Software Architecture Design Report

Submission 4 SAD

NüRoom
SWEN90007 SM2 2021 Project

In charge:

Name	Student ID	UoM Username	GitHub Username	Email
Guo Xiang	1227317	XIAGUO2	moneynull	xiaguo2@student.unimelb.edu.au
Basrewan Irgio	1086150	IBASREWAN	irgiob	ibasrewan@student.unimelb.edu.au
Bai Zhongyi	1130055	zhongyib	Marvin3Benz	zhongyib@student.unimelb.edu.au
Hernán Romano	1025543	hromanocuro	hromanoc	hromanocuro@student.unimelb.edu.au

Release Tag: SWEN90007_2022_Part4_Nuroom

Revision History

Date	Version	Description	Author
23/10/2022	01.00-D1	Initial document	Hernán Romano
24/10/2022	01.00-D2	Restructure of document	Team
24/10/2022	01.00-D3	Adding Optimistic Offline Lock reflections	Hernán Romano
24/10/2022	01.00-D4	Added Testing Results	Zhongyi Bai
25/10/2022	01.00-D5	Add reflections for unit of work	Irgio Basrewan
25/10/2022	01.00-D6	Adding Pessimistic Offline Lock reflections	Hernán Romano
26/10/2022	01.00-D7	Added Lazy Load reflection	Zhongyi Bai
28/10/2022	01.00-D8	Adding reflection of DTO	Xiang Guo
29/10/2022	01.00-D9	Adding reflection of Caching principle	Xiang Guo
29/10/2022	01.00-D10	Added reflections for design principles	Irgio Basrewan
31/10/2022	01.00-D11	Adding Bell's Principle reflection	Hernán Romano
01/11/2022	01.00-D12	Added reflections for identity map	Irgio Basrewan
03/11/2022	01.00-D13	Added Cache & Pipeline patterns	Zhongyi Bai
03/11/2022	01.00-D14	Add reflection of Bell's Principle	Xiang Guo
04/11/2022	01.00	Version 1 & Create Release tag	Team

Contents

1. Introduction

1.1 Proposal

This document specifies the SWEN90007 project's detailed use cases, actors, updated domain model, Class Diagram and the description of each Design Pattern implemented in the system.

1.2 Target Users

This document is mainly intended for SWEN90007 students and the teaching team.

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
UML	Unified Modelling Language
Hotelier Group	A set of Hoteliers under a common hotel business interest
Amenity	A desirable or useful feature or facility of a hotel or room

2. Actors

Actor	Description
Customer	A person who makes use of the Hotel Booking System and acquires services
Administrator	A person who manages the Hotel Booking System
Hotelier	A person who manages hotel(s)

3. Design Patterns Discussion

3.1 Unit of Work

3.1.1Examples and Justification

To reflect on the performance of our system, we decided to focus on the following analysis:

- To compare the used design pattern with a different design pattern
- To compare the performance of the system **before** and **after** the implementation of the **Unit of Work** design pattern

The performance evaluation in this section was done over the following Business Transaction:

Table 3.1: Examples - Business Transactions evaluated

Actor	Business Transaction
Customer	Book Hotel
	Modify Bookings
Hotelier	Create Room
	Modify Hotel

After executing experiments systematically using Jmeter, we arrive at the following reflections on the performance of the system:

- The unit of work pattern has many benefits, but its primary one in terms of performance is its **efficiency in making minimal changes to a large group of objects**
- **Writing changes to a database is an expensive operation**, without the unit of work pattern individual changes to data would be committed separately throughout the business transaction rather than just all together at the end.
- Making these commits together in a single transaction leads to much better performance and efficiency. While there is slightly increased overhead with registering and marking objects with the unit of work, these actions take a negligible amount of resources in comparison to committing data to the database.
- Data from tests done in Jmeter does indeed support the conclusion that the unit of work overall did decrease response times, which makes sense as the Unit of Work should commit all the data at once, which is **much more efficient**.

3.1.2Results

To simulate the scenarios in the real world, we wrote a python script (uploaded under /part4/testing/input_csv_data/) to export several CSV files containing random dates, hotel IDs, bookings, and others, which can be used in Jmeter to run the tests.

Table 3.2: Unit of Work - Book Hotel - Throughput

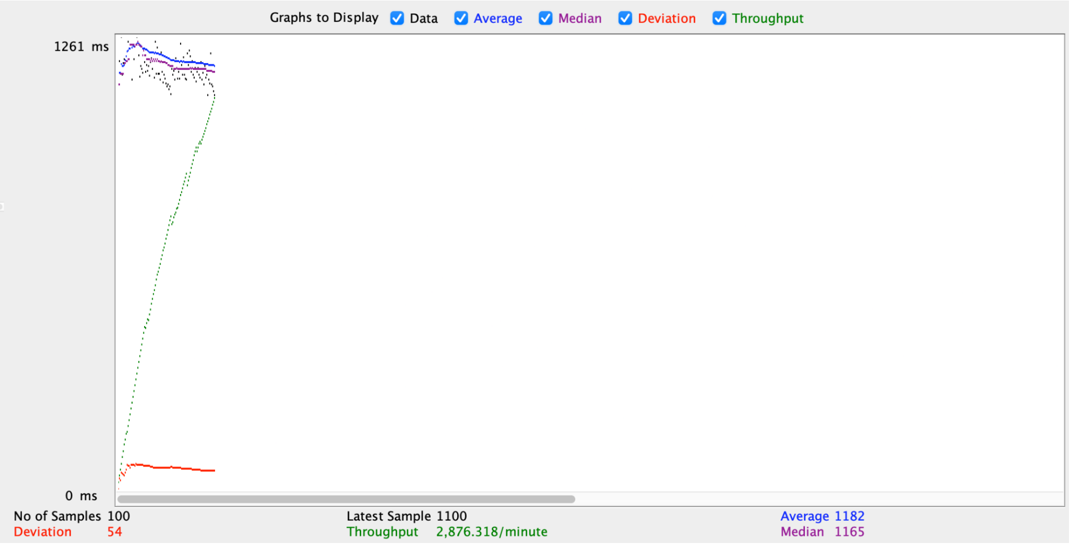
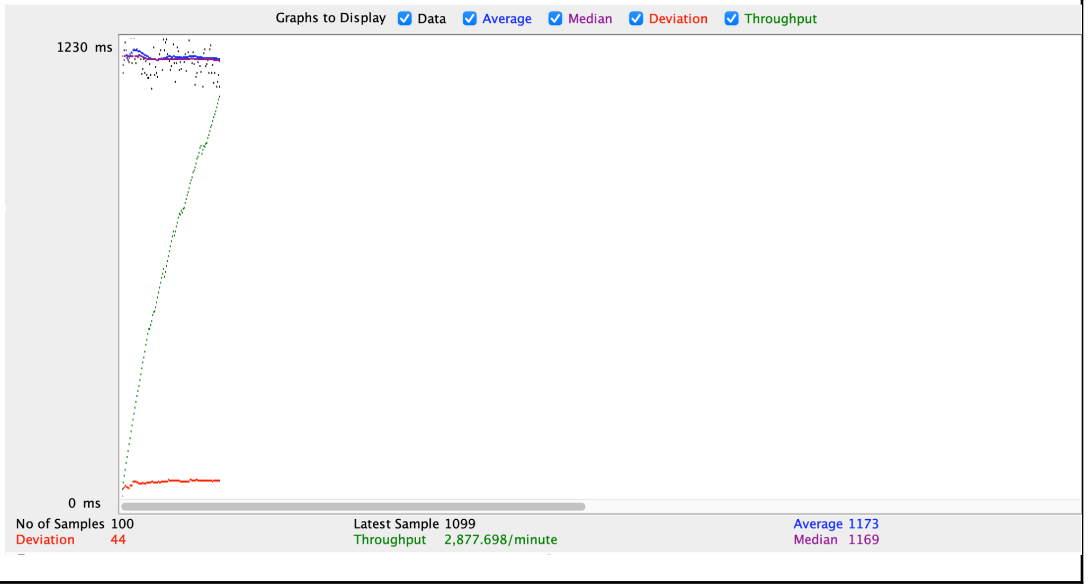
Throughput: Book Hotel	
Before “Unit of Work” implementation	**After “Unit of Work” implementation**
 Graphs to Display ☑ Data ☑ Average ☑ Median ☑ Deviation ☑ Throughput 1261 ms 0 ms No of Samples 100 Deviation 54 Latest Sample 1100 Throughput 2,876.318/minute Average 1182 Median 1165	 Graphs to Display ☑ Data ☑ Average ☑ Median ☑ Deviation ☑ Throughput 1230 ms 0 ms No of Samples 100 Deviation 44 Latest Sample 1099 Throughput 2,877.698/minute Average 1173 Median 1169

Table 3.3: Unit of Work - Book Hotel - Response Time

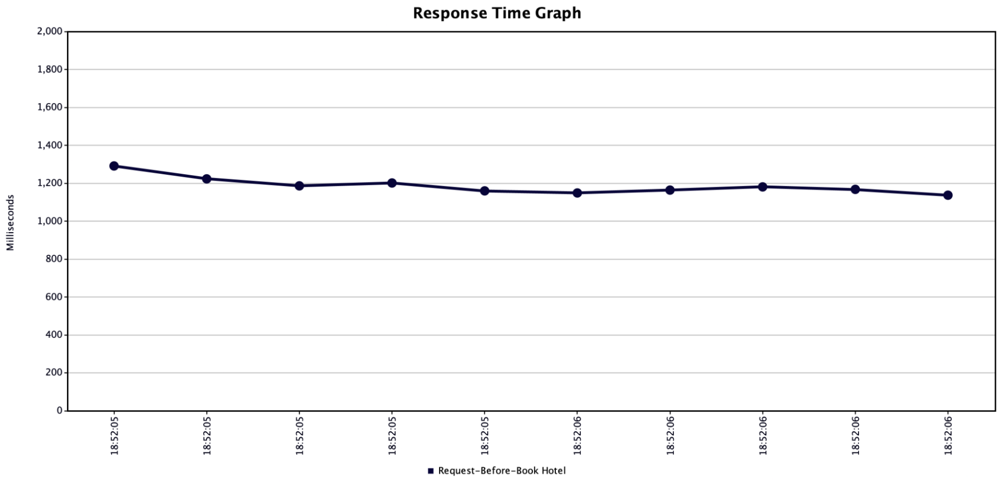
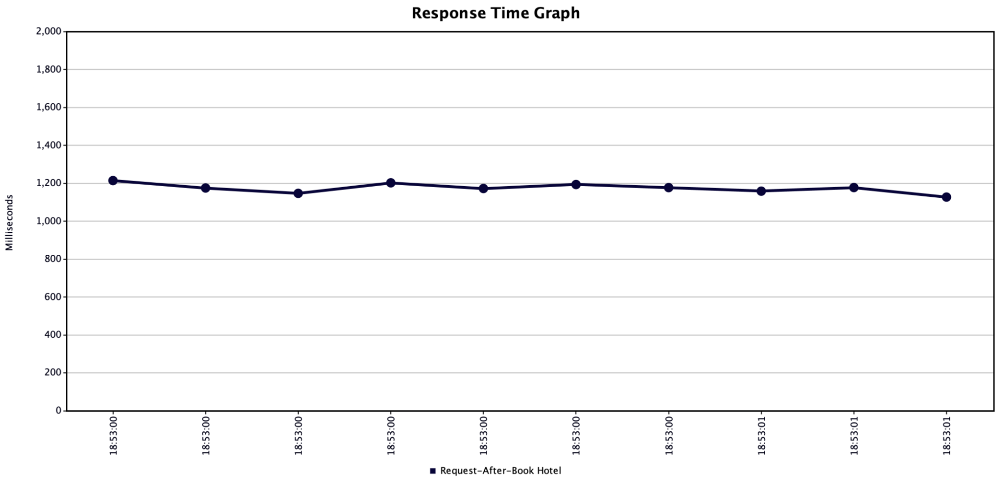
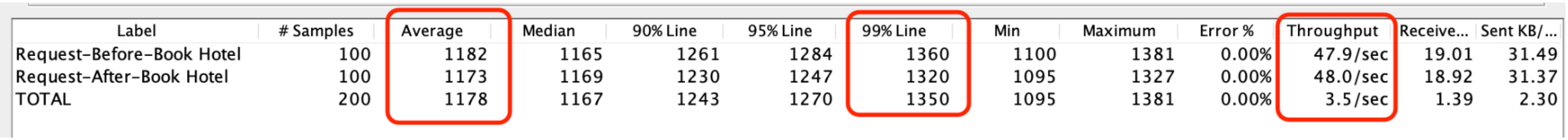
Response Time: Book Hotel	
Before “Unit of Work” implementation	After “Unit of Work” implementation
	

Figure 3.1: Statistics - Book Hotel



Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput	Receive...	Sent KB/...
Request-Before-Book Hotel	100	1182	1165	1261	1284	1360	1100	1381	0.00%	47.9/sec	19.01	31.49
Request-After-Book Hotel	100	1173	1169	1230	1247	1320	1095	1327	0.00%	48.0/sec	18.92	31.37
TOTAL	200	1178	1167	1243	1270	1350	1095	1381	0.00%	3.5/sec	1.39	2.30

