of an instantiated object is stored in a HashMap until the end of the business transaction.

This approach has an unintended consequence of caching the object. In our implementation, Identity Maps are embedded within the Unit of Work instance. Whenever a domain object calls for a read operation from the Unit of Work with a supplied key, the Unit of Work first checks if the Identity Map has an entry for the key. If so, the object is retrieved from the Identity Maps. Since the Identity Maps store objects in-memory, it is much cheaper than database calls, which aids in increasing the response times for the backend server, thus improving server performance.

JSON Web Tokens

JSON web tokens (JWTs) provide a URL-safe method of transferring claims between the server and the client. JWTs were used for the authentication of users as well as for storing user-specific data such as user ID, user roles and group id of users (if they were sellers) of said users.

The primary advantage of using JWTs over other methods of authentication is that it reduces the need to perform requests to a database in order to obtain user or session data. As user information can be encrypted and stored in tokens themselves, this removes the need to contact a database for user information which can be simply obtained by decrypting the token. As performing requests to the database takes time, by reducing the need to contact the database, latency is reduced. Additionally, JWT tokens move the session state storage onto the client side instead of the server- side. As session data would need to be stored either in memory or on the database on the server-side, by moving it onto the client-side, we decrease the amount of memory used by the server/storage space used in the database, thus further increasing the performance of the system.

A disadvantage of JWTs is security. JWT tokens can be stolen as well as improperly shared with other users and this can lead to misuse. Additionally, invalidating JWTs is hard. Invalid JWT tokens would have to be kept track of on the server usually by storing them in memory until they expire due to time. This could result in large amounts of memory being used on the if many user tokens are being invalidated e.g., many users logging out of the system. The server would also have to clear stored invalid JWTs and thus this could decrease performance when handling large amounts of invalidated tokens.

Pessimistic Offline Lock

To handle the concurrency issues, adding locks was necessary and will undoubtedly reduce performance for the relevant operations. Within our application, we have opted to use a Pessimistic Offline Lock which locks a record for exclusive use for the entirety of a business transaction.

With our implementation of the Pessimistic Offline Lock, business transactions are prevented from executing whenever another thread already owns the lock. This means that no unnecessary requests and rollbacks would have to occur, unlike an optimistic lock where a transaction will continue and revert later if the versions are inconsistent. Since an online marketplace usually involves financial transactions, pessimistic locks were chosen to handle concurrency issues for their safety and integrity despite the slight performance trade-off compared to an optimistic lock. Normal optimistic lock operations will still cost less due to not having to acquire a lock for every row but will be more expensive whenever a conflict is encountered. So performance will still greatly depend on the level of contention in the system. The system will handle concurrency well in high levels of contention as transactions are locked early, which ensures no rollbacks are needed while maintaining the integrity of the data. Given that transactions in our system are short, the minor performance benefits of an optimistic lock would be negligible resulting in our choice of using a pessimistic lock.

Using an in-memory Concurrent HashMap to store locking data also removes the need to communicate to the database before transactions and eliminates the need for a locking table in the database, reducing database memory usage. This potentially improves average response time as it doesn't have to initially access the database, increasing the performance of any locking transactions.

Lazy Loading

We have implemented the lazy loading pattern so that only relevant data is loaded when necessary.

With our implementation of lazy loading with lazy initialisation, data is only loaded whenever necessary and when null. If the information already exists in memory from a previous initialisation, the transaction will return the already existing data without having to read from the database. This results in various performance improvements. Reducing the number of reads to the database will lower web pages' latency and load time while also cutting down the amount of bandwidth used by the transactions. This will also lower the load on the database, increasing overall performance. Finally, the memory requirement for the system is reduced because only the required data is retrieved rather than all attributes related to the object.

A possible disadvantage of the lazy loading pattern is possible ripple loading. This occurs when more database calls are executed than what's required. While cases of ripple loading don't necessarily happen in our system, it is still something we considered. For instance, when we get a list of orders for a customer, all data comes from a single request rather than querying over order items one by one. This ensures that efficiency is maintained and no redundant requests are made to the database.

