Investigation of System Architecture, Databases, and Caching Systems For Backend Web Development

Independent Study Thesis

Presented in Partial Fulfillment of the Requirements for
the Degree Bachelor of Arts in the
MCS at The College of Wooster

by
Jungho Park

The College of Wooster
2026

Advised by:

Drew Guarnera (Computer Science)

Abstract

Include a short summary of your thesis, including any pertinent results. This section is *not* optional, and the reader should be able to learn the meat of your thesis by reading this (short) section.

This work is dedicated to the future generations of Wooster students.

Acknowledgments

I would like to acknowledge Prof. Lowell Boone in the Physics Department for his suggestions and code.

Vita

Publications

Fields of Study Major field: Major

Minor field: Minor

Specialization: Area of IS research

Contents

Abstract iii

Dedication iv

Acknowledgments v

Vita vi

Contents vii

List of Figures ix

List of Tables x

List of Listings xi

CHAPTER PAGE

1 Introduction 1

2 Backend Software Design and Architecture Principles 3
- 2.1 Monolithic . . . 3
  - 2.1.1 Layered Architecture . . . 4
  - 2.1.2 Modular Monolith . . . 5
- 2.2 Distributed Systems . . . 5
  - 2.2.1 Microservices Architecture . . . 6
  - 2.2.2 Event-Driven Architecture (EDA) . . . 9

3 Databases 11
- 3.1 Types of Databases . . . 11
  - 3.1.1 Relational Databases . . . 11
  - 3.1.2 Non-Relational Databases . . . 12
- 3.2 Database Design and Evolution . . . 12
  - 3.2.1 Normalization . . . 13
  - 3.2.2 Database Evolution . . . 17
    - 3.2.2.1 Replication . . . 18
    - 3.2.2.2 Partitioning . . . 21
- 3.3 Optimization . . . 23
  - 3.3.1 Indexing . . . 23
  - 3.3.2 Query Optimization . . . 26
  - 3.3.3 N+1 problem . . . 26

4 Caching 30
- 4.1 What is Caching . . . 30
- 4.2 Database Caching Strategies . . . 30
- 4.3 Redis . . . 34
- 4.4 Client Side Caching . . . 35
- 4.5 Caching Invalidation . . . 36

5 Implementation 38
5.1 Backend Design 38
5.1.0.1 Software Architecture 39
5.1.0.2 Database Systems 41
5.1.0.3 Caching 47
5.1.1 Backend Request Workflow 53
5.1.2 Modular Architecture Overview 57
5.2 Frontend Design 66
5.2.1 Frontend Architecture 67
5.2.2 API Communication 68
5.2.3 Client-Side Caching 70
5.2.4 User Interface Design Choices 70

6 Results 73

7 Challenges and Lessons Learned 74
7.1 What worked well 74
7.2 What did not work well 74

References 75

Index 77

List of Figures

Figure Page

3.1 Figure of Leader and Follower replication strategy to explain the flow or write or read database querie [10]. 19
3.2 Model of a binary tree to explain datastructures used for database indexing. 25
3.3 Model of a b tree to explain datastructures used for database indexing. 25
3.4 Model of a b+ tree to explain datastructures used for database indexing. 26

4.1 Model of Cache-aside Pattern to explain common database caching strategies [14]. 31
4.2 Model of Read-Through Pattern to explain common database caching strategies [14]. 32
4.3 Model of Write-Through Pattern to explain common database caching strategies [14]. 32
4.4 Model of Write-Back Pattern to explain common database caching strategies [14]. 33
4.5 Model of Write-Around Pattern to explain common database caching strategies [14]. 34

5.1 Figure of the entity relationship diagram for the local cuisine application. 42
5.2 Figure of the backend workflow sequential diagram for the local cuisine application. 71
5.3 Figure of the Java class relationship diagram for the local cuisine application. 72

List of Tables

Table Page

3.1 Order table before 1NF (with repeating groups) 13
3.2 Order table after applying 1NF (no repeating groups) 13
3.3 Student table before applying 2NF (already in 1NF, but with partial dependencies) 15
3.4 Decomposition into 2NF 15
3.5 Student Table (same as 2NF) 16
3.6 Course Table 16
3.7 Decomposition into 3NF (removing transitive dependencies) 18
3.8 Original Students Table with ID, Name, and Grade 23
3.9 Index on Grade: sorted by Grade with references to Original Table 24

List of Listings

Listing Page

5.1 getCuisineByName – fetching a single cuisine by name 68
5.2 createReview – sending a new review to the backend 69

CHAPTER 1

Introduction

My first web application was a Flask application with a simple backend, frontend, a database connected to it, and a few API calls. I thought that web development was simpler than it sounds. However, I changed my mind within a few months of learning the frontend and the backend side of web applications. As I worked on projects, new problems arose as I try to make the scale of the project larger. Also, depending on the project, I had to learn new strategies and technologies to achieve the goal I wanted. More things were there to learn, and I started realizing that several design choices could be made in every situation. Not being able to answer "why this specific choice?" for some of the choices I made realized that I do not have a full understanding of the decisions that could be made in web development, and led me to research this topic.

Covering all the topics in web development was the primary goal, but it could be a lot to process a deep understanding in one year of the Independent Study period. I narrowed down my subjects to target the topics I encountered the most and personal interests, with the help of the backend roadmap [1]. Software architecture choices were a personal choice of interest. Although most of my applications were monolithic due to a smaller scale, I wanted to dig deeper into the structures of monolithic applications and also distributed systems, so that I could apply the knowledge when I start to work with larger-scale applications.

The second topic is database systems. I have taken the database class during my sophomore year, but web development required much deeper knowledge. Types of databases to choose, ways to structure the database for performance and scalability were always a confusion for me. To fully understand the decisions I am making, I researched this topic to be able to make logical decisions in the future when creating applications.

The last topic is about caching systems. There were multiple times in my past projects where the application was slow due to database queries or functions to fetch data from external APIs. Caching is one of the solutions to solve this problem, and it will be discussed throughout this project. Caching data that calls database queries or API calls often makes the application significantly faster, and I want to investigate which strategies and types of caching systems are sufficient in every scenario.

A web application that uses these three technologies is the best way to show that I have a full understanding of the choices I have made. I developed a local cuisine recommender web application to show the knowledge and research done in these areas. The motivation for creating an application in this domain arises from the discomfort of finding local cuisines when traveling to foreign countries. It is a good experience to try out the local cuisines of the destination area, especially for people who love food. However, searching directly on Google Maps does not accurately show the famous dishes in the specific region, but rather recommends restaurants. Going straight to the restaurants sometimes works, but sometimes it is helpful if there is a set of cuisines to try, already layout and choose the restaurant where the cuisines are available. This needs a two-step search: first on a search engine such as Google to find the famous food in the area, and second, searching the name of the dish on Google Maps. The purpose of the Local Cuisine application is to make the process of searching restaurants more efficient.

This research article will be written in the following structure: Chapters 2, 3, and 4 will investigate the theories of topics that I have chosen, which will include backend software design and architecture, database systems, and caching systems, respectively. Each chapter of the theory will thoroughly research the design choices that could be made within each topic and how to make decisions based on each situation. Chapter 5 will discuss how the local cuisine web application is implemented, the design choices that are made are based on the theory introduced in the previous chapters. The outcome and explanation of the final software will be addressed in chapter 6, including snapshots and images of the web application. Finally, the research article will conclude with the challenges throughout the whole research project and the lessons learned in chapter 7.

CHAPTER 2

Backend Software Design and Architecture Principles

There are multiple architectural styles that could be applied when designing the backend system of a web application. The two main categories are monolithic and distributed. Subcategories such as layered services, microservices, event-driven services, are all under these two architecture styles. Each of these design have their advantages and disadvantages.

2.1 Monolithic

A monolithic application is where all the logical components of the application is deployed as one unit, which also means that it will be run in one process. For example, if there is a stand-alone Python Flask application deployed to a server, this is monolithic application. This means that the UI components should also be included in the same unit. If there is a separate frontend server, this is not a monolithic application anymore. There are multiple reasons to build a project with the monolithic architecture.