Table 3.4: Unit of Work - Modify Bookings - Throughput

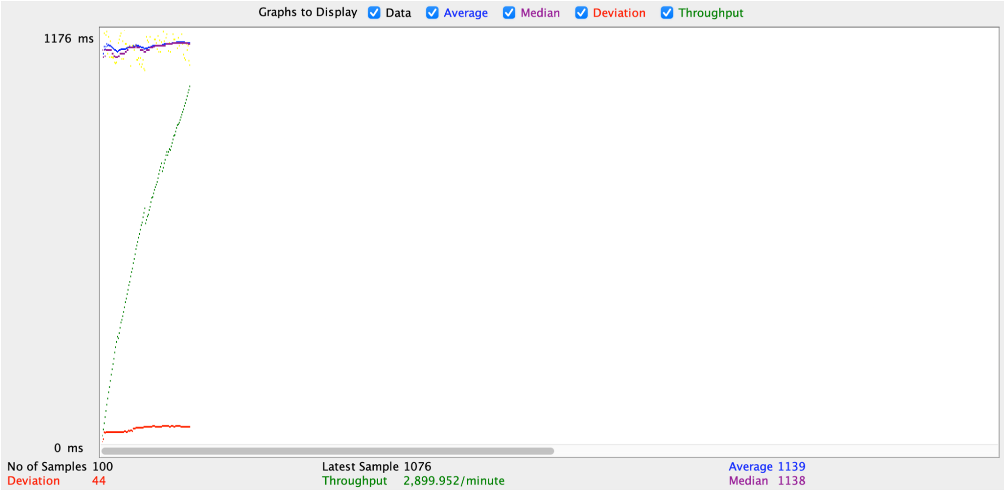
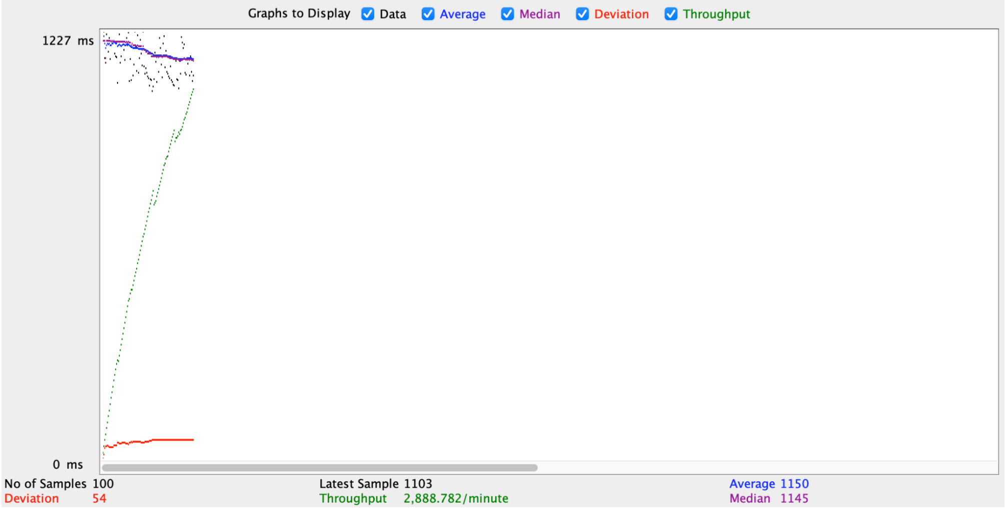
Throughput: Modify Bookings	
Before “Unit of Work” implementation	**After “Unit of Work” implementation**
 Graphs to Display ☑ Data ☑ Average ☑ Median ☑ Deviation ☑ Throughput No of Samples 100 Deviation 44 Latest Sample 1076 Throughput 2,899.952/minute Average 1139 Median 1138	 Graphs to Display ☑ Data ☑ Average ☑ Median ☑ Deviation ☑ Throughput No of Samples 100 Deviation 54 Latest Sample 1103 Throughput 2,888.782/minute Average 1150 Median 1145

Table 3.5: Unit of Work - Modify Bookings - Response Time

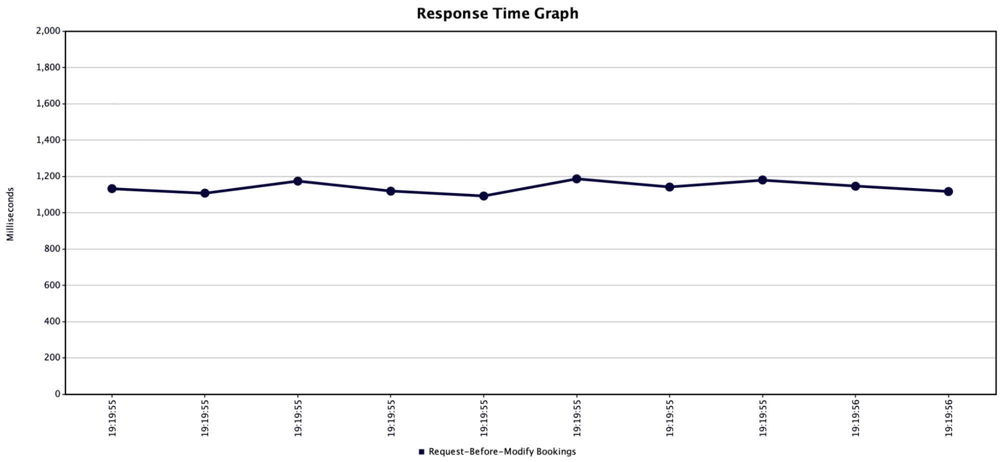
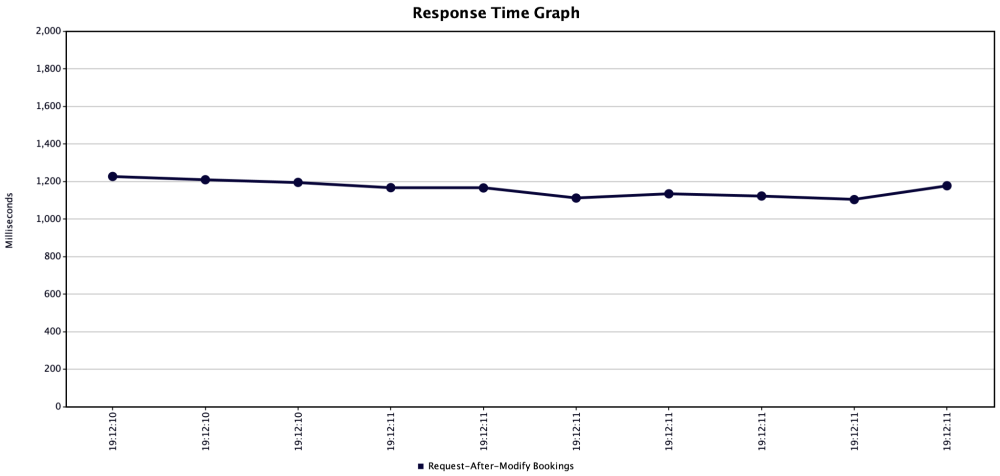
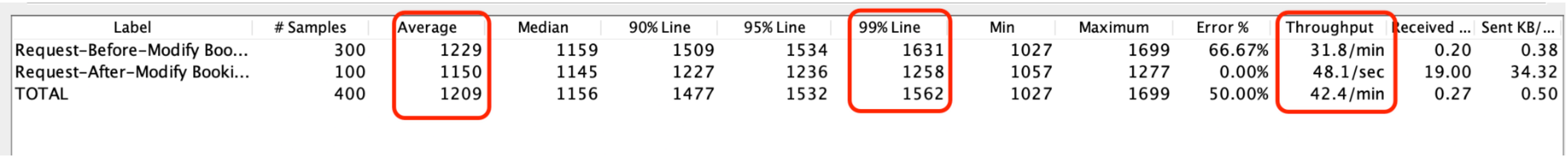
Response Time: Modify Bookings	
Before "Unit of Work" implementation	**After "Unit of Work" implementation**
	

Figure 3.2: Statistics - Modify Bookings



Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput	Received ...	Sent KB/...
Request-Before-Modify Boo...	300	1229	1159	1509	1534	1631	1027	1699	66.67%	31.8/min	0.20	0.38
Request-After-Modify Booki...	100	1150	1145	1227	1236	1258	1057	1277	0.00%	48.1/sec	19.00	34.32
TOTAL	400	1209	1156	1477	1532	1562	1027	1699	50.00%	42.4/min	0.27	0.50

Table 3.6: Unit of Work - Create Room - Throughput

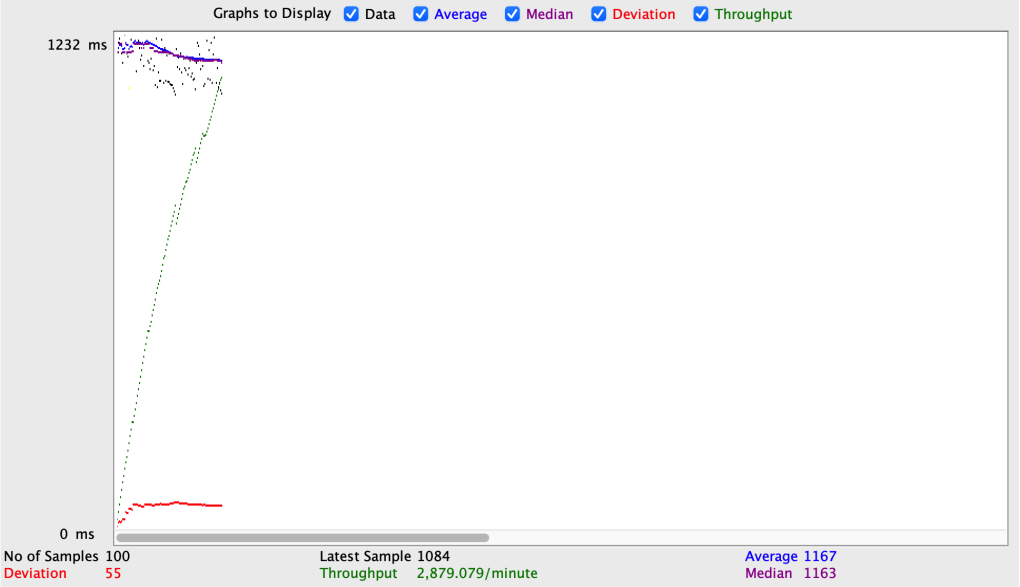
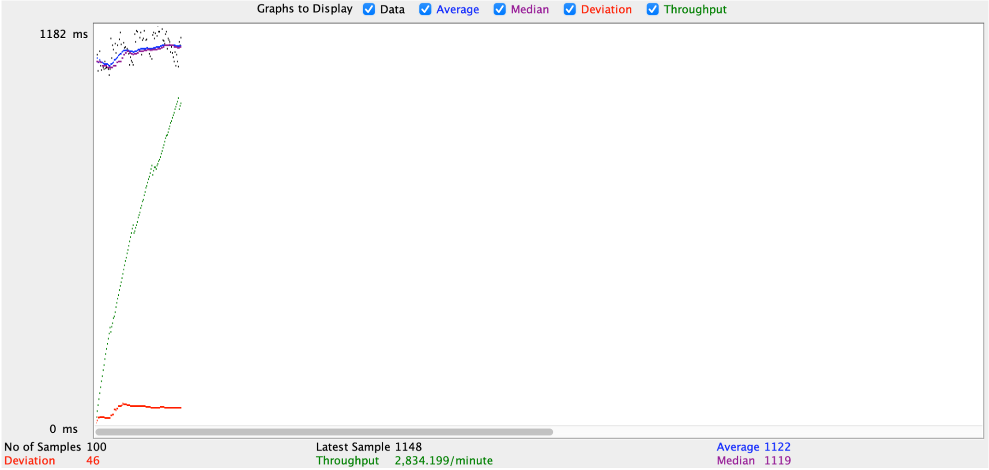
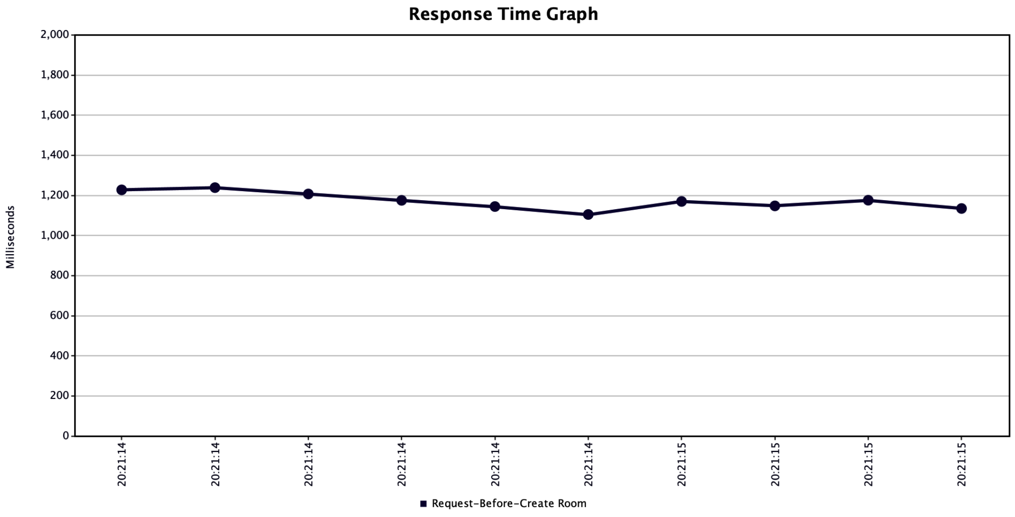
Throughput: Create Room	
Before “Unit of Work” implementation	**After “Unit of Work” implementation**
 Graphs to Display ☑ Data ☑ Average ☑ Median ☑ Deviation ☑ Throughput No of Samples 100 Deviation 55 Latest Sample 1084 Throughput 2,879.079/minute Average 1167 Median 1163	 Graphs to Display ☑ Data ☑ Average ☑ Median ☑ Deviation ☑ Throughput No of Samples 100 Deviation 46 Latest Sample 1148 Throughput 2,834.199/minute Average 1122 Median 1119