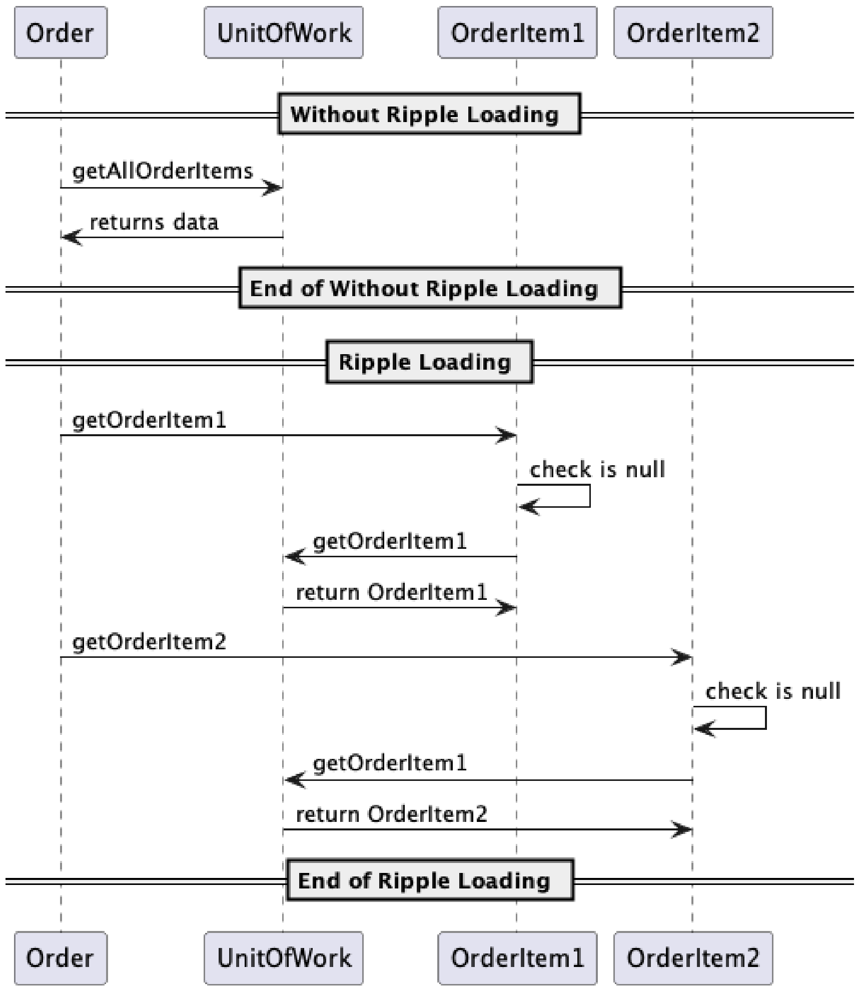


Figure 1 : Operations with and without ripple loading

Unit of Work

The Unit of Work pattern was implemented to group multiple operations into a singular transaction and ensure that the transaction is only executed to the database when all operations pass.

With our implementation of Unit of Work, the changes to the database are only committed if no failures are encountered during the business transaction. The pattern reduces the number of database calls by failing early while also removing the need to undo erroneous changes made to the database. By reducing the round trips to the database and only committing when valid, average response time is improved, decreasing page load times and overall system performance.

How could our system be improved?

