System Performance Report

Submission 4

Team LANS

SWEN90007 SM2 2022 Project

In charge:

Mohammad Saood **Abbasi** <mohammadsaoo@student.unimelb.edu.au>

Arman **Arethna** <aarethna@student.unimelb.edu.au>

Navdeep **Beniwal** <nbeniwal@student.unimelb.edu.au>

Levi **McKenzie-Kirkbright** <levim@student.unimelb.edu.au>

1 Revision History

Date	Version	Description	Author(s)
23/10/22	01.00-D01	Create document	Levi
23/10/22	01.00-D02	Outline sections	Levi
23/10/22	01.00-D03	Introduction section and expected references	Levi
23/10/22	01.00-D04	Section 7.2.3 on caching (included)	Levi
03/11/22	01.00-D05	Section 6.3.1 on caching (improvement)	Levi
03/11/22	01.00-D06	Proofreading and structuring	Levi
02/11/22	01.00-D07	Section 4.2.2 excluded pipelining	Arman
03/11/22	01.00-D08	Section 4.3.2 pipelining with graphQL	Arman
02/11/22	01.00-D09	Section 4.1.1 Bell's Principle	Saood
03/11/22	01.00-D10	Section 4.3.3 Replacing Front Command with Concrete Controllers	Saood
03/11/22	01.00-D11	Section 4.2.1 Bell's Principle: MVC is a more straightforward architecture	Navdeep
03/11/22	01.00-D12	Section 4.2.3 Caching: using a CDN	Navdeep

2 INTRODUCTION

2.1 Proposal

This document is the System Performance Report. It is the **fourth** submission of the project for SWEN90007 Software Design and Architecture, 2022.

2.2 Target Users

This document is intended for SWEN90007 teaching staff and other SWEN90007 students, particularly the LANS team.

2.3 Conventions, terms, and abbreviations

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

Table 1: terms used in this report.

Term	Description
"Team" or "the Team" or "LANS"	The team that created the project. The names of participating students are listed on the front cover of the report.

2.4 Links

Heroku deployment: https://swen90007-2022-lans.herokuapp.com/

Repository: https://github.com/SWEN900072022/SWEN90007-2022-LANS

Tagged release: SWEN90007_2022_Part4_LANS

3 CRITICAL ANALYSIS

LANS Hotels is a web application for finding and posting hotel listings. We discuss and critically analyse the performance characteristics of the LANS Hotels system, focusing on the three principles: Bell's principle, pipelining and caching. Our focus during implementation was not on performance but on delivering the minimum functionality. Our system minimises performance principles, primarily Bell's principle and pipeline. As such, we focus our analysis on the reasons for the exclusions of principles during the implementation of the system and discuss the potential opportunities of enhancing the system's performance if we continue refactoring and extending the system.

An essential aspect of the discussion is the architecture of the system. To briefly summarise: our system is a React single-page application (a "rich client", according to Fowler) interacting with an API server (*Front Command*, Fowler 344) persisting data on a managed instance of PostgreSQL hosted on Heroku. We used a third-party authentication provider, Auth0, to assist with authentication using JSON Web Tokens. We also implemented role-based authentication based on a mapping between emails and roles by replicating emails authenticated by Auth0 to the APP_USER table in our database. Our UI communicated with our backend server and with auth0 via Representational State Transfer (ReST) over HTTP using JSON as the data interchange format.

3.1 Included performance principles

3.1.1 Bell's Principle

"The cheapest, fastest, and most reliable components of a system are the ones that aren't there." - Gordon Bell.

The above principle states that increasing complexity does not always correlate to higher performance. If the application flow is undisturbed with the removal of a specific component, then the design should not have considered it in the first place.

The elements of complexity from the code's perspective were explored using three different approaches while coding.

1. The Halstead complexity metric checks for four elements:
    a. Number of distinct operators
    b. Number of Separate Operands
    c. Total Number of occurrences of operators
    d. Total number of circumstances of operands

The above four elements throughout development were kept to a minimum. This ensured a more straightforward process for the prediction of mistakes during testing. While this may not have direct performance implications in decreasing code run time between processes, this approach eased the task of finding optimal alternatives during internal code reviews.