Table 3.7: Unit of Work - Create Room - Response Time

Response Time: Create Room

Before “Unit of Work” implementation



After “Unit of Work” implementation

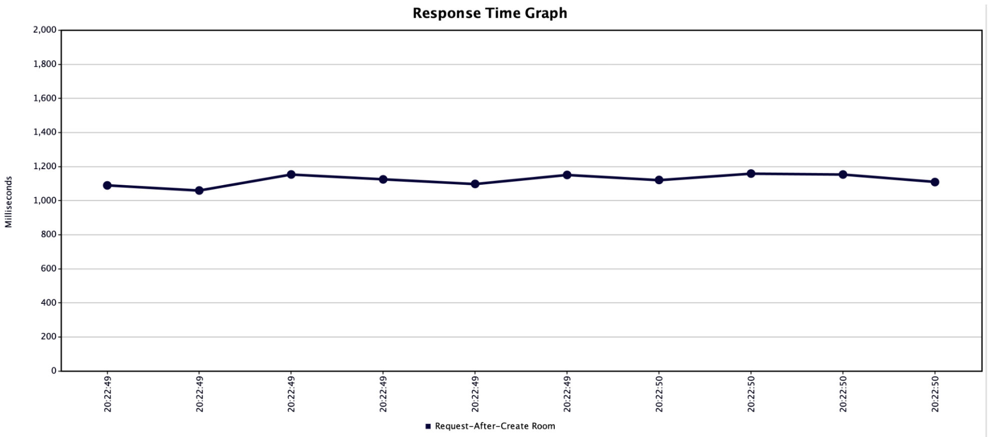
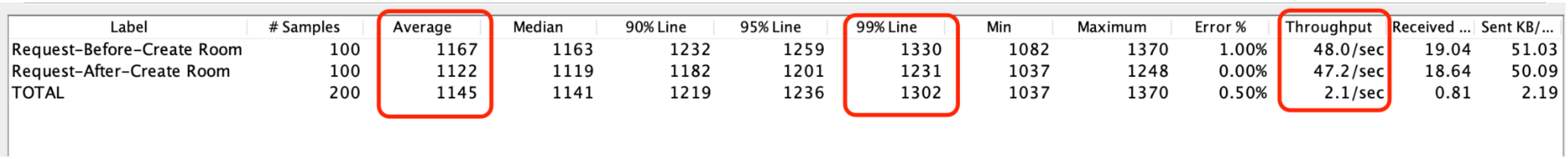


Figure 3.3: Statistics - Create Room



Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput	Received ...	Sent KB/...
Request-Before-Create Room	100	1167	1163	1232	1259	1330	1082	1370	1.00%	48.0/sec	19.04	51.03
Request-After-Create Room	100	1122	1119	1182	1201	1231	1037	1248	0.00%	47.2/sec	18.64	50.09
TOTAL	200	1145	1141	1219	1236	1302	1037	1370	0.50%	2.1/sec	0.81	2.19

Table 3.8: Unit of Work - Modify Hotel - Throughput

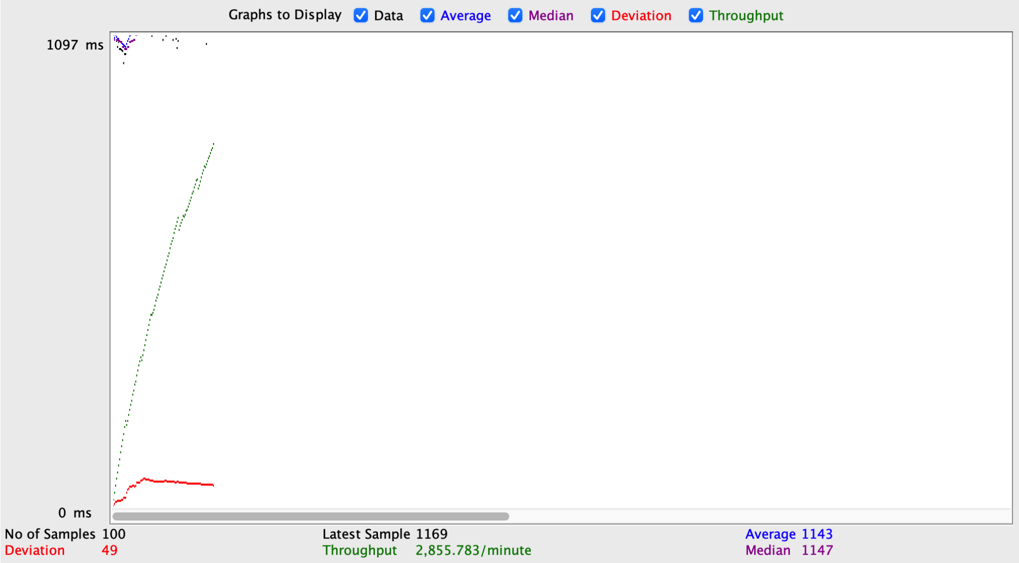
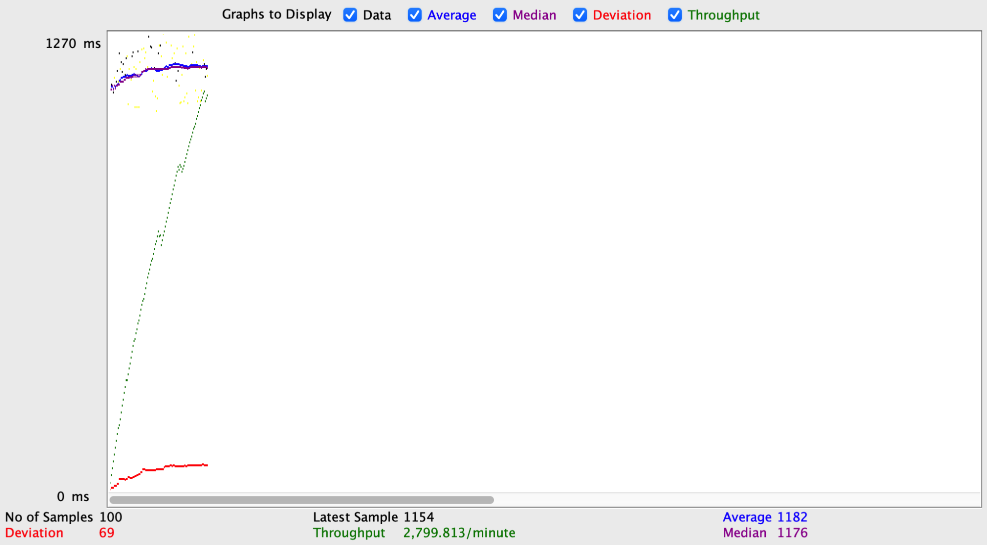
Throughput: Modify Hotel	
Before “Unit of Work” implementation	**After “Unit of Work” implementation**
 Graphs to Display ☑ Data ☑ Average ☑ Median ☑ Deviation ☑ Throughput No of Samples 100 Deviation 49 Latest Sample 1169 Throughput 2,855.783/minute Average 1143 Median 1147	 Graphs to Display ☑ Data ☑ Average ☑ Median ☑ Deviation ☑ Throughput No of Samples 100 Deviation 69 Latest Sample 1154 Throughput 2,799.813/minute Average 1182 Median 1176

Table 3.9: Unit of Work - Modify Hotel - Response Time

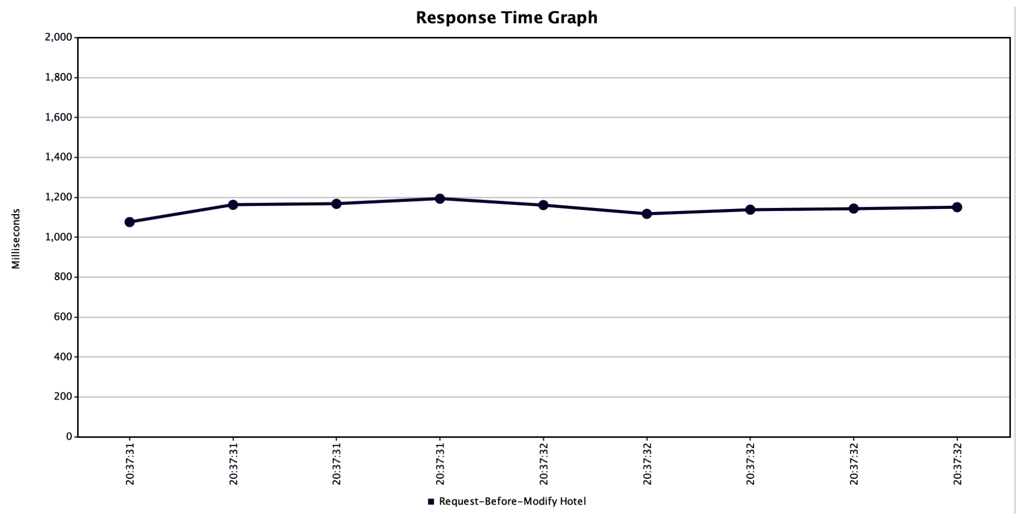
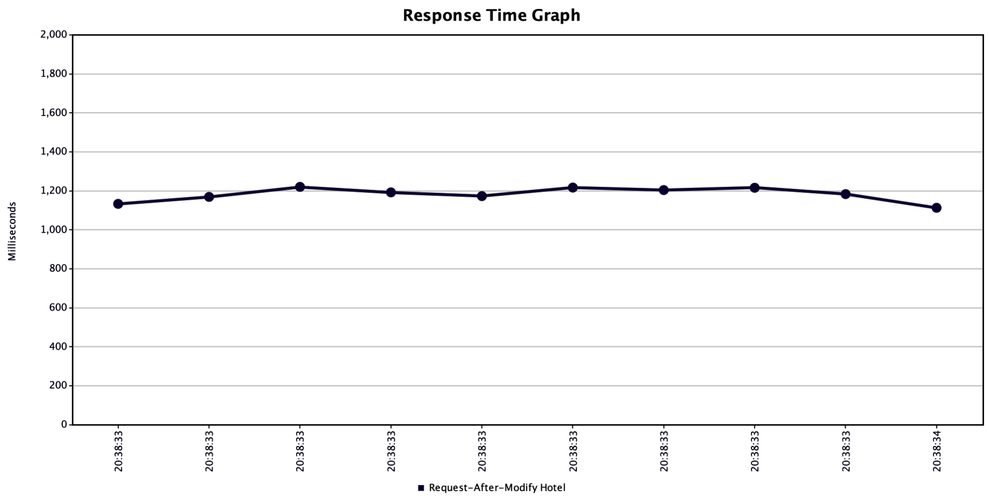
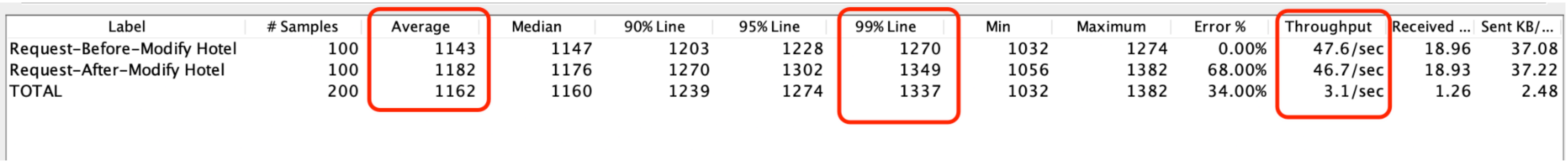
<table>
<tr><th colspan="2">Response Time: Modify Hotel</th></tr>
<tr><th>Before “Unit of Work” implementation</th><th>After “Unit of Work” implementation</th></tr>
<tr><td> </td><td> </td></tr>
</table>

Figure 3.4: Statistics - Modify Hotel



Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput	Received ...	Sent KB/...
Request–Before–Modify Hotel	100	1143	1147	1203	1228	1270	1032	1274	0.00%	47.6/sec	18.96	37.08
Request–After–Modify Hotel	100	1182	1176	1270	1302	1349	1056	1382	68.00%	46.7/sec	18.93	37.22
TOTAL	200	1162	1160	1239	1274	1337	1032	1382	34.00%	3.1/sec	1.26	2.48

3.2 Lazy Load

3.2.1Examples and Justification

To reflect on the performance of our system, we decided to focus on the following analysis:

- To compare the Lazy Load with a CDN and Pagination
- To compare the performance of the system **before** and **after** the implementation of the **Lazy Load** design pattern

The performance evaluation in this section was done over the following Business Transaction:

Table 3.10: Examples - Business Transactions evaluated

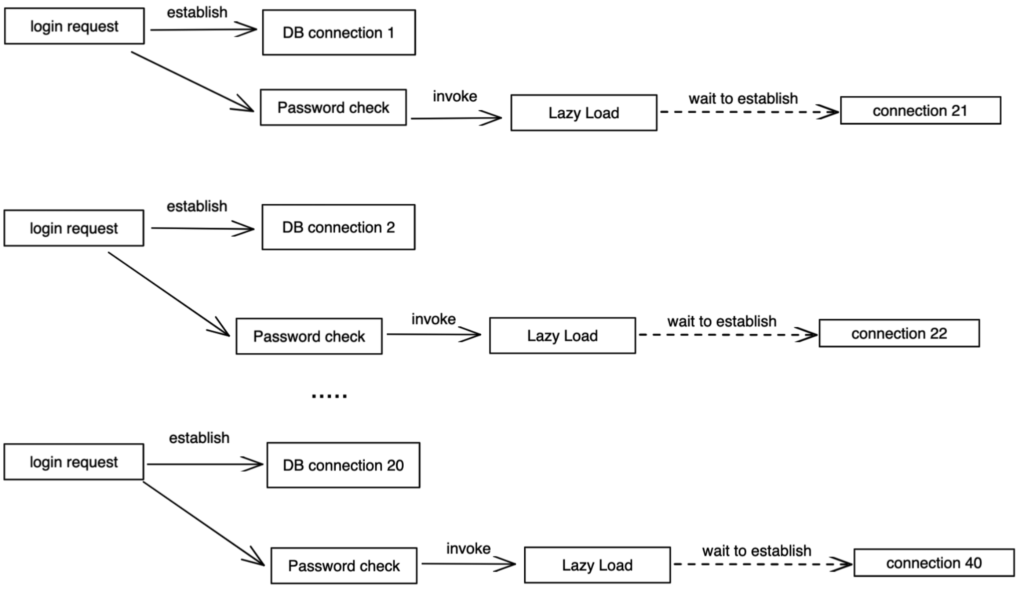
Actor	Business Transaction
Customer	Login
Hotelier	Create Hotel

After executing experiments systematically using Jmeter, we arrive at the following reflections on the performance of the system:

- **The implementation of Lazy Load decreases system's performance.** As shown in Figure 3.7 and Figure 3.8, **the system's throughput decreased by around 7% and the system's response time decreased by 10-16%.** For both business transactions, the version of implementing Lazy Load always has worse performance in response time and throughput. After reviewing our code, we found Lazy Load in the backend is not very helpful for our system, because we are using RESTful API; additionally, the backend and frontend are separate. Each request will establish more Database (DB) connections to get the required data. However, if we use JSP for development, we believe Lazy Load is beneficial to improve system performance since it uses fewer DB connections and saves time.