One of the benefits of the monolithic architecture is its simplicity. Since a typical monolithic application has a single codebase, it is easier for the developers and collaborators to understand the overall structure. Even when a developer joins in the middle of the development process or reading an older code base, understanding program will be generally easier compared to distributed systems structure. Another advantage of using the monolithic structure is the cost. Since only one unit has to be deployed, which is simple and requires less infrastructure, it is cheaper to operate overall. Because of the cheap cost, developers are able to experiment and deliver systems faster, which is beneficial in a fast pace environment. Reliability is another aspect, where monolithic applications make few or no network calls, which usually means more reliable application. Lastly, it is easier to find bugs compared to distributed systems since all the code is in one place.

On the other hand, there are also disadvantages of using the monolithic architecture. As the monolithic application grows, the application becomes harder to scale. Since all the code is in one codebase, it is not possible to adapt different technology stacks to different domains if needed. This also connects to decrease in reliability. A single bug or error occurred in the application could degrade the service and affect the whole application. Lastly, when deploying the application, implementing any change will require redeploying the whole application, which could introduce a lot of risk of having reduced uptime of the application. [6]

2.1.1 Layered Architecture

Layered architecture is a sub category of the Monolithic architecture. Applications that are designed with this architecture have three layered parts: the presentation layer, the workflow layer, the persistence layer, and the database layer. Each layer has its own tasks and separated within one application.

- **Presentation Layer:** The presentation layer is where the user interface (UI) is displayed and where the users interact with the system. All components that are related to the UI are included in this layer.
- **Workflow Layer:** The workflow layer consists of all code related to logic, such as business logic, workflows, and validations. This is where most of the application's code is contained.
- **Persistence Layer:** The persistence layer encapsulates the behavior required to make objects persistent, such as mapping the architecture or code-base hierarchies into set-based relational databases.
- **Database Layer:** The database layer is optional. This layer includes the database or any mechanism used to persist information.

Each layer may contain multiple problem domains. Let's take an application for a restaurant as an example. There could be domains such as place order, deliver order, manage recipes, or mange inventory. Different problem domains could exist together in each layer of the application.

An implication of the layered monolithic architecture is the MVC design pattern, which is an abbreviation for model view and the controller. In MVC, the model represents business logic and entities in the application; the view represents the user interface; and the controller handles the workflow, stitching model elements together to provide the application's functionality.

The layered architecture has several advantages. One of them is the feasibility of the application. Since the structure is organized into distinct layers with specific responsibilities, it is clear for developers to understand the overall structure of the code. This separation also allows technical partitioning, which makes it much easier to manage and maintain big projects. Furthermore, the layered architecture is quick to build particularly among the projects that have clear relationships among the components, as different development teams can work on different layers in parallel.

There are also disadvantages to consider. The layer could add complexity to the application compared to a simple monolithic application. Building a large application using layered monolithic architecture could end up in a confusing relationship between layers and result in challenges maintaining and scale the application. However, this would still be less complicated compared to the complexity of distributed systems discussed in section 2.2. Testing is another problem since most layers are dependent on each other, testing a specific layer requires mock data or simulation of the related layers, which could decrease the quality of the testing.

Some applications using the layered architecture may have different structure from the original presentation, workflow, persistence, and database layer. Some layers might be separated, such as the presentation layer being separated from the other layers. Each structure will have advantages and disadvantages within the layered architecture.

2.1.2 Modular Monolith

Modular monolith is when an application is divided into different modules within the monolithic system. A module is one domain of the application, and each domain will have its independent code base. For instance, an application for a restaurant might have different domains such as orders, deliveries, or recipes. In this case, there could be separate databases for each module, or multiple schemas within a single database. It is important to keep in mind that only the code base is independent in the software structure, meaning that since modular monolithics still follows the monolithic architecture, the code will be compiled together as a single application and run in one process.

One of the benefits of using the modular monolithic structure is the domain partitioning. Similar to the layered architecture, this allows to have separate development teams that specializes in one or more of these domains, which will increase the efficiency of the work. Similar to other monolithic applications, the performance of the application will be fast since there are no network calls within the application. Furthermore, since all modules are separated from each other, it is easier to maintain the code.

However, code could be hard to reuse since each module have its own code. The logic and utilities could be tightly coupled to each domain, decreasing the flexibility to reuse the code across modules compared to the modular monolithic structure [6]. Also, similar to other modular monolithic architecture applications, as the applications grows, the logics and the databases could become complicated since all the code base is in one single application.

2.2 Distributed Systems

Distributed architecture is a design approach where the logical components of an application are divided into multiple units. These units each run in their individual process and communicate with each other over the network. This architecture style encourages loose coupling of each service, which could be a huge benefit.

There are multiple advantages to choosing distributed systems. The one that contrasts with the monolithic architecture is scalability. Since distributed architectures deploy different logical components separately from one another, it is easy to add new services. Furthermore, similar to modular monolithics, distributed architecture encourages a high degree of modularity. This is because distributed systems have loosely coupled logical components. The benefit of having multiple distributed systems is also shown during the testing phase as well. Since each unit is loosely coupled, they can be tested separately. This becomes a huge advantage as the size of the application grows compared to monolithic systems. Lastly, fault tolerance is a big difference compared to the monolithic system as multiple units are deployed in several units. Even though one part of the system is down, the other parts of the system could still be available depending on the coupling.

On the other hand, there are also disadvantages when choosing to build a distributed systems architecture. Performance is generally slower than a monolithic application since distributed system architectures involve lots of small services that communicate with each other over the network. This can affect performance, but there are ways to improve this. Another aspect is cost, since more servers are needed to deploy multiple units. These services will need to talk to each other, which entails setting up and maintaining network infrastructure. This leads to more complexity in the system, and makes it more challenging for the developers harder to understand the overall design. Also, errors could happen in any unit involved in servicing a request. Since logical components are deployed in separate units, tracing errors could become complicated.

There are both advantages and disadvantages of monolithic and distributed systems, and choosing the appropriate architecture for each situation is crucial. After deciding which of the two main architecture will be used, then we could decide on which subcategory of either of the architectures to implement. Each types of subcategories, have their own advantages and disadvantages.

2.2.1 Microservices Architecture

A microservice is a service that is a separately deployed unit of software that performs some business or infrastructure process [6]. Microservices architecture is part of the layered system, where microservices communicate with each other to make an application. Since the system is divided into multiple parts, it is essential when making decisions how to divide the application, such as deciding how small or how big each microservices could get. Generally, it is better to make microservices smaller for multiple reasons. Some of the factors that are considered to make microservices smaller are the following: cohesiveness, fault tolerance, access control, code volatility, and scalability.

If a part of the software has lack of cohesiveness and loosely coupled, it is a good idea to separate it into smaller microservices. Here, cohesiveness means that how close the parts of the application are related to each other [6]. This will allow higher scalability, since separated microservices could be built independently based on the workload and resource requirements. If a certain part of the application produces fatal errors, having those parts in a separate microservice will decrease the probability of shutting down the whole system. For security and authentication, it is important to have these access controls in a single microservice, so it does not get too complicated when managing the information. Finally, if one part of the microservice change, or scale faster than the others, it is good to consider having a separate microservice for that application, since testing the entire microservice would be much more challenging compared to a small portion of the microservice.

On the other hand, there are times when it is encouraged to make the microservices bigger: Database transactions, data dependencies, and workflow. It is not possible to perform a single database commit or rollback when a request involves multiple microservices. For data consistency and integrity, it is important to combine functionalities that require this kind of behavior into a single microservice. If a part of a microservice has highly coupled data, such as when a database table refers to the key of another database table, it is better to keep these functionalities as a single microservice to keep the data integrity of the database. Furthermore, if a single request requires separate microservices to communicate with each other, this request is coupled and recommended to kept in the same service.

Workflows

After setting up multiple microservices, the next question is how these microservices communicate with each other to make the whole application. This is a crucial part of the microservices architecture, and this is where workflow comes in. The term workflow means when two or more microservices are called for a single request. Workflows are in charge of navigating which microservice to start, which one to call next, and which one to end with. There are two ways to handle workflows: centralized workflow management and decentralized workflow management.

The first type of workflow management style is the centralized workflow. This workflow management style is where there is a microservice that coordinates all the microservices that are needed to handle a certain single request. This microservice will be responsible for calling all related microservices, knowing the current state of the workflow and what happens next, summarize all data from each microservice, and handle errors.