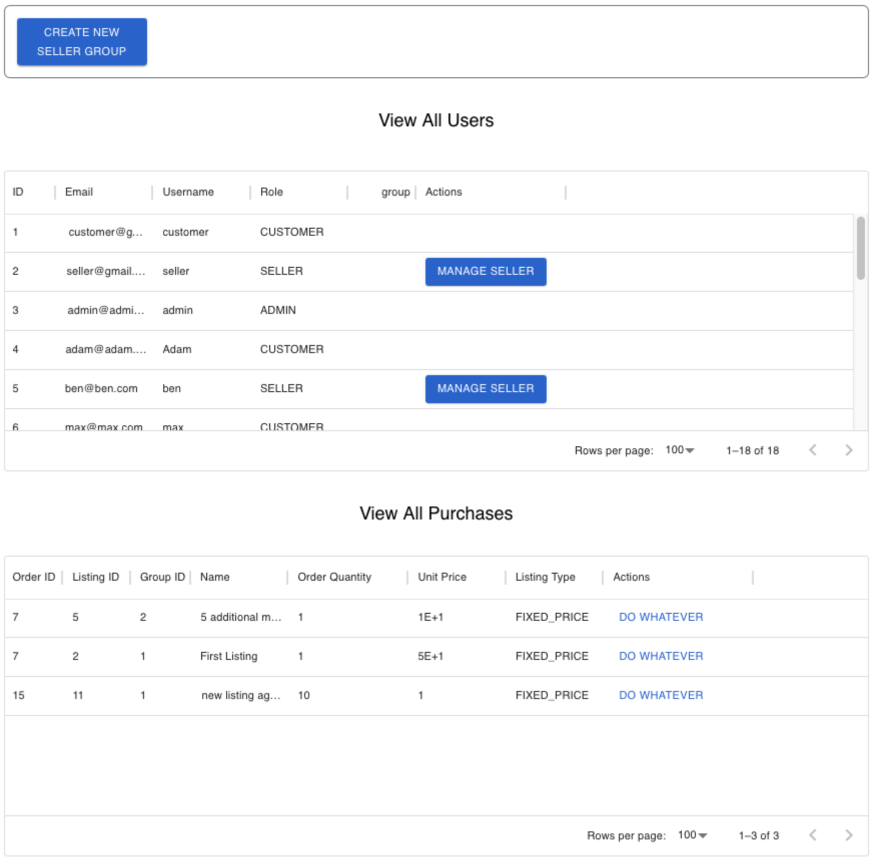
There are several design principles and patterns that were not explored during the development of this project. They are listed below :-

1. Pipelining
2. Caching
3. Remote Facade

The team also believes that implementing auto-scaling capabilities (load balancing) can improve the performance of the server. However, this approach escalates the system setup to a multi-server setup, which would require a reconsideration of the system architecture implemented so far. Therefore, discussions on this topic will be centred around the theoretical aspects of it.

Pipelining

In our project, one of our main gripes is the way the dashboard is loaded. Different components load individually as they request for resources from different API endpoints.



View All Users

ID	Email	Username	Role	group	Actions
1	customer@g...	customer	CUSTOMER		
2	seller@gmail....	seller	SELLER		MANAGE SELLER
3	admin@admi...	admin	ADMIN		
4	adam@adam....	Adam	CUSTOMER		
5	ben@ben.com	ben	SELLER		MANAGE SELLER
6	max@max.com	max	CUSTOMER		

Rows per page: 100 1–18 of 18

View All Purchases

Order ID	Listing ID	Group ID	Name	Order Quantity	Unit Price	Listing Type	Actions
7	5	2	5 additional m...	1	1E+1	FIXED_PRICE	DO WHATEVER
7	2	1	First Listing	1	5E+1	FIXED_PRICE	DO WHATEVER
15	11	1	new listing ag...	10	1	FIXED_PRICE	DO WHATEVER

Rows per page: 100 1–3 of 3

Figure 2 : Admin Dashboard

Using the admin dashboard as an example, the endpoints in the backend for /getAllUsers and /getAllPurchases are called separately. This can be improved with the use of pipelining. Instead of opening a new HTTP connection to the backend for every request, we could establish a HTTP persistent connection by setting the header “Connection: keep alive”. Now, these two components can call to the same API endpoint, /loadAdminDashboard with two different requests. The backend can process the two requests in parallel and return the data requested back to the front-end simultaneously.