- **Using the Lazy Load is easier to cause deadlock in the system, which will significantly affect the system's usability and performance when a larger amount of users log in to the system concurrently.** The reason is shown in Figure 3.5, when a user sends a login request to the backend, the login method needs to establish a DB connection to be prepared for further processing. Then, the system has to check the user's password, at that time, it invokes Lazy Load to get the user's information. Lazy Load needs to create a new connection, whereas, due to the Heroku connection limit (20), if the system wants to establish a new connection, there must be at least a connection has been released. At that time, each login request is waiting for others to release the connection first, however, a connection cannot be released until the login process is complete, which caused the deadlock.

Figure 3.5: Explanation of deadlock



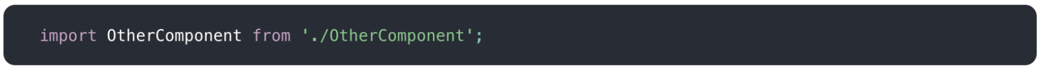
- **Using Content Delivery Networks (CDN) could reduce the system's response time by placing the server closer to users, and they can access the content faster.** We could use the CDN provider's service to distribute some static content files all around the world. If a customer uses our system in Australia, the web hosting in Australia will offer the content. If a customer is in Canada, the CDN will use servers in Canada to provide content. By using CDN, a user can gain a better user experience. Instead, if we only deploy our server in Australia, a user in Canada may wait 2 or 3 times longer than a user in Australia to load the website.

- **Using Pagination could reduce response time.** When more and more customers book hotels in our application, it is predictable that the number of bookings will increase. Say there are 1,000,000 booking records, if the admin wants to view all bookings, it may take over 1 min to load all booking information, which is a very bad user experience. We could use pagination to only load 100 records every time, which can significantly reduce load time and response time.

- **Using Lazy Load in the front end.** In the front end, we could also apply lazy load to reduce initial loading time. As we are using React for front-end development, it naturally provides **React.lazy** to improve application performance. As Figure 3.6 shows, a component will only load the expensive component when it is first rendered. If this component is never used, it will never be loaded, which saves time.

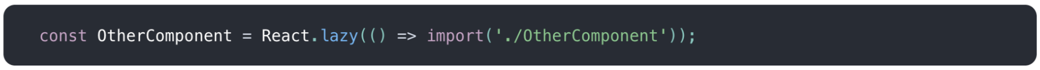
Figure 3.6 React.lazy

Before:



```
import OtherComponent from './OtherComponent';
```

After:



```
const OtherComponent = React.lazy(() => import('./OtherComponent'));
```

3.2.2Results

Table 3.11: Lazy Load - Login - Throughput

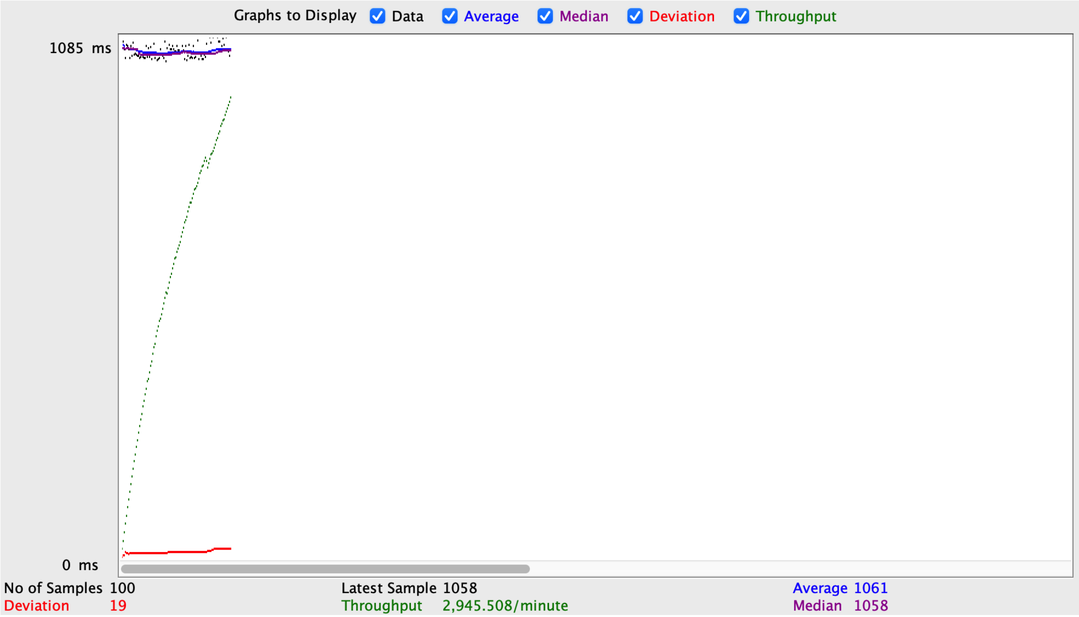
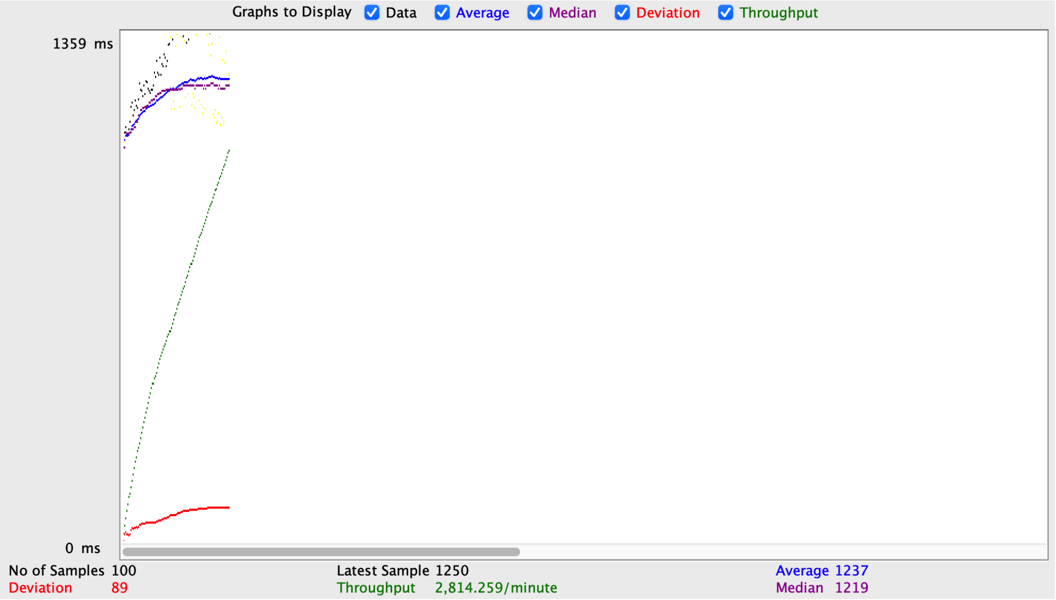
Throughput: Login	
Before “Lazy Load” implementation	After “Lazy Load” implementation
 Graphs to Display ☑ Data ☑ Average ☑ Median ☑ Deviation ☑ Throughput 1085 ms 0 ms No of Samples 100 Deviation 19 Latest Sample 1058 Throughput 2,945.508/minute Average 1061 Median 1058	 Graphs to Display ☑ Data ☑ Average ☑ Median ☑ Deviation ☑ Throughput 1359 ms 0 ms No of Samples 100 Deviation 89 Latest Sample 1250 Throughput 2,814.259/minute Average 1237 Median 1219

Table 3.12: Lazy Load - Login - Response Time

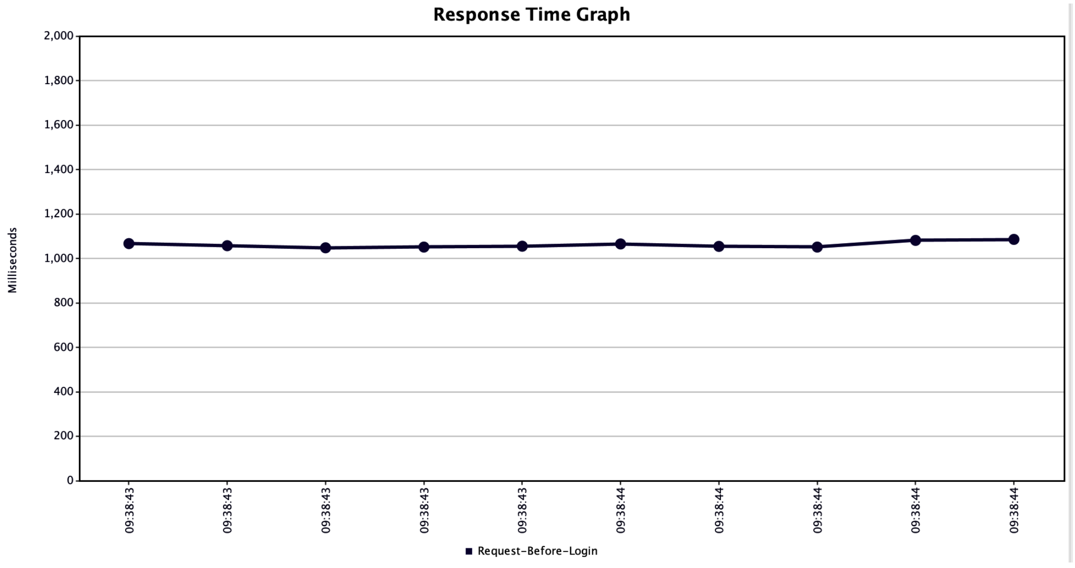
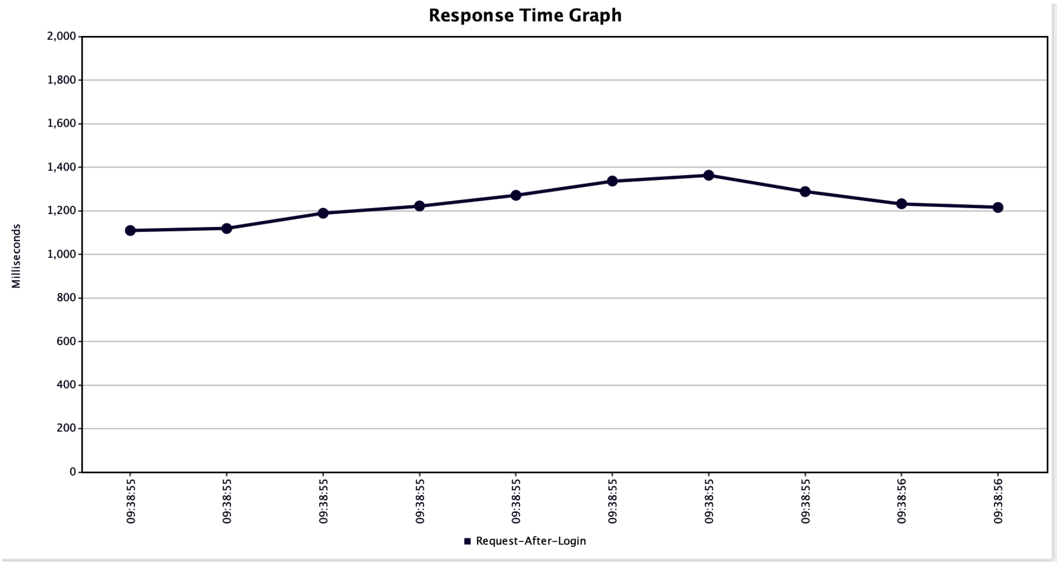
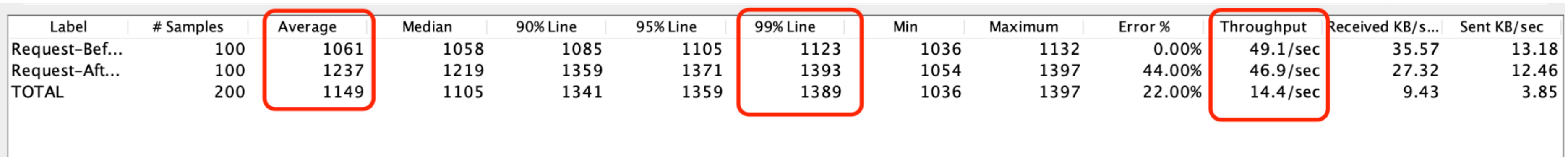
Response Time: Login	
Before “Lazy Load” implementation	After “Lazy Load” implementation
	

Figure 3.7: Statistics - Login



Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput	Received KB/s...	Sent KB/sec
Request-Bef...	100	1061	1058	1085	1105	1123	1036	1132	0.00%	49.1/sec	35.57	13.18
Request-Aft...	100	1237	1219	1359	1371	1393	1054	1397	44.00%	46.9/sec	27.32	12.46
TOTAL	200	1149	1105	1341	1359	1389	1036	1397	22.00%	14.4/sec	9.43	3.85