The advantage of this workflow management style is that the order of microservices for each request is clear. The central microservice always has the exact routes the request has to take, which allows to track the status of where all requests stopped and where to restart. This allows more efficient error handling and simplifies the process of changing a workflow.

The disadvantage is the tight coupling between the central microservice and the other microservices. This can lead to lower scalability, since changing a microservice will affect the central microservice. Moreover, performance might get delayed since the central microservice is indeed another microservice that requires remote calls, and it saves the workflow state data in a database, which slows down the performance.

The next type of workflow management style is the decentralized workflow. This workflow does not have a central microservice, but rather all microservices redirect to another microservice by the given request until the request is complete. It is important not to use one of the microservices as a central microservice.

The advantage of using this workflow pattern is that it is loosely coupled compared to the central workflow management, which has better scalability. Also, since the microservices do not have to connect back to the central microservice, it has better responsiveness, meaning less delay and better performance.

On the other hand, this pattern lacks in error handling, since each microservice is responsible for managing the error workflow, which could lead to too much communication between services. Also, because there is no central microservice that has the order of microservices that is need to be called, it is hard to tell which state the response is at, which decreases the ability to recover when the request is delayed and needs to be restarted.

Sharing functionalities

There are many times when the same code has to be used in multiple microservices. There are mainly two ways for sharing code when building a microservice architecture. Creating a shared service or a shared library. A shared service is a separate microservice that contains a shared functionality that other microservices can call remotely. One advantage of the shared service is that even though code may change in the shared microservices, code in other microservices are not needed to be changed. Also, this shared service could be written in any language, which is useful when microservices are implemented in multiple languages. A disadvantage of using a shared service is the coupling that happens between the microservices and the shared service. This leads to risks when changing a shared service since it can affect the other microservices that call it. Furthermore, when the shared service is down for some reason, the microservices that require the shared service would not function.

A shared library is a more common way for code reuse. A library will be built, including all the code that is reused in different microservices. Once the microservices are deployed, each microservice will have all the shared functionality available. The biggest advantage of using a shared library is that network performance and scalability are better than shared service, since it is not remote, and the code is included in the compile time of the microservice. However, multiple shared libraries will be needed if microservices are written in different programming languages. Also, managing dependencies between microservices and shared libraries could be a challenge if there are multiple microservices using the shared code.

Pros and Cons of Microservice Architecture.

Microservice architectures are commonly used as they have several advantages. Loose coupling is one of them. Each microservice is single-purpose, and they are deployed separately, which makes them easy to maintain. This means that it is easy to change a particular microservice that needs modification, instead of making multiple changes in different parts of the whole application. Also, it is easy to test the application, as the scope is much smaller compared to the monolithic architecture. Fault tolerance is another advantage that comes from loose coupling. Even if a particular microservice fails, it does not break the whole system. Furthermore, microservice architecture is easy to scale and evolve since we just have to add or modify a microservice related to the area.

However, there are also disadvantages when using the microservice architecture. This architecture is complex.

It requires multiple decisions, such as the aspects discussed above (workflow, shared code etc.) depending on the situation. Because of its complexity, as microservices communicate with each other, the performance also decreases. The request might have to wait for the network or undergo additional security checks and make extra database calls. Lastly, deploying all of these microservices will increase the cost of the entire application.

2.2.2 Event-Driven Architecture (EDA)

What is an Event

An event in computer science is a way for a service to let the rest of the system know that something important has just happened. In EDA, events are the means of passing information to other services [6] . The event contains data, and it is broadcast to services using topics that are connected to the sender of the event. However, events are asynchronous, which means that although the services are connected, the sender of the event does not wait for the response of the receiving service. This is the key difference between an event and a message. Messages include a command or some kind of request for the receiving service, and the service sending out the messages requires a response, making it synchronous. Messages are also sent to only a single service using queues.

Asynchronous Communication

EDA relies on asynchronous communication when sending and receiving events. This is because events in the EDA do not wait for a response and is built for loose coupling. Asynchronous communications do not need to wait for a response, even though the receiving services are available or not. These kinds of communications are also called Fire and Forget. On the other hand, synchronous communication needs to stop and wait for the response, which means that the service in response must be available. This make tighter coupling between services, making it not suitable for EDA.

Pros and Cons of EDA

Asynchronous communications have advantages in responsiveness compared to synchronous communications. Since responses by the receiving services are unnecessary, it takes less time to complete a request. However, this is also a crucial disadvantage. Since we do not know if the receiving services have successfully completed the request, it is prone to error handling.

Event-Driven Architecture is highly decoupled, which makes all services independent and easy to maintain. Furthermore, asynchronous communication increases the performance of the application. Since EDAs are highly decoupled with this type of communication, it is easy to scale and evolve.

On the other hand, there are also disadvantages that need to be considered. Similar to microservices, EDAs are complex. Deciding which database topology for the architecture, asynchronous communications and parallel event processing add complexity to the application. Also, asynchronous communication makes it more difficult to test out the program. Since the request is processed without any response or synchronous calls, the context of the test is vague. If the application needs multiple synchronous calls, EDA is not the right choice for the product.

CHAPTER 3

Databases

3.1 Types of Databases

There are mainly two types of database that are used in software engineering: Relational database and non-relational databases. These two databases defer by how they store and data, which influences different aspects if databases including the structure, data integrity mechanism, performance and more.

3.1.1 Relational Databases

Relational Databases store data in tables by columns and rows, where each column represents a specific data attribute, and each row represents an instance of that data [12]. Each table must have a primary key, which is an identifier column that identifies the data uniquely. The primary keys are used to establish relationships between tables, by using the related rows between tables as the foreign key in another table. Once two tables are connected, it is now possible to get data from both tables in a single SQL query.

Advantages and Disadvantages

Relational databases follow a strict structure, which allows users to process complex queries on structured data while maintaining data integrity and consistency. The strict structure also follows the ACID (atomic, consistency, isolation, and durability) properties. Atomicity keeps all steps in a single database transaction are either fully completed or reverted to their original state. Consistency keeps all data meet the predefined integrity constraints and business rules. Even if multiple users perform similar operations simultaneously, data will remain consistent for all. Isolation ensures new transactions accessing a particular record waits until the previous transaction finishes before it is executed. This will allow concurrent transactions to not interfere with each other. Finally, durability makes sure that the database maintains all committed records, even when there is a system failure. These four ACID properties help enhanced data integrity, but because of the rigid structure, it is hard to scale compared to nonrelational database [12].

3.1.2 Non-Relational Databases

Nonrelational databases means that there isn't a schema to manage and store data. Schema is a blueprint of a database describes the relationship between data, tables, or other datamodules. In nonrelational databases, data does not require constraints that a relational database requires, such as fixed schema, primary key constraint, or not null constraints. This allows more flexibility in the structure of the database, or anything that may change in the future. There are different types of nonrelational databases: Key-Value databases, Document database, and Graph database. Key-Value database store data as a collection of key-value pair, where the key is served as the unique identifier. Both the key and values could be anything as objects or complex compound objects. Document databases are used to store data as JSON objects. Since it is readable by both human and the machine, it has the ease of development. Lastly, there are graph databases, where they are used when a graph-like relationships are needed. Unlike the relational databases, which store data in rigid schema, graph databases store data as a network of entities and relationships, providing more flexibility to anything that is prone to change.

Advantages and Disadvantages

Nonrelational databases have a less rigid structure allowing more flexibility. This is useful when the data changes often based on requirements. For example, when a specific table needs to change or add columns, this would be hard to preform because other tables might be associated with the specific column. However, nonrelational databases are not constraint to fixed schema, which allows easy changes on specific columns or unique identifier. Performance is another strength of nonrelational database. The performance of these databases depend on other factors such as network latency, hardware cluster size, which is different from relational databases which depends on internal factors such as the structure of the schema. However, because of the flexibility in the structure, data integrity is not always maintained. Consistency is also an issue since the state of the database changes over time as structures might not be consistent.

3.2 Database Design and Evolution

The design of databases and querying the database are the ones that have the most impact on the performance of the database. This section investigates strategies to design the database to increase performance and efficiency, consider scalability, evolution of databases, and ways to optimize the database and queries.