2. Cyclomatic complexity was reduced as well through linearly independent paths that helped during the development and testing of functionalities independently. The design ensured abstraction at each layer with segmented distributions from the presentation layer to a concrete command funnelled through an abstract front controller. The concrete command is linked to an independent use case called its corresponding data source layer connecting to the database. This path was followed back to transfer information to the front-end client, making each path unique and independent.
3. The above two metrics further contributed to a lesser cognitive complexity of the code. This, coupled with standardised variable naming and a decrease in large functions, made the process of code improvement easier.

The notable design decisions taken to reduce code complexity were:

1. Offline Concurrency: Concurrency was tackled through an optimistic offline lock wherein a version number corresponding to a record was read and updated on change. The update was successful only when the version number of the row matched that of the one in the database. This avoided a stream of complications that could arise through implementing a record lock in the database during updates. The application could render high-volume transactions without maintaining a connection with the session.
2. Search Criteria: The search criteria approach was another means to reduce overall code complexity enabling a more straightforward design. The search criteria ensured that multi-parameter queries from the front end would be constructed through an abstracted class which would add appropriate structure before sending it to the database. This meant that any further optimisations had only to be done in a single place and that development throughout would be reliable. This approach, coupled with an abstracted use case classes approach discussed above, provided a high degree of abstraction, increasing code readability and maintaining reliability.

3.1.2 Caching: speeding up authorisation

We used in-memory caching of our custom authorisation object to speed up per-request authorisation by attaching an auth object to the servlet session object. We need to secure our API on every request, and this design worked seamlessly with our request handling and concurrency model. The authorisation process was required to be as fast as possible so that it did not create a pervasive bottleneck across all our API endpoints, which would slow down every single API request.

We “piggybacked” on the Servlet request handling model to achieve per-request security. Servlets use a “one thread per request” model, which attaches a HttpSession object to each request. We cached authorisation in the session, giving each request access to the same authorisation object. As a bonus, requests from the same user simultaneously have access to the same session object.

Each request hits “/api/*” and is mapped to the “APIEntrypoint.java” servlet. Before a request is routed to the appropriate controller, we check to ensure the controller can access the authorisation information of the sender. On the first request for a particular session, we add an “auth” object to the session. If there is a JWT in the Authorisation header and it passes validation, we create a user object and store a reference to that user in the session’s auth object. We aimed to minimise the frequency of fetching user data from our database by caching the user.

A *cache miss* corresponds to the situation when the session does not yet have an auth object. There are two cases where there could be a cache miss. When the user does not exist, we create the user object by extracting their email from the JWT and adding them to our database. When the user exists, but it is the first request of the session, the user object must be reconstituted from the existing user data in the database. After creating or reconstituting the user, we store a reference to the user in the auth object and attach the auth object to the session.

A *cache hit* corresponds to the situation when there is already an auth object for the session. This auth object references the associated user object used for role-based authentication.