Table 3.13: Lazy Load - Create Hotel - Throughput

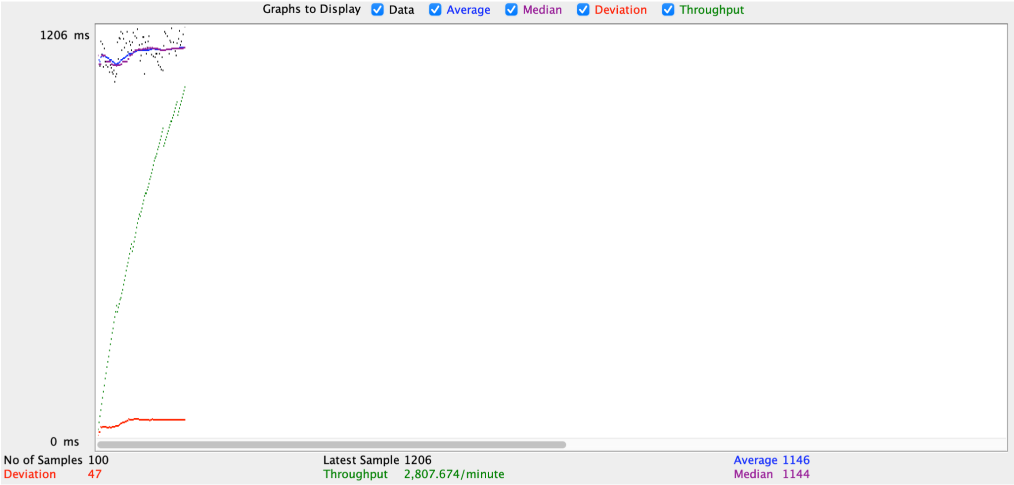
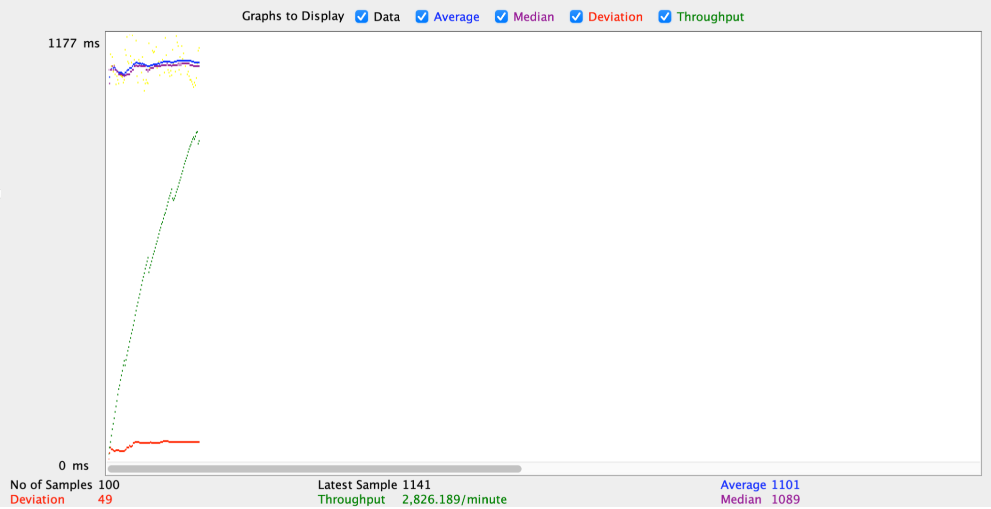
Throughput: Create Hotel	
Before “Lazy Load” implementation	**After “Lazy Load” implementation**
 Graphs to Display ☑ Data ☑ Average ☑ Median ☑ Deviation ☑ Throughput No of Samples 100 Deviation 47 Latest Sample 1206 Throughput 2,807.674/minute Average 1146 Median 1144	 Graphs to Display ☑ Data ☑ Average ☑ Median ☑ Deviation ☑ Throughput No of Samples 100 Deviation 49 Latest Sample 1141 Throughput 2,826.189/minute Average 1101 Median 1089

Table 3.14: Lazy Load - Create Hotel - Response Time

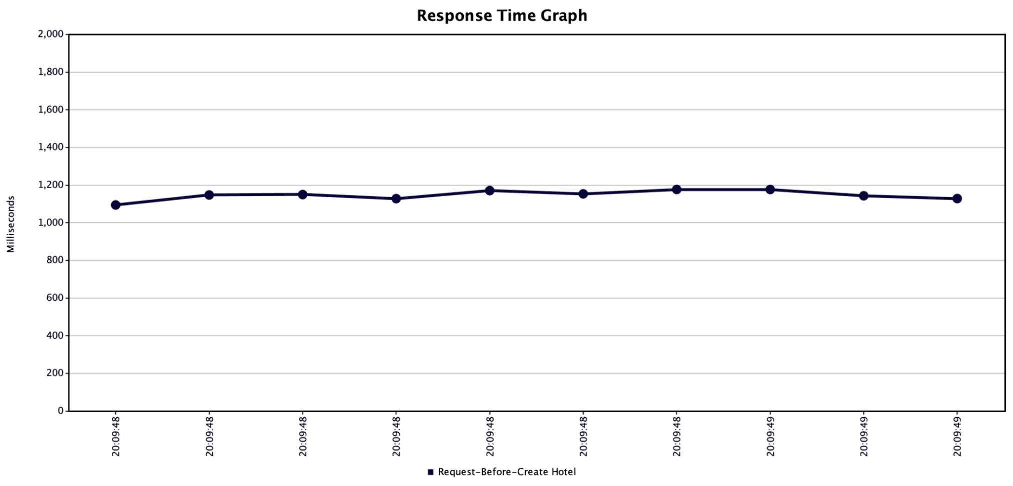
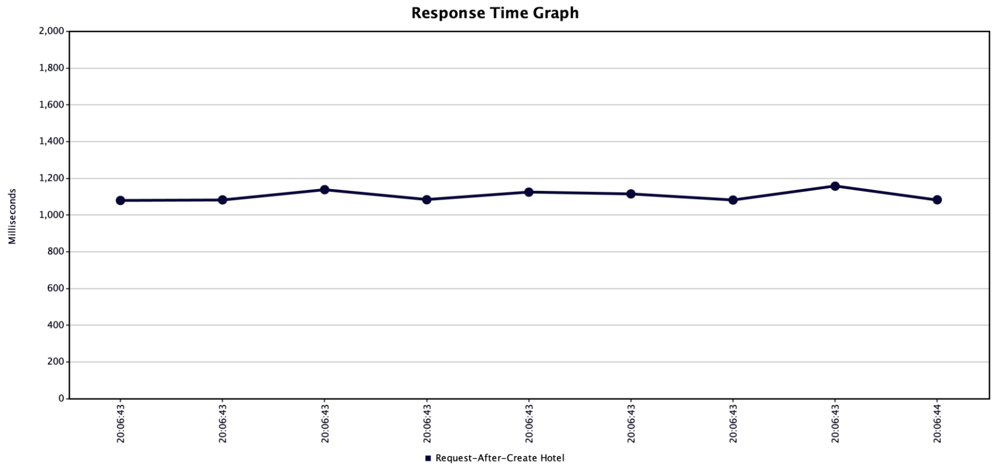
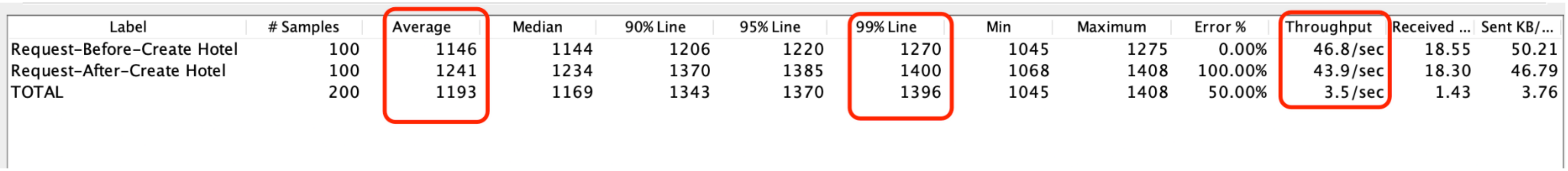
Response Time: Create Hotel	
Before “Lazy Load” implementation	**After “Lazy Load” implementation**
	

Figure 3.8: Statistics - Book Hotel



Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput	Received ...	Sent KB/...
Request-Before-Create Hotel	100	1146	1144	1206	1220	1270	1045	1275	0.00%	46.8/sec	18.55	50.21
Request-After-Create Hotel	100	1241	1234	1370	1385	1400	1068	1408	100.00%	43.9/sec	18.30	46.79
TOTAL	200	1193	1169	1343	1370	1396	1045	1408	50.00%	3.5/sec	1.43	3.76

3.3 Optimistic Offline Lock

3.3.1Examples and Justification

To reflect on the performance of our system, we decided to focus on the following analysis:

- To compare the used design pattern with a different design pattern
- To compare the performance of the system **before** and **after** the implementation of the **Optimistic Offline Lock** design pattern

The performance evaluation in this section was done over the following Business Transaction:

Table 3.15: Examples - Business Transactions evaluated

Actor	Business Transaction
Hotelier	ModifyHotel

After executing experiments systematically using Jmeter, we arrive at the following reflections on the performance of the system:

- **The implementation of Optimistic Offline Lock increased the response time of the system and reduced the system's throughput**. In a real-world scenario, the number of Hoteliers managing the operations of a hotel simultaneously is one, depending on the size of the hotel as stated in [7]; therefore the probability of concurrent requests would be extremely low. In this regard, we could have replaced the Optimistic Offline Lock implementation and let the database handle any update request to the hotel's profile. **This could have reduced our code complexity and increased performance** such as throughput and Response Time.
- As shown in **table 3.16**, **the system's throughput was reduced by almost 5%** after the implementation of the Optimistic Offline Lock. The implementation of a design pattern to handle simultaneous requests comes with the tradeoff of reducing the throughput capacity of the system.
- After implementing the Optimistic Offline Lock, we noticed that **the average response time of the system increased by 7%** as shown in **table 3.17**. To keep track of simultaneous business transactions, we needed to handle concurrent requests but this impacted the response time and performance of the service as shown in **figure 3.9**.

3.3.2Results

Modify Hotel Before:

In this section, we only modify the same hotel information to test Optimistic Offline Lock.

Table 3.16: Optimistic Offline Lock - Modify Hotel - Throughput

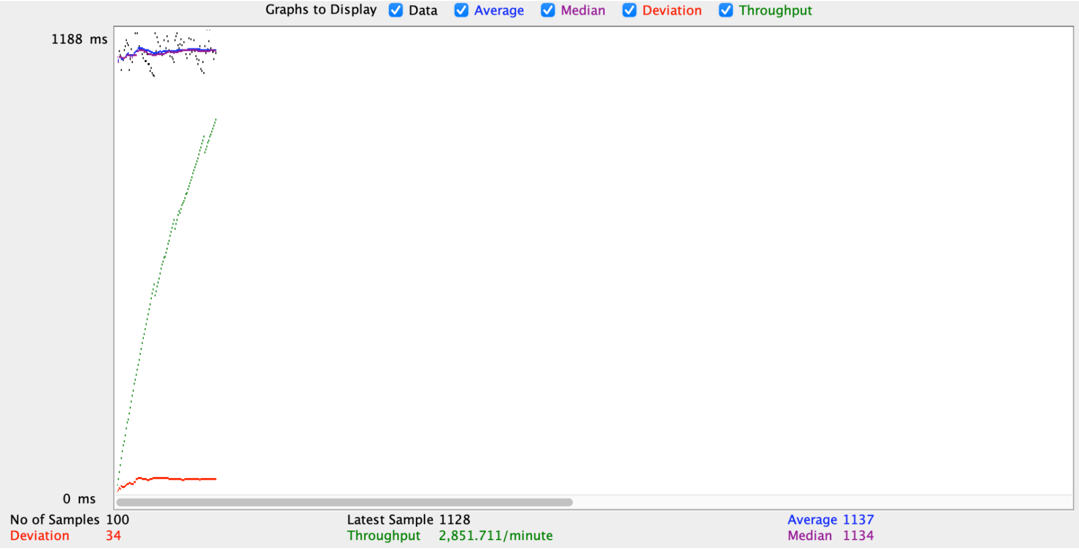
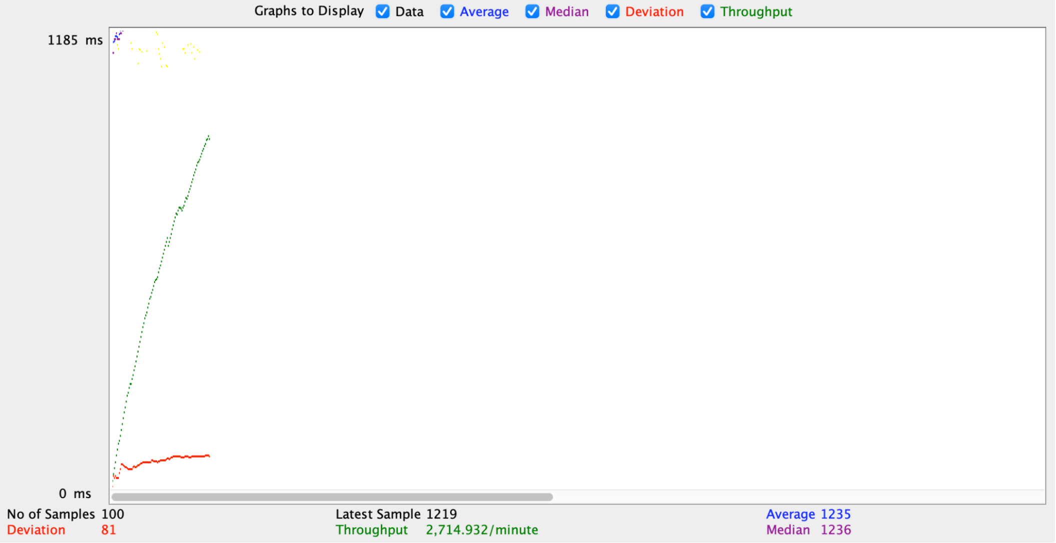
Throughput: Create Hotel	
Before "Optimistic Offline Lock" implementation	**After "Optimistic Offline Lock" implementation**
 Graphs to Display ☑ Data ☑ Average ☑ Median ☑ Deviation ☑ Throughput 1188 ms 0 ms No of Samples 100 Deviation 34 Latest Sample 1128 Throughput 2,851.711/minute Average 1137 Median 1134	 Graphs to Display ☑ Data ☑ Average ☑ Median ☑ Deviation ☑ Throughput 1185 ms 0 ms No of Samples 100 Deviation 81 Latest Sample 1219 Throughput 2,714.932/minute Average 1235 Median 1236

