Product Requirements Document

Part 4 Specification

SCHOOL OF
COMPUTING &
INFORMATION
SYSTEMS

Team Member's name	Student ID	Unimelb Usernames	Github Username	Emails
Zhiming Deng	981607	zhimingd	ZhimingDeng5	zhimingd@student.unimelb.edu.au
Gaoli Yi	1048049	gaoliy	GAOLIY	gaoliy@student.unimelb.edu.au
Lingge Guo	1211499	linggeg1	linggeg1	linggeg1@student.unimelb.edu.au
Chutong Wang	1185305	CHUTONGW	CHUTONGW	CHUTONGW@student.unimelb.edu.au

Release tag: SWEN90007_2021_Part4_<Flying Tiger>

Revision History

Date	Version	Description	Author
26/10/2021	04.00-D01	Initial draft	zhiming
27/10/2021	04.00-D02	Review and the second version of the document	chutong
28/10/2021	04.00-D03	Add Caching	gaoli
29/10/2021	04.00-D04	Add Identity Map	zhiming
30/10/2021	04.00-D05	Add Unit of Work–Lazy Load–DTO and Remote Façade–Optimistic/Pessimistic Offline Lock»Design principles–Bell's Principle–Pipelining –Caching	lingge
01/11/2021	04.00-D06	Add Lazy Load	lingge
02/11/2021	04.00-D07	Add DTO	chutong
03/11/2021	04.00-D08	Add and Remote Façade	gaoli
05/11/2021	04.00-D09	Add Optimistic/Pessimistic Offline Lock	zhiming
05/11/2021	04.00-D10	Add Bell's Principle	lingge
05/11/2021	04.00-D11	Add Pipelining	chutong
05/11/2021	04.00	The final version of the document	gaoli

Heroku App link: https://flying-tiger.herokuapp.com/

Contents

Introduction

1.1 Proposal

This document specifies use cases, actors to be implemented, and the system's domain model of the COVID-19 Vaccine Booking and Management System.

1.2 Target Users

This document is mainly intended for vaccine recipients, health care providers and administrators of the system.

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
COVID-19	Coronavirus disease 2019, which is an infectious disease caused by Severe Acute Respiratory Syndrome coronavirus type 2 (abbreviation SARS-CoV-2).

Actors

Actor	Description
Administrator	The person who manages the COVID-19 Vaccine Booking and Management System.
Vaccine recipients	People who can make a book on vaccination.
Health care providers	People who provide vaccines, add injection timeslots and confirm the vaccination of a Vaccine recipient.

A discussion on patterns performance

3.1 Identity Map

We didn't use Identity Map in our system. When we designed the system, we found this pattern is not necessary for the implementation of features. Therefore we chose not to use this to reduce the complexity.

However, this pattern is helpful to improve the system's performance in terms of response time. In our application, there are some pages for information listing(e.g., vaccine recipients list page, timeslot list page, book list page). Identity Map could save objects that have been loaded once. If using Identity Map combined with the unit of work, pieces of data that have been read from the database once will not be read multiple times if they are clean in the UOW. Those pages will be loaded more quickly when visited for a second time, because much work of loading data from the database has been done in the first time viewing and the system will only load new and dirty objects in the second time viewing.

3.2 Unit of Work

We used UOW in our application. UOW could register objects as new, dirty, clean, and deleted, effectively avoiding unnecessary database updates. For instance, in our application, on the vaccine recipients list page, the administrator can edit/delete an existing vaccine recipient and create a new vaccine recipient. Only the objects that are not clean will be uploaded once to the database when the "confirm change" button is clicked.

The advantage of this is it will need much less time to update the database, especially when handling a mass of data. It requires much fewer requests to the backend to update a large number of data pieces and avoid redundant updates to the database. This could reduce the response time of a request about updating the database. For instance, if the administrator needs to input hundreds of recipients' information into the database, only one request is required instead of sending hundreds of requests and only newly created recipients' information will be input into the database. Thus a large amount of time spent on interaction with the database is saved.