3.2.1 Normalization

Compared to a non-relational database, relational databases benefit from integrity with the process of normalization. The definition of normalization is a way of organizing data in a database. During this process, redundant data will be reduced to improve data quality and optimize database performance. There are several levels of normalization, and as the normalization form gets higher, it generally indicates a more refined relational design. However, most databases tend to be needed until the third normal form, which avoids most of the problems common to bad relational designs [8].

The First Normal Form (1NF)

The two criteria that 1NF meet are the following:

1. The data are stored in a two-dimensional table.
2. There are no repeating groups.

Here, the repeating groups mean that if an attribute has more than one value in each column of a row [8]. For instance, if there is a column that requires more than one value, such as items (instead of item), this will be a repeating group.

Table 3.1: Order table before 1NF (with repeating groups)

Student_ID	**Student_Name**	**Course_ID**
S001	Alice	C101, C102
S002	Bob	C101, C103
S003	Charlie	C101, C102, C103

This table shows the example of a table before the first normalized form. We can see that students have multiple classes in the same column separated by a comma. Having three or more dimensions or having repeated groups for a single table results in more complexity and difficulties when querying the database. This is why we need the 1NF normalization. The 1NF is the simplest normalization that could be done in a relational database, and it is pretty clear.

Table 3.2: Order table after applying 1NF (no repeating groups)

Student_ID	**Student_Name**	**Course_ID**
S001	Alice	C101
S001	Alice	C102
S002	Bob	C101
S002	Bob	C103
S003	Charlie	C101
S003	Charlie	C102
S003	Charlie	C103

As we can see on the table above, there is only one course in the course column by creating a new record for each item. However, the first normalization form is not enough to benefit from using relational databases.

Problems with 1NF

Data could become redundant even though repeated rows are deleted. For example, if there is a table called students with columns(id, name, birthday, course id, course name), every time the same student adds a class, their name will be repeated in each record. Furthermore, there are anomalies in update, insertion, and deletion. Making an update to one of the records (student name) would require updating other records that are associated. Missing any of the records with the student name will result in data inconsistency. Inserting a new entity could be difficult unless there is related data. For instance, adding a new course to the table is not possible until there is a student taking the course. This is because the primary key will be the student's id, and there could not be a row without a primary key. Deletion could also be a problem when the row deleted contained the last data for a certain column. For example, if one student is taking a class called CS200, and that record is deleted, the table would not have the information that the class CS200 exists. To solve these problems, we must use higher levels of the normalization form.

The Second Normal Form(2NF)

The two criteria that 2NF meet are the following:

1. The relation is in first normal form
2. All non-key attributes are functionally dependent on the entire primary key.

Functional dependency is the key term in the 2NF. A functional dependency is a one-way relationship between two attributes, such that at any given time, for each unique value of attribute A, only one value of attribute B is associated with it throughout the relation [8]. In other words, this means that all other columns except the columns of the primary key or keys, are dependent on the primary key or the candidate key.

By using the functional dependencies, we could create the second normal form relations. After analyzing the functional dependencies, primary keys would be decided. It is common to decide on attributes that have dependencies for the primary key of the table, and the all the other attributes will be the non-key attributes. For example, going back to the student table example, the Student name and birthday would be dependent on the student id, so the student id will be the primary key. The courses the student takes do not depend on the student id, but rather on the course id. We could create another table using the course id as the primary key and the course name as the functional dependencies. However, the courses the student takes will be dependent on the courses, so there will be a foreign key to construct that relationship. So far, we can identify the relationship of the tables as the following:

- **Student**(*student_id* (PK), *course_id* (FK), *name*, *birthday*, *room_num*)

- **Courses**(*course_id* (PK), *name*)

We are assuming that each student can only take one course.

The relationship can be represented as:

$$\text{Student(course\_id)} \rightarrow \text{Courses(course\_id)}$$

Table 3.3: Student table before applying 2NF (already in 1NF, but with partial dependencies)

Student_ID	**Student_Name**	**Birthday**	**Course_ID**	**Course_Name**
S001	Alice	2002-03-14	C101	Database Systems
S002	Bob	2001-06-02	C102	Data Structures
S003	Charlie	2002-07-22	C101	Database Systems
S004	Diana	2003-01-18	C103	Operating Systems

Table 3.4: Decomposition into 2NF

Student Table			
Student_ID (PK)	**Student_Name**	**Birthday**	**Course_ID (FK)**
S001	Alice	2002-03-14	C101
S002	Bob	2001-06-02	C102
S003	Charlie	2002-07-22	C101
S004	Diana	2003-01-18	C103

Courses Table	
Course_ID (PK)	**Course_Name**
C101	Database Systems
C102	Data Structures
C103	Operating Systems

By doing this, some problems from the first normal form are solved. The functional dependency solves the insertion, deletion, and update anomalies. We can now insert a new course into the course table without needing to insert a student at the same time, delete a student's course without losing the record of the course itself, and update course information in one place instead of multiple rows, which preserves data integrity. This will provide a more stable and consistent database design compared to 1NF.

Problems with 2NF

Although some of the problems with 1NF were solved, there are still anomalies to be resolved. Insertion, deletion, and update anomalies still exist. For example, let us assume we have the following dependencies and the table:

- **Student**(*student_id* (PK), *course_id* (FK), *name*, *birthday*, *room_num*)

$$\text{Student}(\text{course\_id}) \rightarrow \text{Course}(\text{course\_id}, \text{name}, \text{room\_num}(FK))$$

- **Course**(*course_id* (PK), *name*, *room_num* (FK))

$$\text{Course}(\text{room\_num}) \rightarrow \text{Room Number}(\text{room\_num}(PK), \text{name})$$

- **Room Number**(*room_num* (PK), *name*)

Table 3.5: Student Table (same as 2NF)

Student_ID (PK)	**Name**	**Birthday**	**Course_ID (FK)**	**Room_Name**
S001	Alice	2002-03-14	C101	Database Room
S002	Bob	2001-06-02	C102	CS Lab
S003	Charlie	2002-07-22	C101	Database Room
S004	Diana	2003-01-18	C103	Systems Lab

Table 3.6: Course Table

Course_ID (PK)	**Course_Name**	**Room_Name**
C101	Database Systems	Database Room
C102	Data Structures	CS Lab
C103	Operating Systems	Systems Lab

Even though the tables are in 2NF, insertion, deletion, and update anomalies can still occur due to these transitive dependencies.

The functional dependency Course → Room Number introduces anomalies that still remain in 2NF. For example, we cannot record a student's enrollment in a course if the course's room number has not been decided, which creates an insertion anomaly. Also, if we delete the last student enrolled in a course, we also lose the record of the room number for that course, which is a deletion anomaly. Finally, if a course's room number changes, we must update it in every student's record for that course, creating an update anomaly. These problems occur because the room number depends on the course (a non-key attribute) rather than directly on the student ID, and they are resolved by moving the design into 3NF.

The Third Normalization Form(3NF)

The two criteria that 3NF meet are the following:

1. The relationship is in second normal form.

2. There are no transitive dependencies.

Transitive dependencies

Transitive dependencies exist when the following functional dependency pattern occurs:

$$A \rightarrow B \text{ and } B \rightarrow C, \text{ therefore } A \rightarrow C$$

This is the same type of relationship where we had problems with 2NF. For example:

- **Student**(*student_id* (PK), *course_id* (FK), *name*, *birthday*, *room_num*)

$$\text{Student}(\text{course\_id}) \rightarrow \text{Course}(\text{course\_id, name, room\_num}(FK))$$

- **Course**(*course_id* (PK), *name*, *room_num* (FK))

$$\text{Course}(\text{room\_num}) \rightarrow \text{Room Number}(\text{room\_num}(PK), \text{name})$$

Going back to this example, we can see that a non-key attribute in the **Student** table (*room_num*) depends on another non-key attribute (*course_id*).

To remove transitive dependencies, we should break the relations into separate tables. In this case, we can:

- Remove the *room_num* column from the **Student** table to remove the relationship between non-key attributes.

- Alternatively, make the second determinant in a table a candidate key so that no non-key attribute depends on another non-key attribute within the same table.

Applying either method will resolve the anomalies we encountered in the second normalization form (2NF).