Table 3.17: Optimistic Offline Lock - Modify Hotel - Response Time

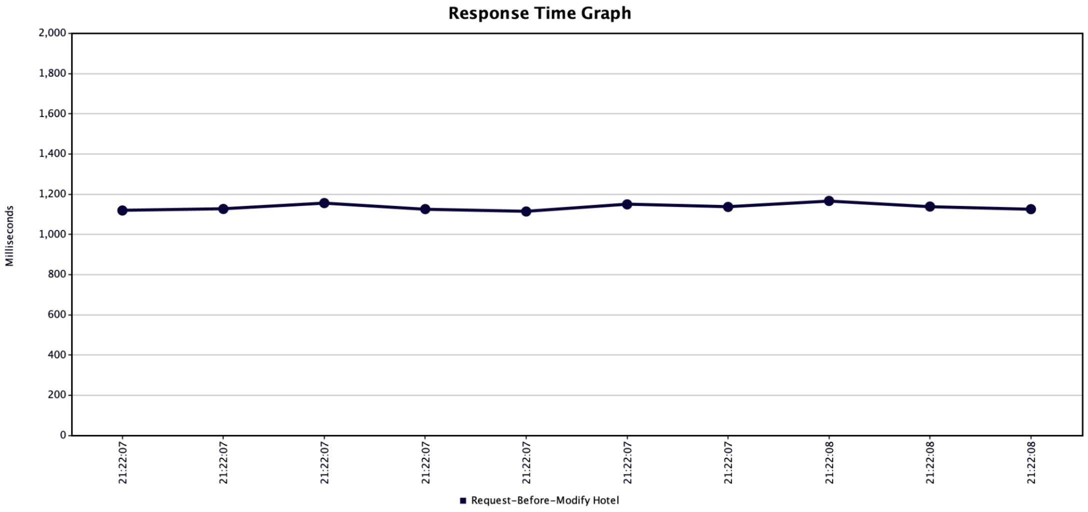
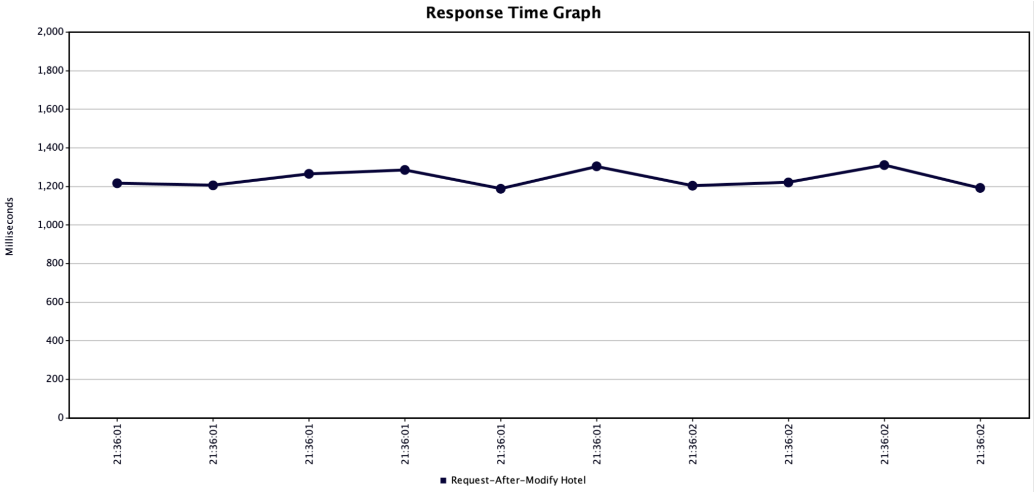
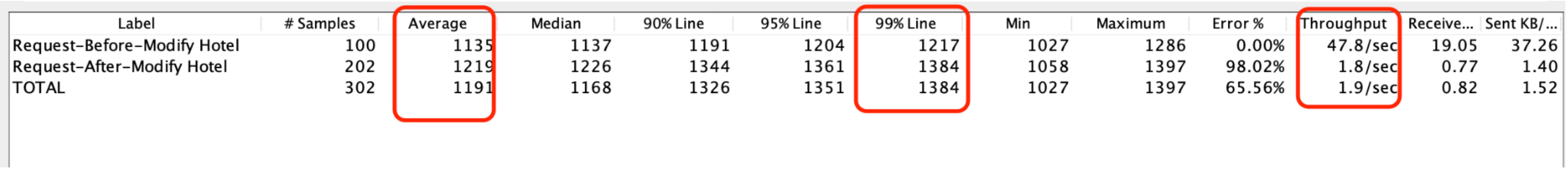
<table>
<tr><th colspan="2">Response Time: Create Hotel</th></tr>
<tr><th>Before “Optimistic Offline Lock” implementation</th><th>After “Optimistic Offline Lock” implementation</th></tr>
<tr><td> </td><td> </td></tr>
</table>

Figure 3.9: Statistics - Modify Hotel



Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput	Receive...	Sent KB/...
Request–Before–Modify Hotel	100	1135	1137	1191	1204	1217	1027	1286	0.00%	47.8/sec	19.05	37.26
Request–After–Modify Hotel	202	1219	1226	1344	1361	1384	1058	1397	98.02%	1.8/sec	0.77	1.40
TOTAL	302	1191	1168	1326	1351	1384	1027	1397	65.56%	1.9/sec	0.82	1.52

3.4 Pessimistic Offline Lock

3.4.1Examples and Justification

To reflect on the performance of our system, we decided to focus on the following analysis:

- To compare the used design pattern with a different design pattern
- To compare the performance of the system **before** and **after** the implementation of the **Pessimistic Offline Lock** design pattern

The performance evaluation in this section was done over the following Business Transaction:

Table 3.18: Examples - Business Transactions evaluated

Actor	Business Transaction
Customer	Book Hotel
	Modify Bookings

After executing experiments systematically using Jmeter, we arrive at the following reflections on the performance of the system:

- After analysing the resulting Response Time of a Pessimistic Offline Lock and compare it against the Optimistic Offiline Lock, as **tables 3.17 and 3.20** show, the system response time was highly impacted when using Pessimistic Offline Lock, therefore **our system response time could have been reduced if we used Optimistic Offline Lock** but with the tradeoffs of reducing the correctness capacity of our system.
- As shown in **table 3.19**, **the system's throughput was reduced by more than 12%** after the implementation of the Pessimistic Offline Lock. The implementation of a design pattern to handle simultaneous requests comes with the tradeoff of reducing the throughput capacity of the system.
- After implementing the Pessimistic Offline Lock, we noticed that **the average response time of the system increased by 25%** as shown in **table 3.20**. To keep track of simultaneous business transactions, we needed to handle concurrent requests but this impacted the response time and performance of the service as shown in **figure 3.10**.
- One option to improve the average response time of the system and not increasing the response time of the system, as shown in **table 3.20,** we could have **used the Optimistic Offline Lock instead of the Pessimistic Offline Lock**, and to ensure data integrity we could have created a timer to let the Customer that we will ensure his/her transaction progress only if the business transaction is finished before the timer expires. This is a typical process used by many real-world applications in the market.

3.4.2Results

In this section, we simulated multiple customers booking the same hotel at a time.

Table 3.19: Pessimistic Offline Lock - Book Hotel - Throughput

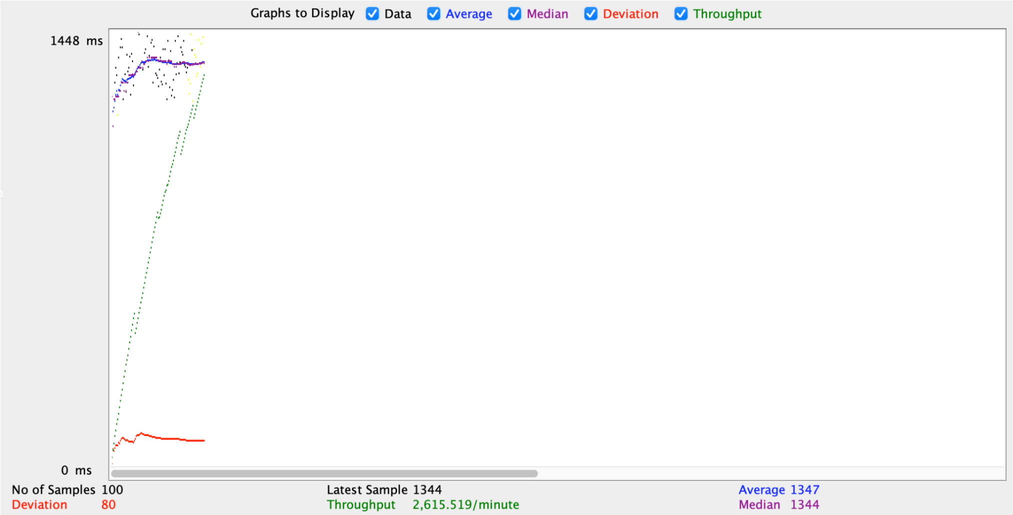
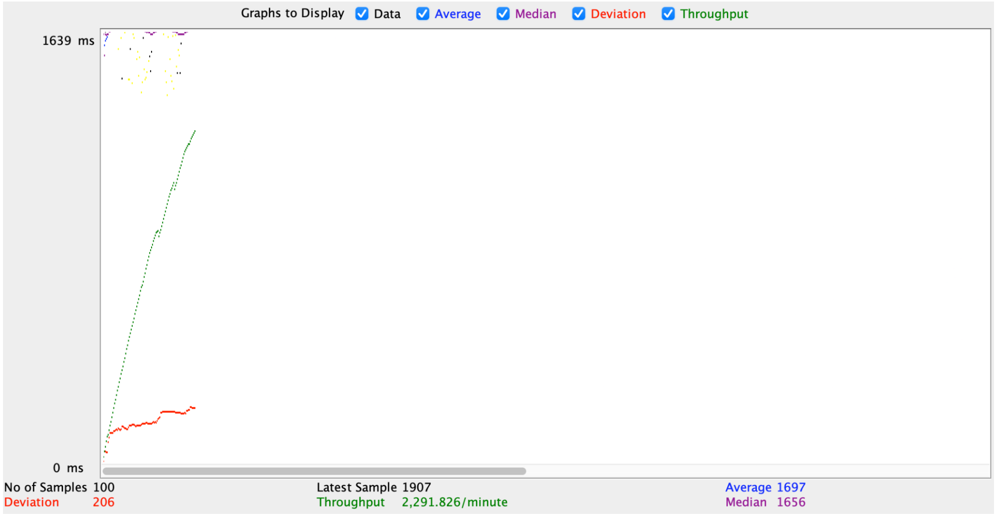
Throughput: Book Hotel	
Before "Pessimistic Offline Lock" implementation	**After "Pessimistic Offline Lock" implementation**
 Graphs to Display ☑ Data ☑ Average ☑ Median ☑ Deviation ☑ Throughput No of Samples 100 Deviation 80 Latest Sample 1344 Throughput 2,615.519/minute Average 1347 Median 1344	 Graphs to Display ☑ Data ☑ Average ☑ Median ☑ Deviation ☑ Throughput No of Samples 100 Deviation 206 Latest Sample 1907 Throughput 2,291.826/minute Average 1697 Median 1656