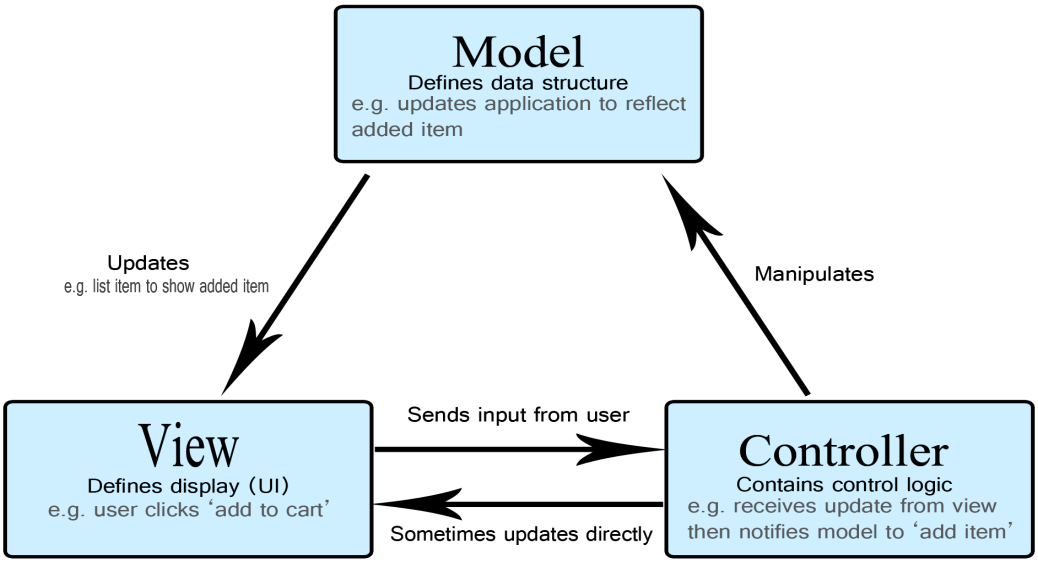
This design positively impacted the performance of our per-request authorisation compared to alternative implementations. For example, looking up the user in the database on every request incurs an expensive disk IO operation unless the database is configured to perform temporal caching. Regardless of database caching, since we used a managed database instance hosted on Heroku, each request to the database server also incurs a network IO operation over the internal network of the data centre hosting Heroku. As we have no insight into the co-location or networking of Heroku’s infrastructure, we cannot make assumptions about latency. Keeping the authorisation in memory reduced the exposure to disk and network IO performance penalties when looking up the user.

Code: using Servlet session as a cache for authorisation

```
//  Location: /src/main/java/lans/hotels/api/auth/Auth0Adapter.java
public static Auth0Adapter getAuthorization(HttpServletRequest request, Callable<Void>
onUnauthorized) {
    Auth0Adapter auth = (Auth0Adapter) request
                                        .getSession()
                                        .getAttribute(Auth0Adapter.AUTHORIZATION);
    if (auth == null) auth = new Auth0Adapter();
    auth.onUnauthorized = onUnauthorized;
    return auth;
}
```

3.2 Excluded performance principles

3.2.1 Bell's Principle: MVC is a more straightforward architecture



MVC (Model - View - Controller) is an architectural pattern that divides the application into three components.

- Model: The models are responsible for managing the data and further access to this data.
- View: The views are responsible for initiating the UIs that the user sees and further updating those UIs based on user actions and requests.
- Controller: The controllers are responsible for routing commands to the correct model, which then updates the views according to the changes made to the model. The controller can also update the UI if the request does not require access to any further data.

The main advantage of using an MVC is the separation of concerns, as seen above, which makes a large codebase more maintainable and makes it easy to integrate more functionalities into the application.

MVC has a tighter coupling with the other components of the system compared to a React Single Page Application, which only needs to follow the defined REST protocols to access the data. These REST protocols are often standardised, allowing the client-side application to seamlessly integrate with the backend regardless of how the backend is implemented. We preferred this approach in contrast to MVC as our main concern was the development of a robust backend for the system using the various

architectural patterns taught in the class, and using MVC allowed us to develop the backend without worrying too much about how it is going to be integrated and used by the frontend. Using this approach instead of MVC also made our application more extensible, which was an area of focus for all of us from the start of this project.

Modifying the existing system to follow the MVC pattern is not that hard (at least from a design perspective), given that we have followed the MVC pattern closely. We already have data models established within our system, which currently fall under the domain and data source layer. Theoretically, we also have the controller component setup, given that we had to use the FrontController pattern, which further routes the request to more dedicated controllers to parse the HTTP requests and execute different use cases based on the resource requested and the request type. The primary adjustments on the controller side would require making these dedicated controllers to be directly accessed with simple, functional calls from the views component by setting up a mapping between action commands invoked from the views with the dedicated controllers and their request handlers, as well as the front controller pattern to be removed. Also, one significant change that the system would require would be to redesign and reimplement the views component, such as JSPs with direct access to the controllers, which currently sits as a separate Single Page Application designed to talk to any backend by following the REST interfaces and protocols defined by the backend application.