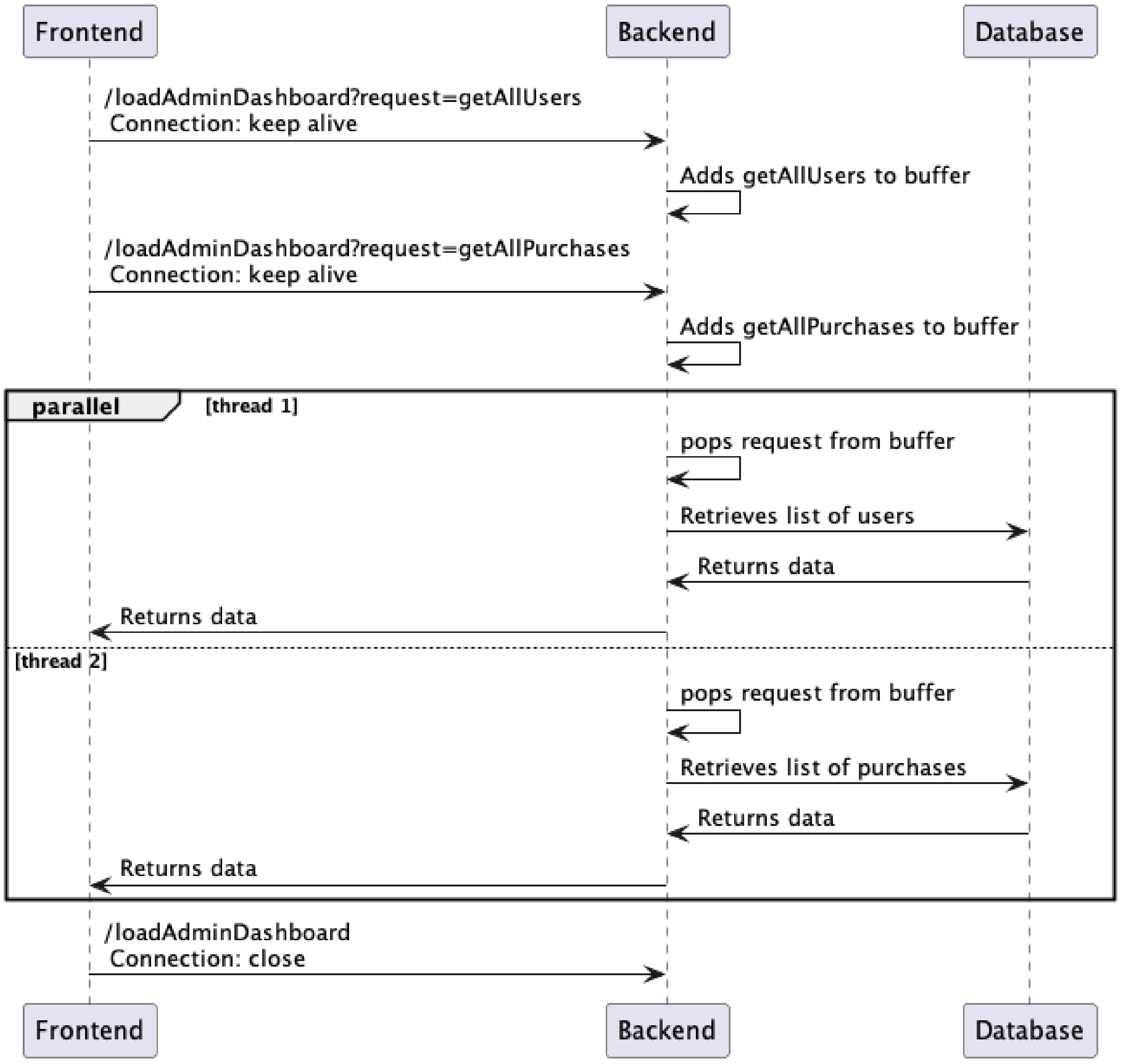


Figure 3 : Sequence diagram for cases when pipelining is implemented

Pros and Cons of pipelining

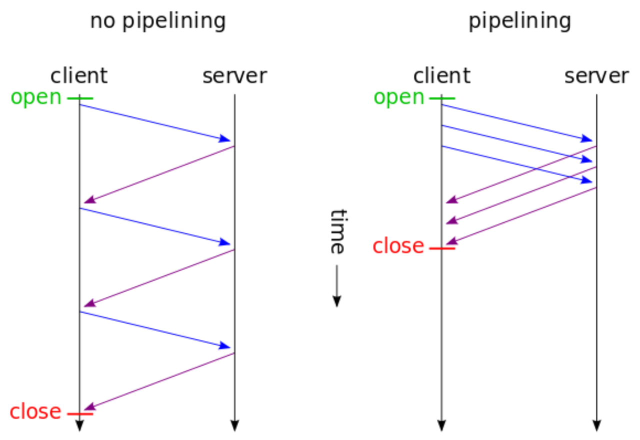


Figure 4 : Difference with and without pipelining

One of the strengths of pipelining is the ability to process multiple requests at the same time. With reference to figure ?, when pipelining is not implemented, we follow the “send a request, wait for a response” model. The issue with this model is that the backend is constrained to only be running on a single thread, which hampers the response time of the system. With pipelining implemented, a single business transaction that consists of multiple requests can be processed in parallel. This helps reduce the response time of the system as more computing resources can be allocated.