3.2.2 Database Evolution

Database design also influences how a database evolves over time. Performance, latency, and fault tolerance are affected by the database designs as the amount of stored data increases. There are multiple techniques to design a database suitable for scaling. Vertical scaling refers to when the physical parts of the computer such as CPU or RAM are upgraded, so the database can handle more data and queries. Horizontal scaling is when multiple machines with independent physical parts (CPU and RAM), store the data on an array of disks that are shared among multiple other machines, which is also called nodes. One advantage of using vertical scaling is that the structure is very simple. Only

Table 3.7: Decomposition into 3NF (removing transitive dependencies)

Student Table

Student_ID (PK)	Name	Birthday	Course_ID (FK)
S001	Alice	2002-03-14	C101
S002	Bob	2001-06-02	C102
S003	Charlie	2002-07-22	C101
S004	Diana	2003-01-18	C103

Course Table

Course_ID (PK)	Course_Name	Room_Num (FK)
C101	Database Systems	R12
C102	Data Structures	R14
C103	Operating Systems	R15

Room Table

Room_Num (PK)	Room_Name
R12	Database Room
R14	CS Lab
R15	Systems Lab

one database needs to be taken care of, which could be easier compared to handling multiple databases. However, there are more disadvantages. Cost increase faster than linearly as higher performance is required. Fault tolerance is problematic as there is only one database running, hardware or software failures on the database can cause service interruptions. For horizontal scaling, there are mainly two strategies that we will discuss: replication and partitioning. These two strategies have its own sub strategies which will be discussed in the next subsection.

3.2.2.1 Replication

Replication is when there are multiple copies of the database in different nodes, which could be in different virtual, or physical machines. This leads to redundancy in data, but it also allows for fault tolerance if one of the database hosts is unavailable [10]. Another reason to use replication is to increase availability by having databases in different regions to decrease latency. Having multiple database to read queries also help increase the performance of the application as the size of the database grows.

There are multiple ways to implement this strategy in different situations. The two topics discussed will be Leader and Followers and Multi Leader Replication.

Leaders and Followers

Going deeper into how the replicas are established, it is important to ensure that each of the replicas have the same data. For example when a user updates a table, this should include all the other replicas to update the same table. Here is an example how the leader follower replication setup could look like:

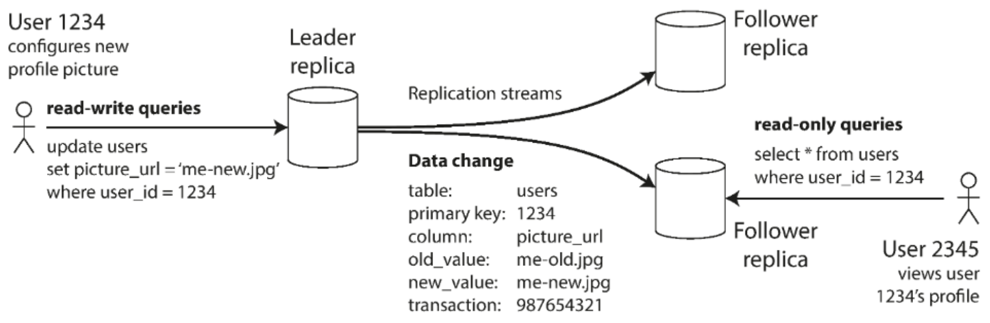

Figure 3.1: Figure of Leader and Follower replication strategy to explain the flow or write or read database querie [10].

The leader and follower structure has one lead node (leader) that allows writes to the database, and the rest of the nodes are assigned as followers. If there is a change data in the leader, the followers are updated accordingly. Reading the database could be done by any of the nodes including the leader node, but writing the database could only be done by through the leader node.

Multi-Leader Replication

One downside of a Leader and Follower replication was that write queries were only allowed in the leader replica. If the application could not connect to the leader for any reason such as network issues, write queries could not be performed. Multi-Leader replication allows multiple leaders, which means that write queries could be done in multiple replicas. Although it is rare for a multi-leader replication structure to outweigh the benefits against the complexity added, there are a few use cases for this structure.

- Multi datacenter operation
- Clients with offline operation
- Collaborative editing

In situations where there are multiple datacenter operation, multi-leader replication allows each datacenter to process write requests locally, which will reduce latency and improve availability during datacenter or network failures. Also, for applications that are necessary to support offline operation, each device can function as an independent leader, enabling uninterrupted read and write access when disconnected. If this were to be a single-leader replication, clients would not be able to perform write operations offline, limiting usability in environments with low network connectivity. Finally, in collaborative editing systems, multi-leader replication enables multiple users to make concurrent updates with low latency. If using a single-leader replication, all edits would need to pass through the leader, and clients would need to wait for others to finish their changes. This will increase latency and reduce the responsiveness of real-time collaboration [10].

However, the biggest issue in the multi-leader replication structure is handling multiple write query conflict. For instance, if a same record was modified by multiple users on different leader replicas, there will be different data among replicas resulting in data inconsistency. This was not an issue for a single leader and followers structure since only one leader accepted write queries, and they were processed one by one. However, if multiple leaders receive queries at the same time, the conflict should be resolved. There are multiple ways to handle write conflicts.

The simplest strategy is to just avoid the conflicts. If write queries could be avoided in the application layer, where the application ensures that all write queries for a particular record go through the same leader. This would work until a particular records needs to change its designated leader. In this case, simply avoiding conflicts in the application layer would not work

The next strategy is to converge toward a consistent state. When there is a situation where avoiding conflicts do not work, the conflicts should be handled. There are various ways to converge the replicas into a consistent state. One way could be to simply give each write query a unique ID, and if there is a conflict, the higher unique ID overwrites the conflict. Another way is to give each replica a unique ID, and the higher unique ID overwrites the conflict. Or, there could be another explicit data structure that preserves the information and write application code to resolve the conflict in a certain logic. There are trade-offs such as which part of the data is going to be saved over the other, so it is essential to choose the write way to handle write conflicts in a multi-leader replica structure.

Communication between replicas: Synchronous vs Asynchronous Replication

Communication that happens between the leaders and the followers is also another important factor to decide whether it is a leader and follower setup or multi-leader setup. The two types of communication are synchronous and asynchronous. After the leader receives a write query, synchronous replication waits until all the followers are updated and synced. The leader will not take any queries before this process is finished. However, asynchronous replication keeps taking queries regardless of the status of the followers. Although handling requests to copy status of the leader on the database replicas are fast, which usually happens less than a second [10], there is no guarantee of how long it would take depending on the physical status of the replica and the network. This is an advantage of synchronous replication, since it is certain that all database replications are up-to-date with the leader database, but it could be a disadvantage in a perspective where the database queries will be slow. If all of the followers are synchronous, the system would be too slow. Most of the times, one of the followers will be set to being synchronous so that there will be at least two replications(one leader, one follower) up-to-date with all the write queries, and the rest of the followers being asynchronous. This configuration is sometimes also called semi-synchronous.(Designing Data-Intensive App ch5)

Asynchronous replications on the other hand are very fast since the leader does not have to wait until the followers have synced data. However, it does have a disadvantage that all of the followers might not be up-to-date right away, which is also known as the replication lag. This might sound like a huge disadvantage, but asynchronous replication is often times used if there are many followers or if they are geographically distributed. Eventually, all the followers will have a synced replication of the leader, since inconsistency is just a temporary state. This effect is called as eventual consistency [10]. If the application that is being created is okay with possibility of inconsistency due to replication lag, asynchronous replication is perfect. However, if consistency is crucial to the application, other solutions such as synchronous or semi-synchronous replication will be the go-to when using leaders and followers method.

3.2.2.2 Partitioning

Partitioning is when a data is divided into multiple smaller pieces of data such as a record, a row, or a document. Replication was used for scalability and performance, but as the size of the database increases, only replication itself become inefficient since all the data has to be copied to the replicas. In order to scale database applications, partitioning is commonly used. As the data is distributed among multiple partitions, multiple queries could be run at the same time.