One significant overhead that the current approach of using REST-based interfaces and SPA for the client side is that it requires the use of the FrontController pattern, which acts as the gateway for the backend application, and is responsible for finding an appropriate controller to handle the request based on its URL. The computation done by the front controller to find the proper controller in the file system brings a significant overhead as files and directory lookups are generally expensive. Another overhead that the application is facing is in the operations that involve marshalling and un-marshalling data sent to and from the application along with the added operations to parse the different requests, which not only includes parsing and mapping the URL and request type to the correct use case but also requires operations that would need to be performed to check the syntax and validity of the incoming request. All these checks and the data serialisation operations could be avoided using the view components such as JSPs that have direct access to this data.

3.2.2 Pipelining: intentional over-fetching for entire user flows

The load time of a web page when using data requests to a server to fetch data can be decreased by refactoring the requests made to implement pipelining. This primarily applies to web pages that must fetch large amounts of data using a single large request. Pipelining of requests could be by sending multiple requests at once or splitting large requests into multiple smaller requests.

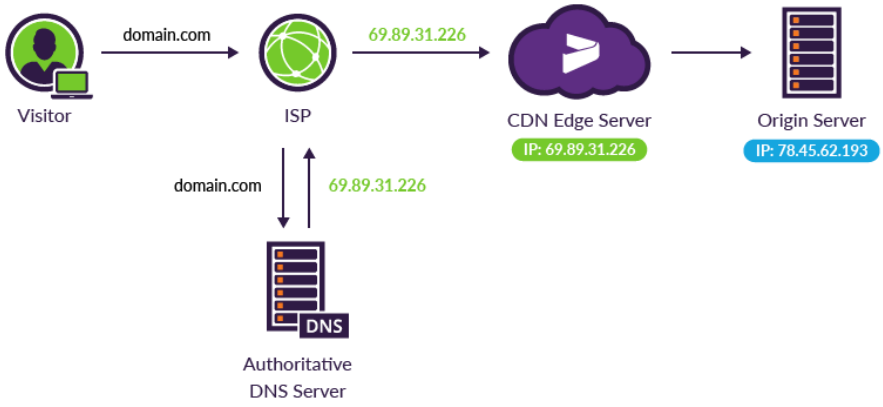
While designing the hotel management system, the team was focused more on the correctness and integrity of the data being fetched. There was a higher weightage on server-side pattern implementation and lower importance given to front-end design and usability. Pipelining requests would only affect usability; hence, it was not a priority, as the only gain would be reduced rendering time and a slight improvement in user experience. The rendering time issue would only arise when the database was scaled to have a large amount of data and rows and was flooded with multiple concurrent user data requests. The team was satisfied with the rendering times when testing the front end for primary concurrency use cases. Hence, they left the data requests in their original form without pipeline-based refactoring.

In the current LANS hotel system, refactoring large user data requests to include pipelining concepts could be done for data retrieval use cases like seeing all available hotels for users or viewing all hotel group bookings by hoteliers. In both these cases, the number of rows being read and fetched from the database might be extremely high. The back end could split the query into more minor queries so that we only show 10 (or any arbitrarily decided number) hotels or bookings simultaneously

and navigate to the next set of 10 using the front end. This would help the pages load faster for the user and improve user experience.

The impact of modifying all user data requests to account for pipelining will only be seen on page rendering times. The rendering times could be reduced by significant fractions depending on the amount of data being fetched and the complexity of the query. The. The speed of rendering also depends on the speed of the database itself in generating responses and the number of queries it can handle simultaneously. These factors cannot be improved by pipelining and are inherent tradeoffs depending on the chosen database implementation.

3.2.3 Caching: using a CDN



A CDN is a network of servers called edge servers that are placed worldwide to reduce latency for requests coming from a location that is too far from the origin server by responding with data from an edge server located near the origin of the request. These edge servers are responsible for fetching copies of original data from the origin server and then temporarily storing those copies based on the demand in that region. We can utilise this approach to cache temporal data, e.g. static assets related to hotels, such as images and videos that are most popular among the customers in a region, or spatial data, such as assets related to other hotels in a similar location or price range on an edge server. Furthermore, the data can be further categorised based on its demand in various regions and then prepositioned onto edge servers close to the areas where the need for a given data is higher.