With the improved performance, we should also be aware of the downsides of pipelining. Intermediate proxy servers that service the network traffic between the frontend and the backend could present a problem to pipelining. If the proxy servers do not support pipelining, then the pipelined request would not work as expected, despite the backend and frontend supporting pipelining.

Another con that is important, but not necessarily relevant to our system is the increased implementation complexity. Typically in pipelining, the order at which the requests are sent is important. This leads to the need of implementing mechanisms to keep track of the order of the requests in the frontend, so that the content is loaded correctly. In our system, the order at which data is returned plays a less significant role as in all cases, we either load everything or none at all.

Caching

Caching was performed on a per-business transaction basis as a side effect of the identity map implementation in Unit Of Work. While this allowed for frequently-accessed data to be quickly accessed during a business transaction, the same data accessed from a different business transaction would need to be reloaded from the database. This is where we believe a more robust caching strategy could come into play.

When to use caching

For caching to be effective, it needs to fulfil two conditions, temporal locality and spatial locality. Temporal locality refers to the tendency of a program to use the same data item over and over again throughout the program's lifespan. A good example of this in our project is the Checkout functionality. A popular listing could be frequently read and checked out by different buyers. If we store the listing data in the cache, instead of reading it from the database every time it is needed, we can reduce the response time.

Spatial locality, on the other hand, refers to the tendency of a program to use data items near a frequently accessed data item. Using our project as an example, this could be when a number of sellers from the same seller group looks to modify orders. Since the Order table only has the reference to the order items in the order, preprocessing order items and loading it into the cache could lead to a performance gain in the system.

Pros and Cons

One of the main attractions of caches are the lower wait times for access to resources. Since the data is stored in memory, it can be accessed quickly, instead of relying on the round-trip database reads.

However, since cache only stores a snapshot of the data and does not represent the state of the data(the master data is stored in the database), long-running cache runs the risk of referencing stale data in a multi-server setup. A possible remediation is to have the server broadcast database changes upon each commit. While this enables the cache to have an up-to-date copy of a frequently accessed data item, it also floods the network with non-business related packets. This puts a strain in the server's ability to handle requests.

Besides that, the data stored in the cache needs to have spatial and temporal locality. Without any of these two conditions, there is no benefit to implementing a cache. If a data access stream has no temporal locality, then the data item stored in the cache will not be referenced at all. It is worse if the data access stream lacks both temporal and spatial locality, as we will be loading and storing associated data items for a data item presumed to be frequently accessed. This leads to unnecessary computations and could lead to slower response times from the server.

How can we implement this

Caching can be incorporated in the read process to act as an intermediary before the expensive reads from the database. In cases where validation is done, such as uniqueness check and listing validity, the cache will not be checked as the freshness of the data in caches can be difficult to validate.