Table 3.20: Pessimistic Offline Lock - Book Hotel - Response Time

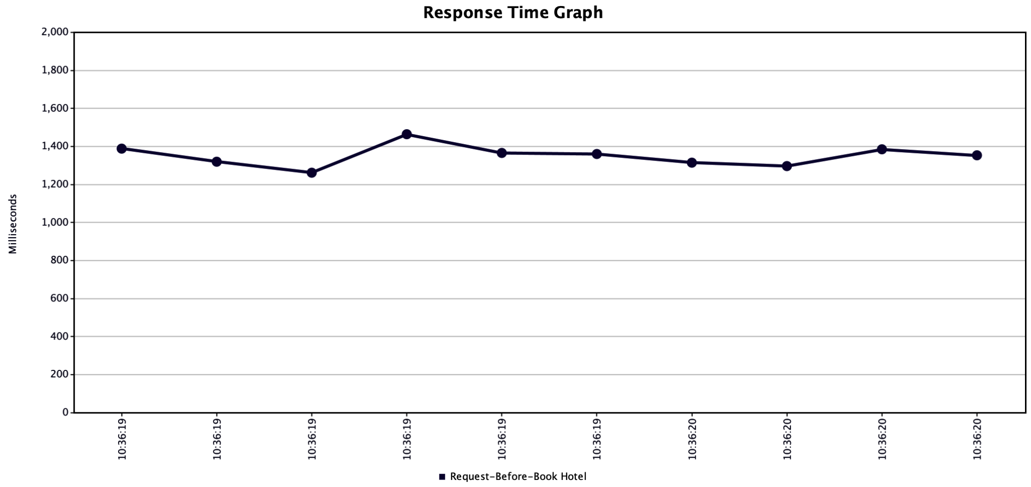
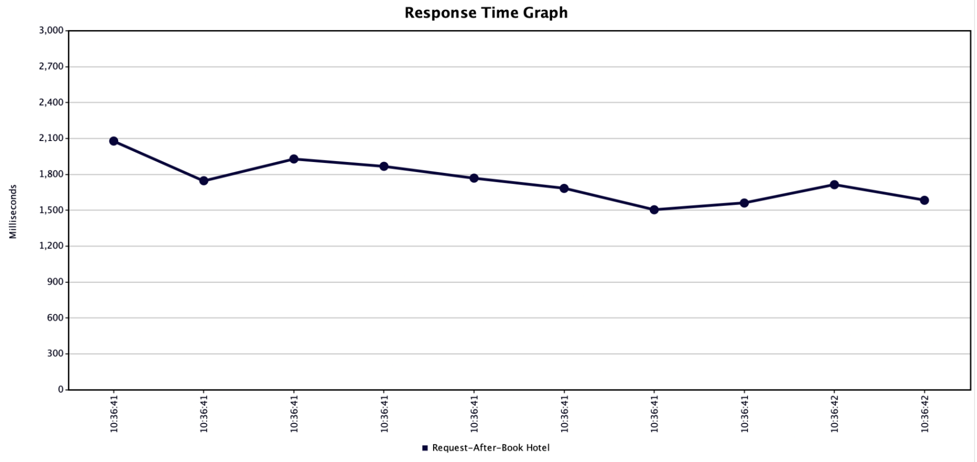
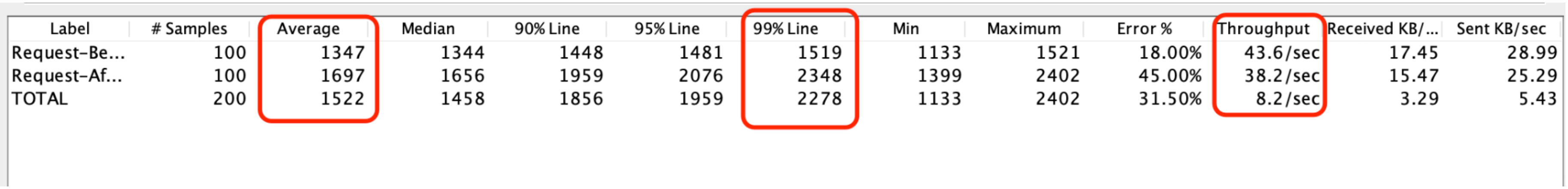
<table>
<tr><th colspan="2">Response Time: Book Hotel</th></tr>
<tr><th>Before “Pessimistic Offline Lock” implementation</th><th>After “Pessimistic Offline Lock” implementation</th></tr>
<tr><td> </td><td> </td></tr>
</table>

Figure 3.10: Statistics - Book Hotel



Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput	Received KB/...	Sent KB/sec
Request-Be...	100	1347	1344	1448	1481	1519	1133	1521	18.00%	43.6/sec	17.45	28.99
Request-Af...	100	1697	1656	1959	2076	2348	1399	2402	45.00%	38.2/sec	15.47	25.29
TOTAL	200	1522	1458	1856	1959	2278	1133	2402	31.50%	8.2/sec	3.29	5.43

3.5 DTO and Remote Facade

3.5.1Examples and Justification

To reflect on the performance of our system, we decided to focus on the following analysis:

- To compare the used design pattern with a different design pattern
- To compare the performance of the system **before** and **after** the implementation of the **DTO** design pattern

The performance evaluation in this section was done over the following Business Transaction:

Table 3.21: Examples - Business Transactions evaluated

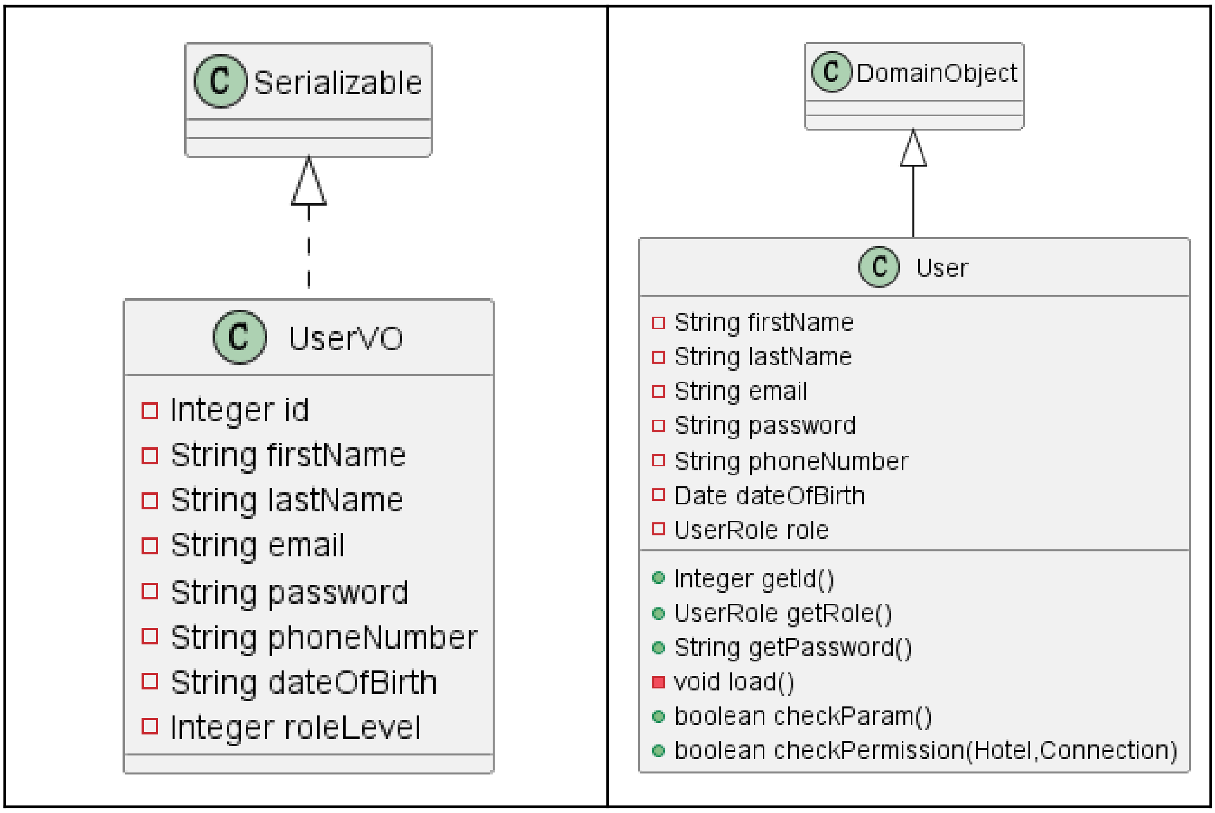
Actor	Business Transaction
Customer	Customer Sign Up
	Search Accommodation

After executing experiments systematically using Jmeter, we arrive at the following reflections on the performance of the system:

- Taking one of the data transfer object we implemented as an example:

Table 3.22: Comparison of data transfer object and domain object

User data transfer object	User object



Because our group is using frontend and backend separately architecture, we have to face the issue that passing the entire object with different data through network at very beginning. As the comparison above, the User object has type of date and enumeration data that can not be passed through the network, thus, we define a related User data transfer object which transfers the type of date to a string and type of enumeration to an integer. However, we did not realise this pattern is called "Data Transfer Object", and we named this type of object 'view object' because it is close to the view level of MVC architecture.

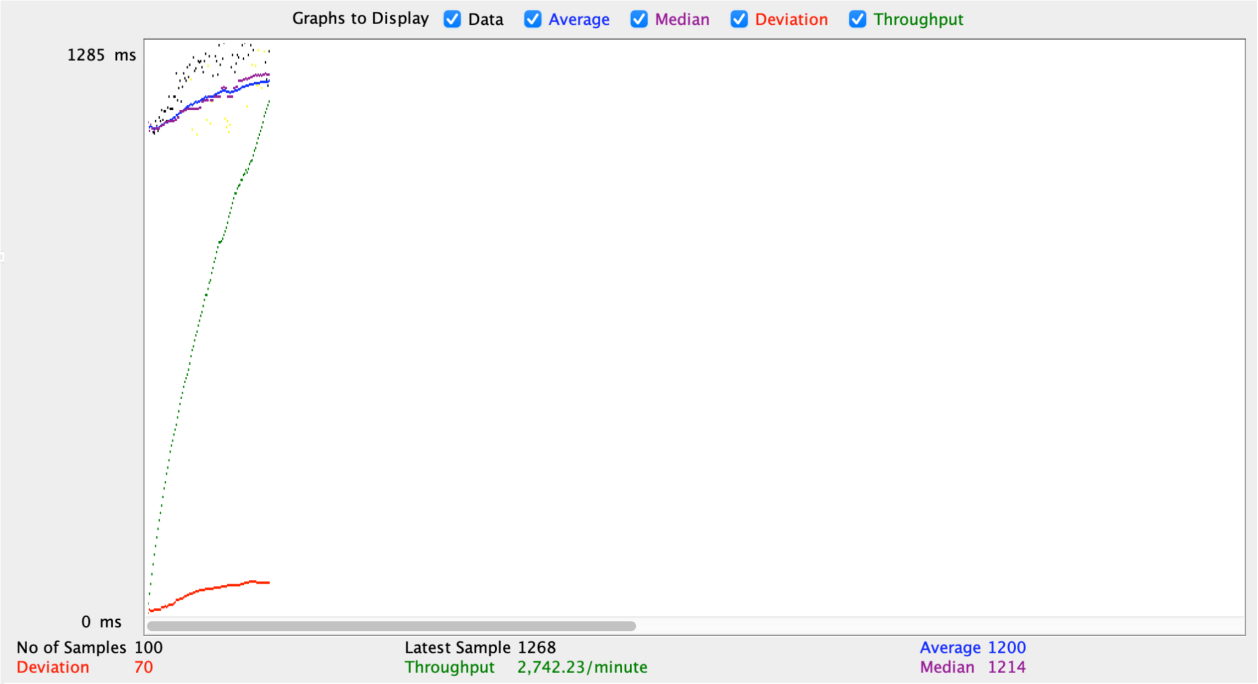
- As the JMeter result shows below, it proves that our DTO works really well and smoothly as we expect. It does have some advantages, such as making objects that can be understood by both front and back end, and reducing the number of calls by combining different attributes from different domain objects. But it does **improve the complexity** of the view and controller level of our system and the **low cohesion** as we have to use data transfer objects to new another domain object.

- Unfortunately, we do not have enough time to implement the remote facade. But we have some discussion on the remote facade pattern, and we all thought this pattern could improve our system performance. Because we have several complicated objects that should be serialised and sent over the network in our system, these objects always make the request and response data messy and large. If we could decrease the granularity of this kind of object, **it would not only improve the performance of response time but also will improve the readability of the code and data.**

3.5.2Results

Customers Sign Up:

Figure 3.11: DTO and Remote Facade - Customers Sign Up - Throughput



No of Samples 100
Deviation 70
Latest Sample 1268
Throughput 2,742.23/minute
Average 1200
Median 1214

Figure 3.12: DTO and Remote Facade - Customers Sign Up - Response Time

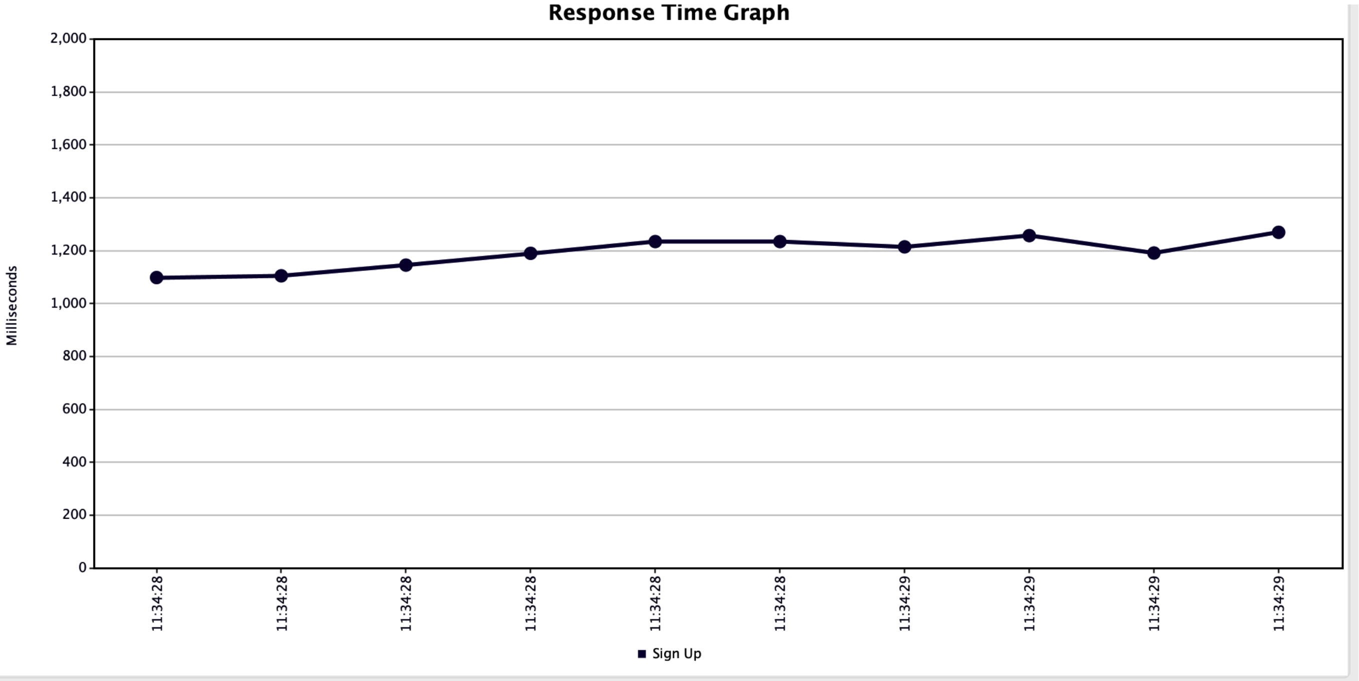
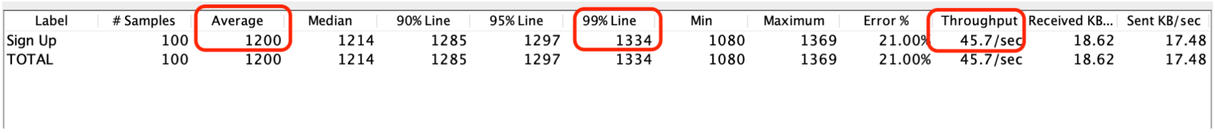