Since the application is not supposed to be used commercially and can be considered a prototype at best, making use of a CDN would only have been a waste of resources and would have brought added cost that is not justified. Also, in case we do decide to utilise it as a commercial application, using a CDN at the start would only hurt the performance as most of the CDNs use a pull model for caching files, which means that the cached files would be dropped after not being used for a while which would be a frequent occurrence as initially, we would neither be aware of the data which is frequently requested and hence needs to be cached, nor we would have the amount of traffic it would require to extract out this data.

An advantage of using this approach is that it does not require changes to the system design or implementation. Many services are available, such as Edge and AWS CloudFront, which could be easily integrated into the application as an add-on service. All it requires is some configuration steps.

As discussed above, setting up a CDN for the platform would only hurt the performance given that the site is not receiving many requests or if the request received by the server is highly distributed in terms of data being accessed, which would fill up the cache with random data that has no pattern. This would

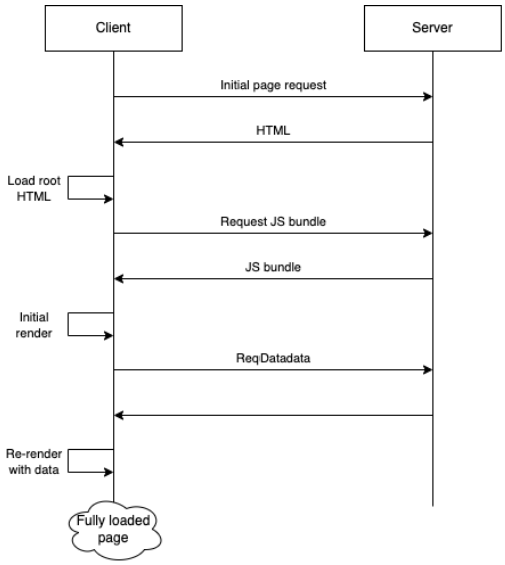
further mean that for each incoming request, there is a higher chance that the CDN server cache would not have the requested information and would require another call to the server, hence resulting in a delay for almost every request, which could be easily avoided by not using a CDN. Although, given that the site has become popular and traffic has increased so significantly such that the throughput by the server is not able to compensate for the latency, and we also have a promising idea of what data is being accessed more frequently, caching such data onto a CDN would provide a superior performance boost by both reducing the latency that the users experience, as well as in the number of requests the server would need to process.

3.3 Improvements

3.3.1 Caching: Jamstack architecture for the frontend to leverage CDN-based deployment

We built our application as a React JS single-page application. In the traditional single-page web applications architecture, a single large JavaScript file is sent to a client without any live data. The client makes one or more requests to the backend to load data for a view. As seen in the diagram below, this requires a minimum of three requests to get to a fully loaded page when the client first loads the app. Furthermore, the app fetches data from the backend as the client navigates through and interacts with the app.

Figure: requests to get to full page load



One increasingly popular method for improving the speed of the user experience is the “jamstack” architecture. The core idea of the jamstack is to pre-render your single-page application and distribute it on a CDN. This approach provides fine-grained control over the application’s data-fetching strategy. An app can select which UI components and pages should be fully rendered on the server and distributed on the CDN and which UI elements should be “hydrated” with new data on the client by standard SPA-style fetching. A typical pattern is to pre-render common data or data that does not change often and to hydrate authenticated or user-specific data, such as user profiles.