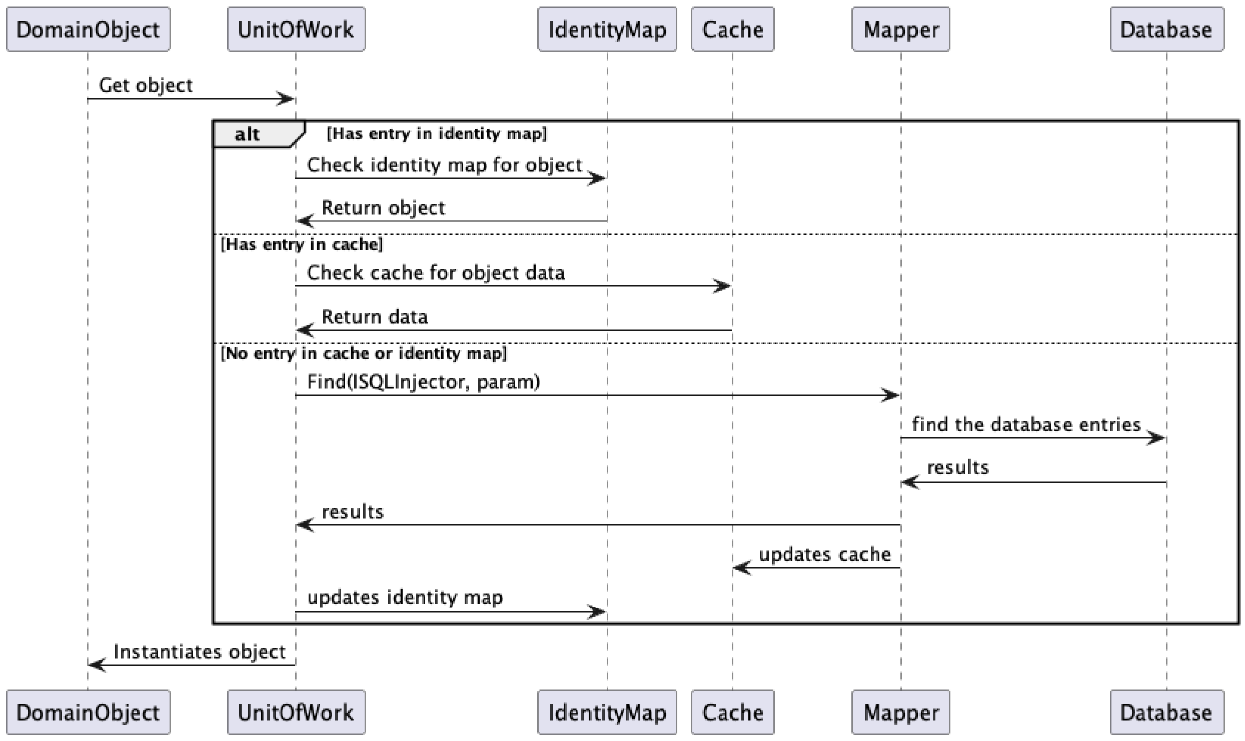


Figure 5 : Sequence diagram for read operations with cache

Remote Facade

Remote facade is a design pattern intended for minimising the number of remote calls to our system. As an example, our project utilises two servers, one is a React server serving front-end pages and the other is a Java EE server providing backend services. Interactions between the two servers over the web is much more expensive than if the interactions are done over a single address space. Therefore, there is a need to minimise the number of calls made by the React server to improve the responsiveness of the webpages. Remote facade offers a way to achieve that.

Remote facade comes with an important caveat. With the remote facade, we are giving up the fine-grained control of domain object methods for coarse-grained interfaces by aggregating a selected number of method calls. This presents two issues to the developer. The first being the difficulty in identifying the suitable level of granularity for the coarse grained interfaces. If not careful, a god object that is coupled to every class in the system could be created. The second issue is that the remote facade increases the difficulty of programming. As an example, instead of interacting with a single domain object that holds the desired behaviour, remote facade leads to the need for the developer to handle a group of objects that may not be relevant to the business transaction. This could result in more unnecessary memory being consumed by irrelevant objects, hampering the performance of the system.

How can we implement this

In our current implementation, the servlet takes on the task of the remote facade and DTO. The servlet API endpoints call the methods from different domain objects and aggregate the result into a JSON result string. We have also built in JSON deserialisation and serialisation in the servlets. This is a sub-optimal implementation as this led to bloated servlets, which increases code duplication and logic flaws.

While the remote facade design pattern would not necessarily lead to a gain in performance as its job has already been accomplished in the currently implemented servlet, it can help reduce the bloat in the servlets and improve the readability of the code. Extending on the pipelining example earlier, the sequence diagram below outlines the relationship between the servlet and the facade.