3.3 Lazy Load

We used Lazy Load in our application. Lazy loading can reduce the amount of data read from the database by reading only what is needed. In this project, we use Ghost to implement it. Sometimes, we do not use the question content in the subject questionnaire. Therefore, we can use lazy loading here and only load the problem from the database when the problem is needed and the object is empty. For example, when we want to use the function getQuestions() and a question variable in the questionnaire class is null, it will use *QuestionaireMapper* to load the value of each question. Since we store a string in the question column of the table questionnaire, we need to use string values to generate question objects, and then put them back into the object questionnaire.

The advantage of this is to reduce the amount of data read from the database, while reducing the time to read. Especially when a large amount of refresh and display data is required, the workload of the database will increase. Using Lazy Load can reduce the response time of database update requests.

3.4 DTO and Remote Façade

We did not use DTO and Remote Façade in our system. The direct reason is that this pattern was not included in the initial assignment requirements, and we did not consider this pattern in order to reduce the complexity of the system in the process of system design.

The usage scenarios of DTO and remote facade are when working with a remote interface (e.g., web services), each call to it is expensive. As a result we need to reduce the number of calls, and that means that you should use DTO that aggregates the data to transfer more data with each call. Remote facade is intended for minimizing the number of remote calls。

Using this pattern can reduce the overhead and improve the performance of the system. We will try to use it in future improvements.

3.5 Optimistic/Pessimistic Offline Lock

We used Optimistic/Pessimistic Offline Lock in our application. Using optimistic and pessimistic locks can avoid many concurrency problems, such as lost updates and dirty reads. There are multiple concurrent scenarios in our system. For example, multiple users with the same health care provider account edit the same timeslot's capacity. An user of a healthcare provider edits a timeslot’s capacity and then submit meanwhile another user with the same account does the same operations. The expected output is the update submitted first will be successfully processed, while others failed to process. It avoids losing the update and here we used the Optimistic offline lock.

We also used pessimistic lock, for example, when a healthcare provider is editing the questionnaire about some specific vaccine type, another user that logs in to the same healthcare provider account edit the same questionnaire at the same time.The expected output is that The one who acquires the lock after clicking editing the questionnaire can continue processing, while others with the same account failed to enter the editing page until the one finishes editing by clicking submit or going back.

Using optimistic lock and pessimistic lock avoids the possible concurrency problem of the system, although it will cause some overhead. For concurrent scenarios, such as questionnaire editing, if updating the questionnaire fails, it will cause a great waste of workload, and dirty reading will even lead to unknown errors in the system. Therefore, these two locks provide a great improvement for the security and robustness of the system.

A discussion on design principles performance

4.1 Bell's Principle

Bell's Principle aims to utilize simple designs instead of more complex ones, that is, avoiding adding features and complexity until it's actually needed. In our project, we only implement the functions that we should do on a basis. Also, we didn't add any pessimistic locks or optimistic locks until fixing concurrency problems is actually needed. Any added complexity would affect performance negatively by adding more application latency.

4.2 Pipelining

Pipelining aims to process requests simultaneously instead of processing only one task at the same time to increase the throughput of the application. In our project, we support concurrency and allow requests to be processed parallelizing at the same time. For example, different users of different kinds, different users with the same account can access the website and submit requests almost at the same time, and then the server will process them and return results parallelizing. In this case, throughput and then, performance can increase significantly, and clients need less time waiting for resources.

4.3 Caching

We are not using caching in our system. The purpose of caching is to transparently store data in a faster access location (cache) to reduce the time required to access the data. The advantage of caching is that the waiting time for users to access resources is shorter and relatively simple. But caching also has certain limitations. In our project, if caching is used, it may limit the response speed and cause diminishing returns. The lack of temporal and spatial locality of the cache will slow down the response speed, so we choose not to use it to reduce complexity.