Our application would benefit significantly from adopting this deployment approach. For example, we could pre-render our homepage and cache it on a CDN, updating it as needed (e.g., when new hotels are added). The user would then have a single, fully loaded page when they land on our website instead of loading the root HTML, requesting the JS bundle, and then fetching the hotels. Suppose the home page stays mostly the same. In that case, a significant amount of redundant computation, including database lookups, is performed every time someone navigates to our home

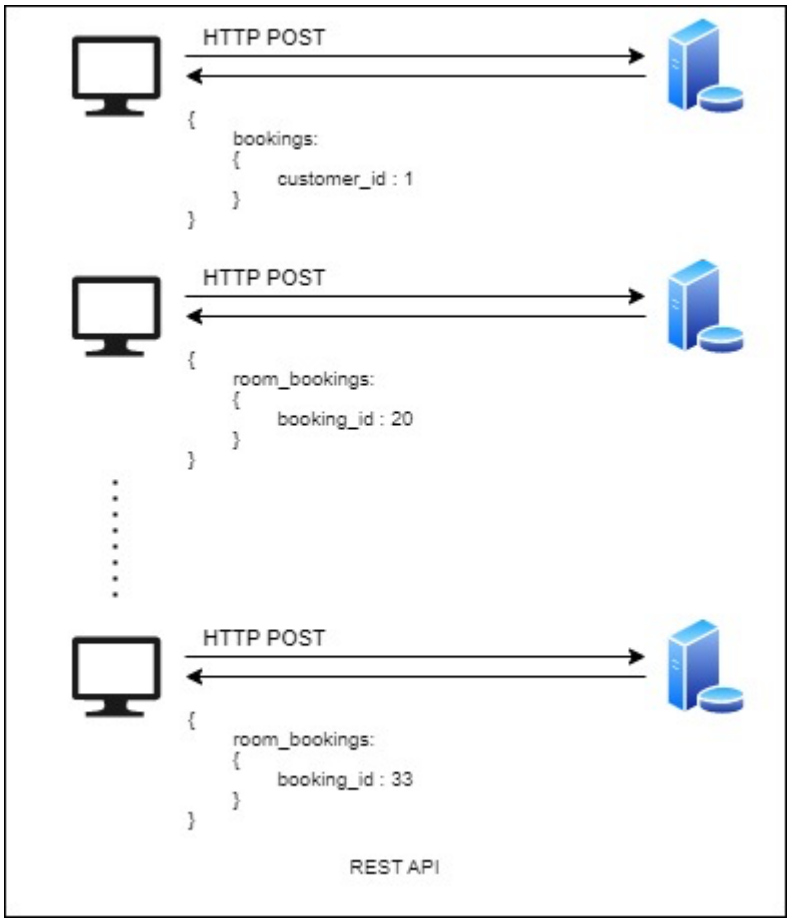
page. Thus, using the CDN would also improve performance by reducing the load on our server by eliminating this redundant computation, thus improving the throughput of our API endpoint.

Many technologies are emerging to support this style of frontend deployment and caching. For example, zero-configuration SaaS providers like Vercel and Netlify handle the complexity of managing a CDN-based deployment. However, if more control is required, the public cloud providers provide infrastructure primitives that can be customised and configured as needed, such as AWS CloudFront.

3.3.2 Pipelining: adopt GraphQL and design a fetching strategy around the most common user flows

REST APIs tend to be more rigid and difficult to change when designing systems with changing requirements. The endpoints defined in REST also cause systems to develop high levels of dependencies between queries. Using GraphQL, which has higher levels of efficiency and flexibility, we could remove the rigidity and dependencies of REST queries and hence lower the latency of the system while putting the burden on the back end and database to respond to complex queries. This is a way to improve the system using pipelining principles.

REST APIs make us use multiple queries to fetch data that might be needed together on a page. By using GraphQL, we can use a single query containing complex data requirements to fetch data. An example of this would be when for a customer, the LANS hotels system needs to load bookings and room bookings within the bookings. The REST structure would first require loading all customer bookings with a request and then using subsequent requests to load room bookings of each booking separately. If this were implemented using GraphQL, the system would send one large request to return all bookings and associated room bookings. This would reduce system latency due to removing the dependency of each room booking query on the result of the booking query. The throughput would depend on the speed and capability of the underlying database to return the results of the single large query.