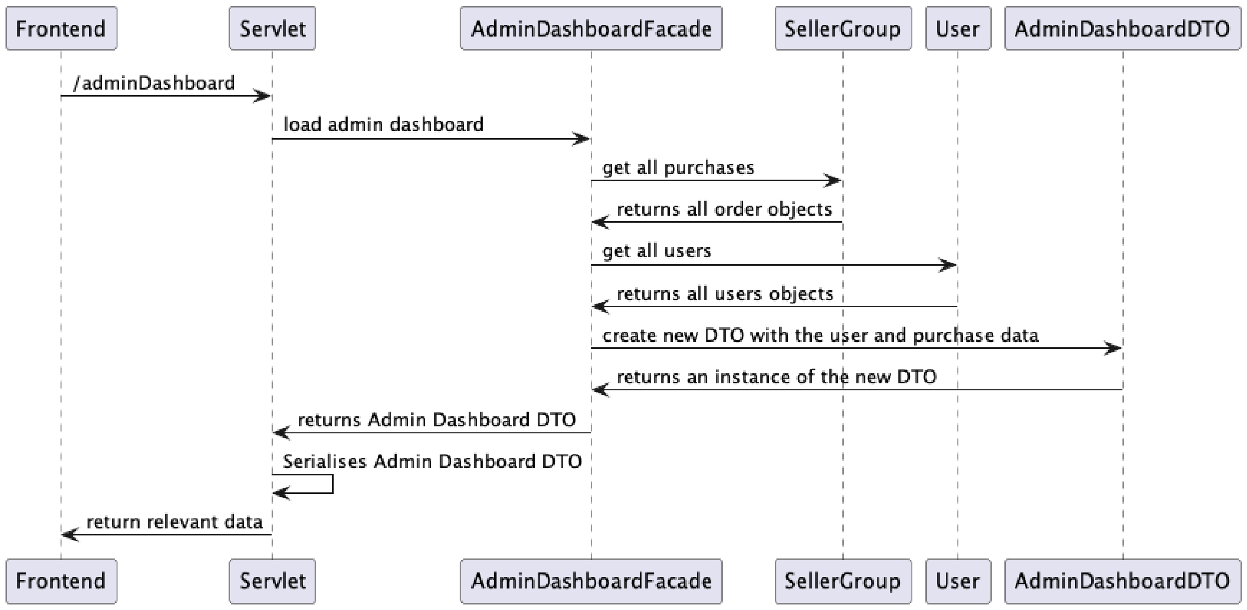

Frontend
Servlet
AdminDashboardFacade
SellerGroup
User
AdminDashboardDTO
/adminDashboard
load admin dashboard
get all purchases
returns all order objects
get all users
returns all users objects
create new DTO with the user and purchase data
returns an instance of the new DTO
returns Admin Dashboard DTO
Serialises Admin Dashboard DTO
return relevant data

Figure 6 : Sequence diagram for remote facade interactions

With this implementation, we can keep our servlets lean as it would simply call the facade to call the relevant methods for the business transaction, instead of having to make calls to both the Seller Group and User domain object.

Auto-scaling capabilities

Auto-scaling capabilities (Load balancing) refers to the process that the system automatically adjusts its resources according to the current load of the system. With the Auto-scaling capabilities, our marketplace is brought online or offline automatically and the workload can then be distributed to the server pool depending on the current load, depending on the auto-scaling strategy we use for our system.

Strategy for auto-scaling

To implement the Auto-scaling capability, we also need a strategy to define how scaling happens, and there are three strategies that are usually used, including scheduled, reactive, and predictive. With different strategies, the market system could be able to achieve the purpose of automatic scaling according to this strategy.

The system with a scheduled auto-scaling strategy could be able to start or shut down the server according to the scheduled plan. With reactive auto-scaling strategy, the servers can start and shut down as the load changes, but the effectiveness of the strategy requires us to track the load of the current server closely. On the other hand, with the predictive auto-scaling strategy, the system could make correct adjustments by using historical data or identifying the usage patterns of resources, but this significantly increases the implementation complexity. A well-structured auto-expansion system and an appropriate auto-expansion strategy can improve the flexibility and resilience of services and the ability to handle unexpected loads, leading to lower response times and more efficient use of system resources, such as memory. Thereby, improving the overall performance of the system.

Conclusion

In this report, we have considered the performance implications of different design patterns that the team has implemented in our MarketPlace system. We have defined the performance metrics used to measure different aspects of the system where performance is measured. Finally, we have also considered several additional performant design patterns / principles that were not considered in this project.