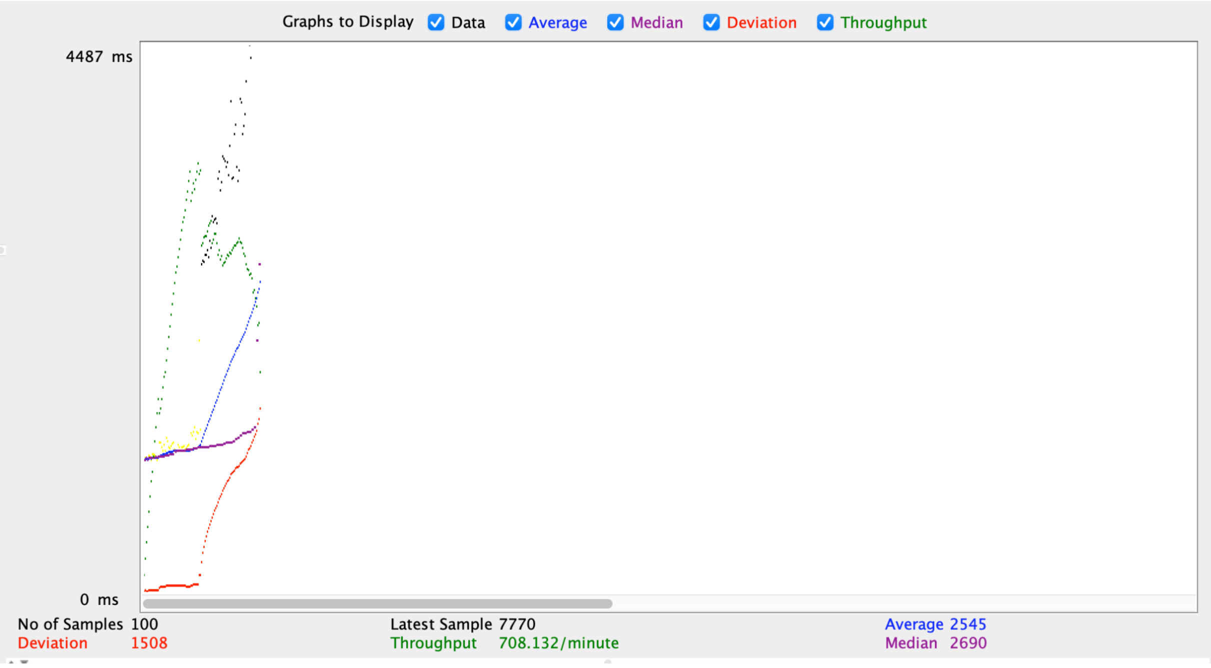
Figure 3.13: Statistics - Sign Up



Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput	Received KB...	Sent KB/sec
Sign Up	100	1200	1214	1285	1297	1334	1080	1369	21.00%	45.7/sec	18.62	17.48
TOTAL	100	1200	1214	1285	1297	1334	1080	1369	21.00%	45.7/sec	18.62	17.48

Search Accommodation:

Figure 3.14: DTO and Remote Facade - Search Accommodation - Throughput



Graphs to Display ☑ Data ☑ Average ☑ Median ☑ Deviation ☑ Throughput
4487 ms
0 ms

No of Samples 100

Deviation 1508

Latest Sample 7770

Throughput 708.132/minute

Average 2545

Median 2690

Figure 3.15: DTO and Remote Facade - Search Accommodation - Response Time

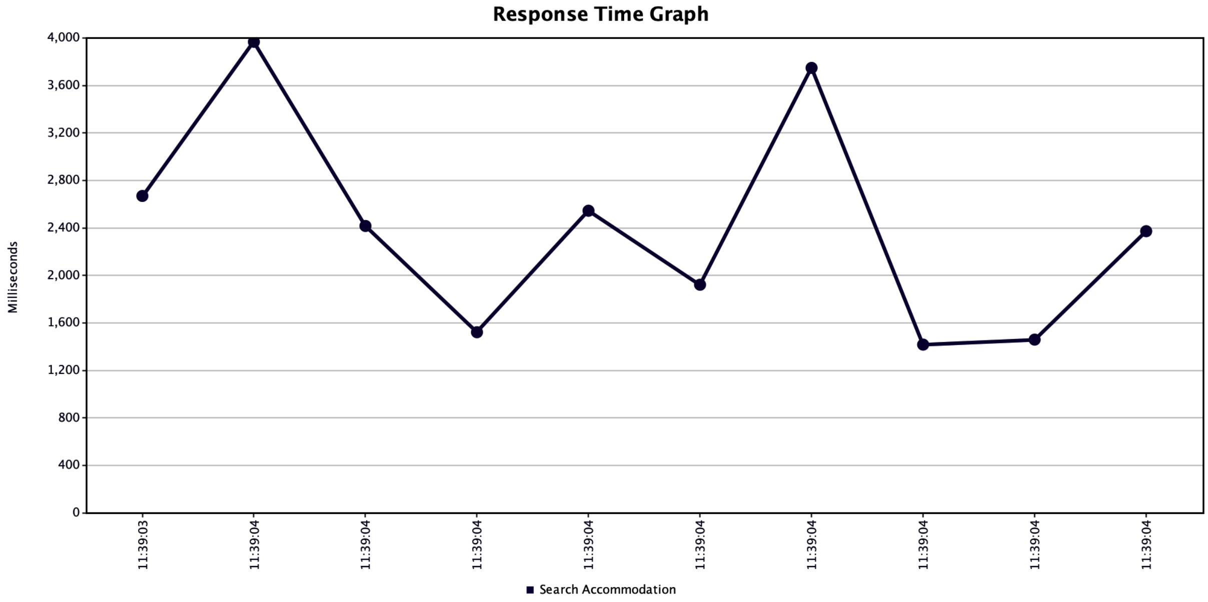
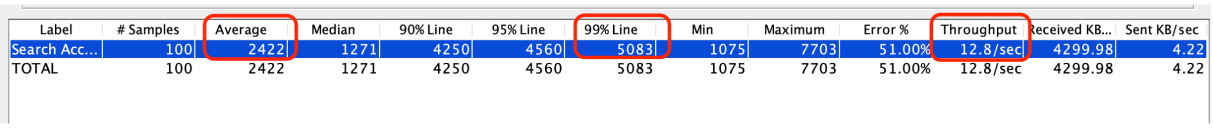


Figure 3.16: Statistics - Search Accommodation



Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput	‍Received KB...	Sent KB/sec
Search Acc...	100	2422	1271	4250	4560	5083	1075	7703	51.00%	12.8/sec	4299.98	4.22
TOTAL	100	2422	1271	4250	4560	5083	1075	7703	51.00%	12.8/sec	4299.98	4.22

3.6 Identity Map

3.6.1Examples and Justification

The identity map has numerous uses, but in terms of performance, it is a pattern that aims to reduce data fetching redundancy by storing data that's already been queried, so that if it is required again within the same business transaction, it can just be looked up a local map rather than queried again.

An identity map can have obvious benefits for business transactions that require using the same data object multiple times, as querying a database can often be a large bottleneck in business transactions, skipping that step and looking up the value locally could lead to a significant performance boost in certain circumstances.

Although the hotel booking system does not have many use cases where the same data needs to be loaded more than once, there were some use cases that did, such as fetching all bookings which (as some may book the same hotel), which would benefit from an identity map that improved data querying efficiency.

4. Design Principles Discussion

4.1 Pipelining

- **Reflection 1: Multi-threading and Servlets**
  In our backend, we use the servlet to manage our application. **Servlet natively uses threads to handle multiple requests at the same time**. Thus, one request does not need to wait for the previous request complete, each request is independent. **The multi-threading mechanism helps the system improve the performance of concurrency**. Due to Heroku's limitation, we cannot manage and configure the number of the database connection. But in the future, if we have full control of a backend server, we could design and use our own thread pools and database connection pools, to help the system handle more concurrent requests.

- **Reflection 2: Independent tasks**
  We decided to use pipelining principle whenever possible on each of our front-end React pages such as requesting the Hotelier information about the list of Hotels and Bookings independently. This worked in accordance with pipelining principle and **helped us to reduce the response time (Latency) of the requesting and receiving data** that are independent of each other, improving the customer experience since the loading time of a webpage is reduced.

- **Reflection 3: Room for improvement**
  Although pipelining was something we were able to implement between the client front end and our REST API written in Java, it was not implemented for fetching data from the Heroku PostgreSQL instance. While there were a few **endpoints that would have benefitted from pipelining** as fetching data would have been theoretically quicker, the use cases for multiple requests between the server and database were very minor in comparison to between the server and frontend, which is why pipelining was implemented there instead.

4.2 Bell's Principle

- **Reflection 1: Designing for simplicity**
  The code written was designed based on well-known patterns which helped us to keep the **code with a high level of simplicity and easy to scale up** if required. Writing understandable code with a pattern that fits the purpose of a particular type of problem lets us simplify the code to the point that it is **easy to reuse** whenever needed; in this manner, no extra code is needed. Therefore, in accordance with Bell's principle, **with no extra code then, no extra debugging will be needed.**

- **Reflection 2: Reducing complexity**
  While we did our best to maintain the simplicity of the overall system, there are tradeoffs that were considered during design that added complexity and boilerplate, but for good reason. **It could be argued that in terms of components, transaction scripts would be relatively simpler**, as they require less code and separate components to work which by Bell's principle would result in better performance. **However, the system ended up using a more layer architecture centered around a domain model**, and while this not be as simple code-wise as compared to a transaction script, it **improves the overall readability and extensibility of the system**. For the design path we chose, we did indeed minimize unnecessary complexities that allowed the system to work at a satisfactory performance level.

- **Reflection 3: Readability and Maintainability**
  After implementing several patterns, we have known the importance of keeping the simplicity of the code, because **simplicity also brings readability and maintainability.** However, it is ideal for keeping all the components and coding simple because we need to keep the availability and robustness of the system. Thus, what we try to do is to keep the simplicity of each part locally as much as possible. For example, we are using a 'Result' paradigm class to encapsulate the response data and result because we are using JSON serialisation to communicate between the front end and back end, but it will make the data mapper layer much more complex and hard to read if we implement this into data mapper instead of using exception directly. Therefore, the way we try to keep the data mapper layer pure and clean is quite close to Bell's principle. As there are numerous critical properties of a system, it is always a compromise between simplicity and performance.

4.3 Caching

- **Reflection 1: Room for improvement**
  In our system, we have not implemented the caching principle. However, the way we implement lock manager is quite close to the concept of caching, which is to store and access the lock data in JVM. Because we implement the pessimistic offline lock on booking hotels

which is the most important and frequently used logic of our system, the lock will be added and removed very frequently. If we choose to store the lock data in the database, it will double the total number of database access while users are booking the hotel. Besides, the caching principle could be used for some frequently accessed data, such as the basic information of some hotels. And we could define a strategy to replace and store some most frequently searched hotels into caching, which **could obviously improve the performance of searching**. Moreover, **we could also implement some NoSQL databases** such as Redis to improve the performance of user management. Because we are using JWT as authentication and the token is a stateless hash value string, it is best to implement a token management sub-system to control the status of the token. Storing such data into NoSQL databases based on caching memory instead of relational databases will improve the performance of the system essentially.

- **Reflection 2: Memoization**
  In our front-end application, **we did not use memoization to cache the data we fetched**. In the future, we could use React memoization to temporarily store some datasets that may be very large in the front end. The benefit is the user does not need to fetch the data again and again, which is time-consuming and wastes network traffic. Using memoization to cache, **we could improve the response time of the application**.

- **Reflection 3: CDN Technology**
  We did not use CDN in this project. However, using **CDN could significantly improve our website's latency when our application becomes world-famous.** We could cache some static content, such as images, text, promotion videos or the latest news about a famous travel place. People worldwide could get information faster because they can get the content from the server close to them. Using CDN could also help our system provide more reliable content since each distributed server also caches the content. One server's down will not affect the entire system.

5. References

[1] Larman, C. (2005). *Applying UML Patterns: an introduction to object-oriented analysis and design and iterative development (3rd ed.).* Pearson Education, Inc.

[2] Cockburn, A. (2016). *Writing effective use cases.* Boston Addison-Wesley

[3] Oliveira, Eduardo and Rodriguez, Maria. Lecture Notes, Topic: *"SWEN90007. Software Design and Architecture"*. Engineering, Melbourne, Victoria: Department of Computing and Information Systems, Melbourne University, 2022

[4] PlantUML. *Domain Model Tool Creator*. https://plantuml.com/commons

[5] Visual Paradigm Online.
https://online.visual-paradigm.com/diagrams/solutions/free-visual-paradigm-online/

[6] Heroku. Concurrency and Database Connection. *"Maximum Database Connection"*. The last date accessed: 2022-10-19

https://devcenter.heroku.com/articles/concurrency-and-database-connections#maximum-database-connections

[7] OZ Studies. *"How To Become A Hotel Manager In Australia"*. The last date accessed: 2022-11-02
https://www.ozstudies.com/blog/australia-careers-guide/how-to-become-a-hotel-manager-in-australia