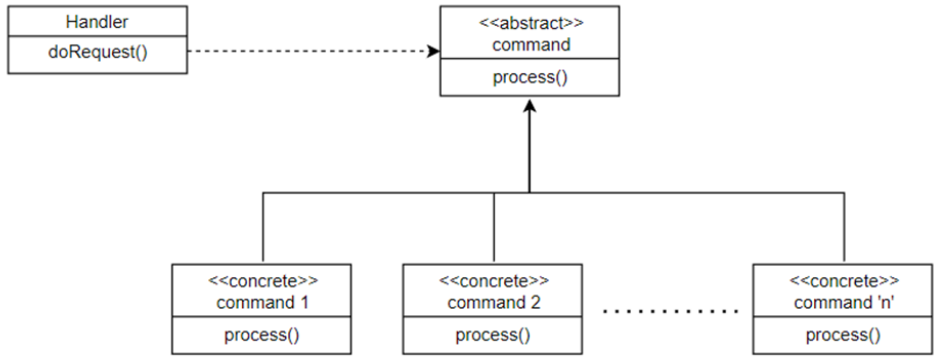


Also, when small changes (data request parameters or required return parameters) in front-end data requirements occur in a system using REST APIs, the back end needs to refactor to account for the change. This also means that a separate endpoint is needed in REST for requests with mostly similar fields but a few different fields or filters. Using GraphQL, these problems could be solved more efficiently as it allows client-side changes to be handled without changing server-side code. An excellent example of this would be when using the search for hotels endpoint in the LANS hotel system. If we were to try and add filtering functionality to send parameters like price range or hotel rating, the back end would need to be modified every time a new filter might be introduced. Using GraphQL, this could be easily handled by the back-end team providing a contract for all available filters and then the front-end team implementing them as and when desired.

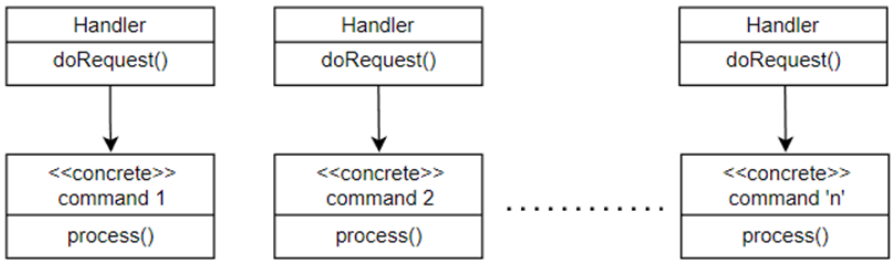
In addition to the above advantages, using GraphQL resolver functions, we can monitor system performance to identify potential bottlenecks that might cause increased latency. These identified bottlenecks could then be engineered to remove design flaws and query dependencies to reduce the latency.

3.3.3 Replacing Front Command with Concrete Controllers

As discussed earlier, Bell's principle suggests a more straightforward design than a more complex one. Our approach from the beginning was to explore the subject solely from a learning perspective and to do the same; our team started with an overly optimistic design. One design feature added was the choice of a front command acting as a layer on top of controllers to help direct traffic from the front-end client service to its appropriate controllers to enhance abstraction to its most significant degree. However, following Bell's principle on performance, a great deal of complexity is encountered when connecting a React-based front end to the front command pattern while ensuring authorisation can be forgiven in exchange for a direct connection to concrete controllers.



The above picture indicates our current design, wherein a centralised control handles all requests to the web application.



The above picture shows an alternative to our approach, removing the abstract command altogether and directly connecting requests with their concrete command processes. While we acknowledge the

first approach to be cleaner with a centralised dispatcher to handle all requests increasing thread safety, the tradeoff is compromising performance by not handling single requests uniquely.

4 CONCLUSION

This report discusses the inclusion, exclusion, and opportunities to apply performance principles to the LANS Hotels application. Although we had some relatively minor instances of using performance principles, our team focused on delivering functionality and did not focus on performance. As such, our team still needs to achieve a high-performance application. However, performance principles can be applied to an existing system with stable functionality through refactoring guided by regression tests. A critical first step towards improving a system's performance is analysing the areas of weakness that present opportunities for improvement. This report discusses the reasons for excluding performance principles in areas of the system's architecture and explores the primary options for advancement if further development were to continue, giving the team a chance to implement a systematic performance strategy through iterative refactoring and redesign.