Partitioning by Key-Value Data

Having a key-value data is one way to create partitions. The reason for partitioning is to scale the database by distributing the query load on one node. One way to achieve this is to have a key range for each partition. For example, when we think of a dictionary, we know that words that start with the letter “a” is in the beginning, the letter “m” is somewhere in the middle, and the letter “z” is at the end. By dividing the range of partitions by the key-value data, it will be easier to locate which partition contains the data and distribute the query. In this case, the partitions might not be evenly divided since each range of key-value pairs could contain a different amount of data. It is important which key-value pair to use. If a key is assigned as a column that is not evenly distributed, this leads to a hot spot (a partition with disproportionately high load) [10].

A method that has a lower chance of being affected by hot spots is partitioning by hash of key. A good hash function makes the key-value pairs evenly distributed among the partitions. However, this loses the efficiency that the key range had, since keys that are close in value may be distributed across different partitions. This leads to range queries to access multiple partitions and merge their results, which will increase query latency.

When choosing the strategy, hashing keys and having key ranges to partition data both have advantages and trade-offs, so it is important to choose the one suitable in the situation. Key range could be used if there is a key range that is evenly distributed, and key hash when there are still a lot of hotspots by using the key range.

Partitioning and Secondary Indexes

Partitioning by key-value pairs work nicely with the partitioning strategies discussed above. However, things get more complicated with secondary indexes. Secondary indexes will be discussed more in the upcoming sections. A brief introduction for secondary indexes is that they are created to improve the query performance that are based on multiple keys. For example, when searching for a job in a job board, selecting criteria such as industry or experience level could be a situation to use secondary indexes. Secondary indexes make partitioning more complicated since they don't identify a single record, but rather search for records based on the conditions. Partitioning by key-value pairs wouldn't work in this situation.

Scatter/gather is a technique where each partition has its own secondary indexes. In this case, writing data would be efficient since only the partition containing that specific part of the index is where the data is written. However, reading queries are less efficient. If queries require searching on different partitions, each partition is searched separately which is inefficient.

It is also possible to partition secondary indexes by the term. Instead of having distributed secondary indexes across the partitions, this approach has a global index that covers all the partitions. Having a range of secondary indexes separated in multiple partitions make read queries more efficient from the scatter/gather technique.

Based on the situation, it is important to choose the correct technique. If write queries are used more often, it is better to use the scatter/gather technique, whereas if read queries are dominantly used, partitioning indexes by term is the structure to choose.

Request Routing

After partitioning the dataset into multiple nodes, it is important to make sure the request connects to the right partition. On a high level there are mainly three ways to achieve this.

1. Allow clients to contact any node (e.g., via a round-robin load balancer, which distributes incoming requests evenly across nodes without considering partition ownership.). If that node coincidentally owns the partition to which the request applies, it can handle the request directly; otherwise, it forwards the request to the appropriate node, receives the reply, and passes the reply along to the client.

2. Send all requests from clients to a routing tier first, which determines the node that should handle each request and forwards it accordingly. This routing tier does not itself handle any requests; it only acts as a partition-aware load balancer.

3. Require that clients be aware of the partitioning and the assignment of partitions to nodes. In this case, a client can connect directly to the appropriate node, without any intermediary [10].

The key problem is that how does the component, (which could be the routing tier, the client, or the partition in the node) know about changes in the other partitions in other nodes? If the deciding component is not up-to-date with the location where data that the client is requesting, this could lead into data inconsistency.

To solve this problem, many distributed data systems use a separate coordination service such as the ZooKeeper, which registers each node and maintains the authoritative mapping of partitions to nodes, to keep track of the data [10]. Whenever a partition changes or a node is added or removed the ZooKeeper is updated to keep data consistency.

3.3 Optimization

Several factors influence the performance of the database. The design of the database, including normalization and partitioning, which was discussed in the previous subsection, is one of the factors. Indexing is another huge part of increasing the performance of the database as the amount of data increases. Lastly, changing the physical query of the database is a way to optimize the performance of the database. In this section, we will be discussing specific ways to optimize the performance of databases.

3.3.1 Indexing

Suppose that there is a table called students with the following keys: id, name, grade. Here is an example table:

ID(PK)	**Name**	**Grade**
1	Alice	90
2	Bob	80
3	Charlie	70
4	David	70
5	Eva	80

Table 3.8: Original Students Table with ID, Name, and Grade

If we want to find students with a certain grade (80%), we could run a SQL query something like this.

```
SELECT *

FROM Students

WHERE grade = 80;
```

This query will visit all the rows and then find the students with grades 80. However, this query becomes more inefficient as the size of the table grows, as the time complexity would be O(n). If the table was sorted by grades, this will be much easier, since we could do a binary search to make the search speed O(log n). This is where indexing is applied.

An index could be created where the grades are sorted in order, each having a reference to the row of the student table. This way, whenever we want to find the name of the students with a certain grade, instead of looking at the Students table, we can look at the index, where the grades are sorted, find rows with a certain grade, for each row, find the reference of the student table, and return the name of the student. Here is an example how the index might look like:

Grade	**ID**
70	3
70	4
80	2
80	5
90	1

Table 3.9: Index on Grade: sorted by Grade with references to Original Table

In this index, grades are sorted by order, and they include a reference to the student table's primary key, which is the student's id. This will significantly improve the read queries for databases for large sets of data.

However, indexes take up additional memory and can slow down write operations, such as inserts, updates, or deletes. It is important to use indexes on every column. Indexes should be used in situations where they could significantly increase the performance of read queries.

There are multiple data structures that could be used to implement indexes, but the most common ones are B+trees. In order to understand why B+trees are used the most we have to learn what are B trees. B trees are somewhat similar to the binary search trees. However, instead of having one value for every node, it contains multiple values for each node.

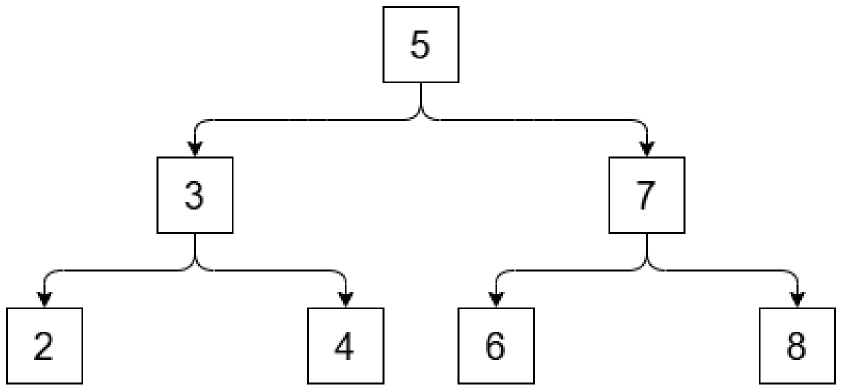

Figure 3.2: Model of a binary tree to explain datastructures used for database indexing.

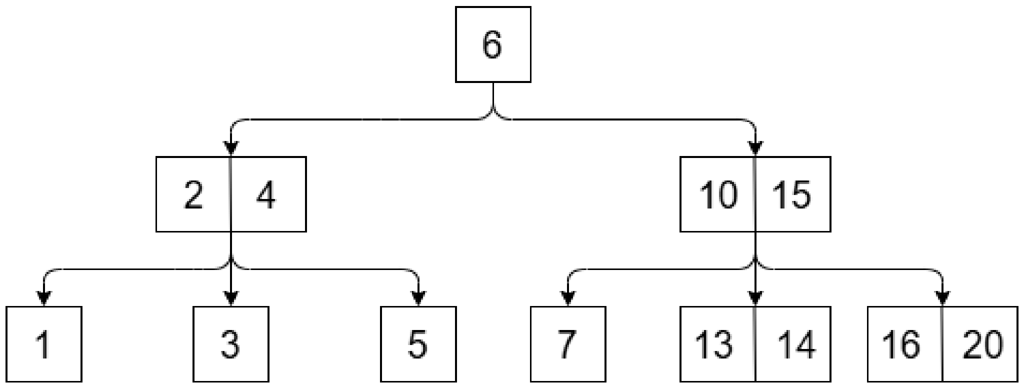

Figure 3.3: Model of a b tree to explain datastructures used for database indexing.

As we can see from the figures above, B trees can hold multiple nodes, such as nodes 2 and 4, or 10 and 15. This allows multiple partitions, allowing a balanced and faster search. Furthermore, as multiple values could be in a single node, the height of the tree would be smaller. This means that there are fewer disk I/O's per file operation as the database stores data on disk, resulting in faster query execution [7]. One down side of B trees is when making range queries, such as finding all students who have grades 20 to 80. If these two data are separated from the root, this could be inefficient. B+trees are a variation of B trees to solve this issue.

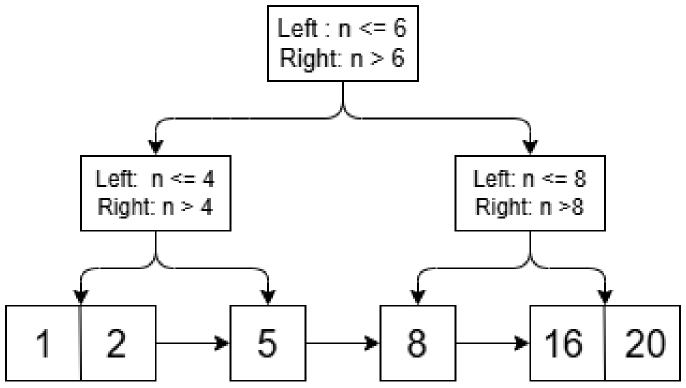

Figure 3.4: Model of a b+ tree to explain datastructures used for database indexing.

B+trees store data only in the leaf node, which are the bottom most nodes without any children. The other nodes store keys only for navigation, like the explanations on the internal node (less than equal to six goes left, greater than six goes right) on the figure above. Also, all the leaf nodes are connected to the next leaf node. B+trees are commonly used in indexes for modern database management systems such as MYSQL, PostgreSQL, and SQLite.

3.3.2 Query Optimization

Optimizing the SQL queries could also increase the performance of the database. Here are the 6 queries tune SQL queries. https://www.geeksforgeeks.org/sql/sql-performance-tuning/

- SELECT fields instead of using SELECT *
- Avoid SELECT DISTINCT
- Use INNER JOIN instead of WHERE for Joins
- Use WHERE instead of HAVING
- Limit Wildcards to the end of a search term
- Use LIMIT for sampling query results

3.3.3 N+1 problem

The N+1 problem is a common problem when using the object relational mapping, or simply ORM. ORM is the process of abstracting the connection between programming language entities and their corresponding database elements [12]. The definition of the problem is when an application retrieves a list, then performs additional queries for each item's related data, which results in an inefficient query. (1 + N queries instead of optimized joins).

Similar to the example above, assume there is a students table with keys student_id, student_name, and another table called courses with the keys course_id, student_id, and course_name.

If we were to get all students and courses they are taking in plain SQL, we would write something like this:

```
SELECT s.student_id, s.student_name, c.course_name
FROM students s
JOIN courses c ON s.student_id = c.student_id;
```

Using an ORM, we could do something like this:

```
students = session.query(Student).all()
for s in students:
    for c in s.courses:
        print(s.name, c.name)
```

This is where the N+1 problem happens. The way of getting all students and their courses through the ORM translates to the following raw SQL query:

```
SELECT * FROM students;
SELECT * FROM courses WHERE student_id = 1;
SELECT * FROM courses WHERE student_id = 2;
SELECT * FROM courses WHERE student_id = 3;
```

This is inefficient since we are performing 1 query (`SELECT * FROM students`), and then performing *n* queries (`SELECT * FROM course WHERE student_id = n;`). The performance of the database will get worse as the size of n increases. It might be questioned why the ORM translates to inefficient SQL code, and the reason for this is lazy loading. Lazy loading is when the application loads data only when the related data is accessed, and this is a default setting in many ORM frameworks. In this example, the ORM does not know that all the students and their courses are going to be fetched. They only understand each iteration of the nested for loop as a separate query and fetch the data for that specific iteration [9].

There are a few ways to solve the N+1 problem, depending on the tools available by the ORM. Most ORMs provide eager loading, which allows for fetching the related data right away instead of waiting until specific data is called. This is the opposite of lazy loading. This reduces the need for additional queries and increases the efficiency when fetching data. Most ORMs provide customization to decide when and how to perform eager loading, allowing optimization when querying databases. For example, if we want to use eager loading in SQL Alchemy, we would do something like this:

```
from sqlalchemy.orm import joinedload


students = session.query(Student).options(joinedload(Student.courses)).all()
for s in students:
    for c in s.courses:
        print(s.name, c.name)
```

The difference from the previous query is that instead of just fetching data from the students table, this query fetches all course data related to the student beforehand to prevent lazy loading.

Another way to solve the N+1 problem is batch fetching. Batch fetching still keeps the idea of lazy loading, but instead of fetching a single row of data, it fetches multiple rows of data at once. For example, we can write something like this in SQL Alchemy:

```
from sqlalchemy.orm import relationship


class Student(Base):
    __tablename__ = ”students”
    id = Column(Integer, primary_key=True)
    name = Column(String)
    courses = relationship(
        ”Course”,
        back_populates=”student”,
        lazy=”selectin”,       # enable batch fetching
        batch_size=10          # load 10 parent IDs per batch
    )


students = session.query(Student).all()
```

When creating the relation in SQLAlchemy, we can specify the batch size. In this example, by setting `lazy=”selectin”` and `batch_size=10`, we are telling the ORM when to fetch courses from students and to load 10 student IDs per batch, resulting in the following query:

```
SELECT courses.id, courses.name, courses.student_id
FROM courses
WHERE courses.student_id IN (1,2,3,4,5,6,7,8,9,10);
```

```
-- Batch 2

SELECT courses.id, courses.name, courses.student_id

FROM courses

WHERE courses.student_id IN (11,12,13,14,15,16,17,18,19,20);
```

This will avoid the N+1 problem by instead of fetching for n queries, it will fetch for m queries, where m is n/batch size. Furthermore, batch fetching is useful when the size of the data becomes larger. Unlike eager loading, this avoids data duplication, where we do not get the same student row repeated for every course. Being able to adjust the batch size is also beneficial when scaling the application.

CHAPTER 4

Caching

4.1 What is Caching

Caching data means to store frequently accessed data in a location that is easy and fast to access (usually in RAM). The reason why caching is used is to ultimately increase the performance and efficiency when retrieving data. Data in the database are usually stored in a disk, which could be a hard drive or an SSD, which usually takes more time to process I/O. Caching could help decrease the amount of time to process this data significantly. One might suggest then why not put all the data in the cache, but this would not work since the cache could not store a lot of data as much as the database does. The cost of the hardware of the cache is much more expensive, and as the size of the cache increases, the search time will also increase, decreasing the speed and defeating the whole purpose of caching. Caching can be used at several levels. This article will mainly focus on caching in databases and web pages. There are multiple strategies that could be used when caching these two areas. Another important part of caching is the invalidation strategies. Deciding when the cache expires and when it updates could affect the performance and data consistency. Choosing the right technique in the given situation will be essential.

4.2 Database Caching Strategies

There are multiple database caching strategies that could be applied when designing cache systems. Depending on the type of request (read, write) of the user, and which part of the application is responsible for fetching data from the database or managing the cache, there are five main database caching strategies.

Cache Aside

Cache Aside is also called Lazy Loading. The application is in control of managing the cache. Let's look at the diagram below.

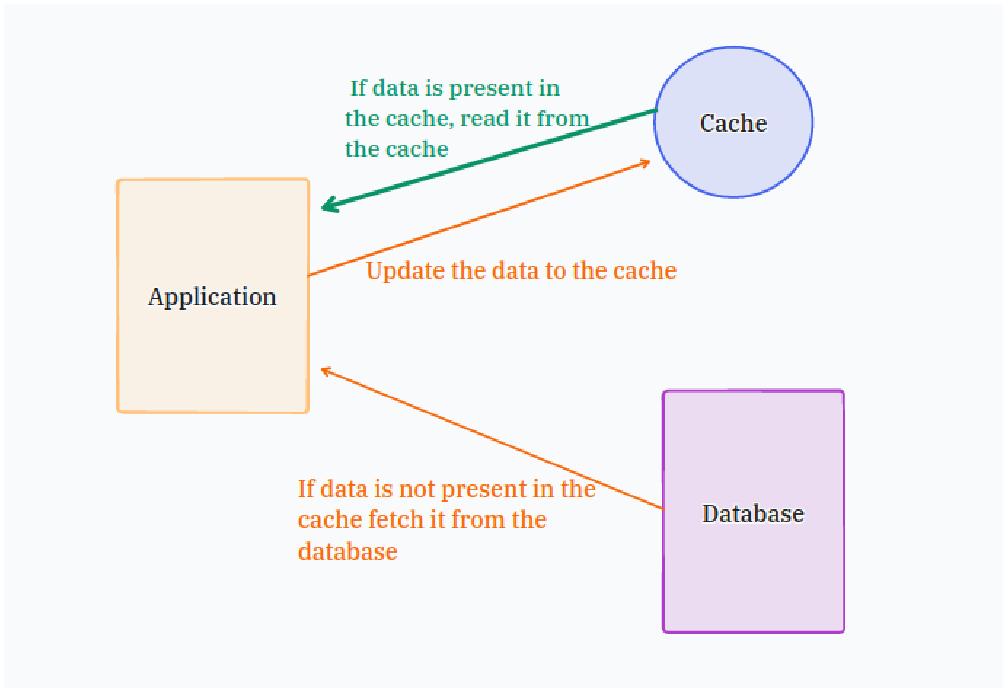
If data is present in the cache, read it from the cache
Cache
Application
Update the data to the cache
If data is not present in the cache fetch it from the database
Database

Figure 4.1: Model of Cache-aside Pattern to explain common database caching strategies [14].

As the diagram shown above, the client first checks the cache to see if the read request data is in the cache. If the cache exists, this is called a cache hit, and the client uses it right away. If the cache does not exist, this is called a cache miss. When a cache miss occurs, the client then sends a request to the database server to fetch the data. After that, the client transfers the data to the cache, so that there could be a cache hit the next time the data is needed. This will increase the performance of the application when the same data is called multiple times. When there is a write request, the application will first communicate with the database and then update the cache.

This strategy is useful when the application requires a lot of read requests from the database, since the application could just check the cache instead of querying the database. One disadvantage of this approach is that there could be an inconsistency between the cache and the database. Data directly written in the database might not be consistent with the data in the cache, since writing data to the database and writing data to the cache does not happen at the same time.

Read-Through

In the read-through strategy, the cache level manages fetching data from the database. Here is a diagram of how the read-through strategy works:

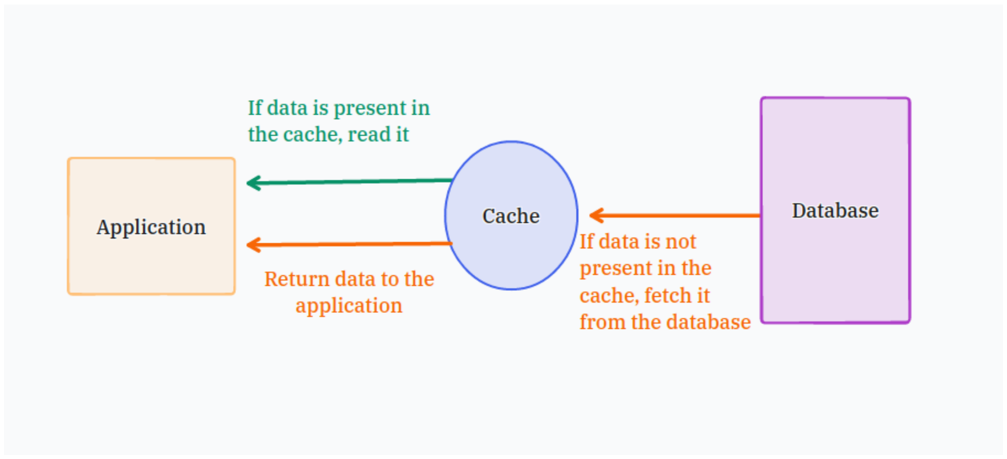
If data is present in the cache, read it
Application
Cache
Database
Return data to the application
If data is not present in the cache, fetch it from the database

Figure 4.2: Model of Read-Through Pattern to explain common database caching strategies [14].

Whenever there is a read request from the application, the application first checks the cache. If there is a cache hit, it simply returns the data back to the application. If there is a cache miss, the cache level fetches the data from the database. Since the cache manages fetching data, it simplifies the application logic when retrieving data compared to when the application handles fetching data.

This strategy only involves read requests from the application, which means that other strategies could be used for write requests. The write-through strategy is generally great to be used with the read-through strategy.

Write-Through

The write-through strategy is very similar to the read-through strategy. The diagram is almost identical.

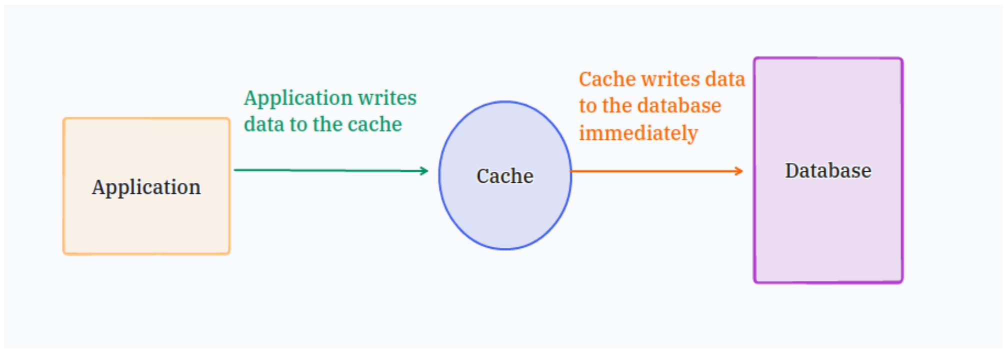
Application writes data to the cache
Cache writes data to the database immediately
Application
Cache
Database

Figure 4.3: Model of Write-Through Pattern to explain common database caching strategies [14].

Nothing happens when there is a read request from the application, but when there is a write request, the data will first be written in the cache and then into the database. This ensures data consistency when paired with the read-through strategy. One downside of this strategy is that since it writes to both the cache and the database layer, the latency of the request increases.

The read-through and the write-through strategies are not meant to be used for all access to data in the application layer. Using these strategies for all database queries could result in a decrease in performance, since the caching layer is not intended to keep every single data.

Write-Back

The write-back strategy is similar to the write-through strategy, but the writes to the database happen with a delay. The cache is still in charge of writing data to the database. Let's look at the diagram:

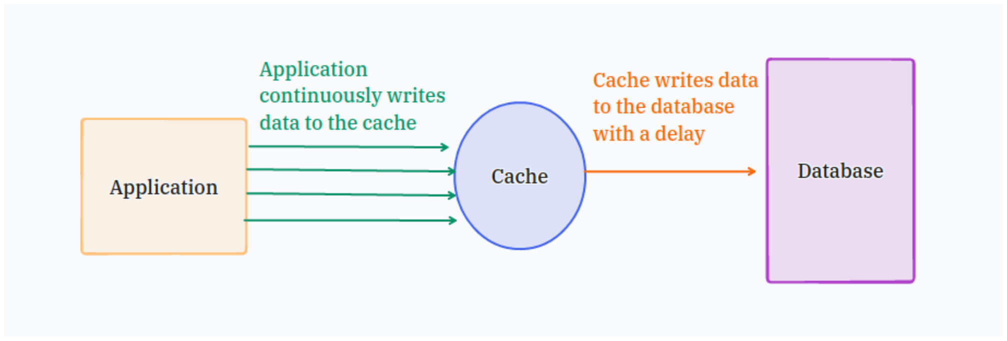
Application
Application continuously writes data to the cache
Cache
Cache writes data to the database with a delay
Database

Figure 4.4: Model of Write-Back Pattern to explain common database caching strategies [14].

Since the cache only writes to the database after a certain amount of time, this strategy is beneficial when there are multiple write requests to the same data at the application level. This will decrease the write queries from the cache level to the database, which could increase the performance. However, similar to the read-through and write-through strategies, since the data is written to the cache first, writing data to the database could potentially fail if the caching fails.

Write Around

This strategy is similar to the cache-aside strategy, but it has more specific instructions for write requests.

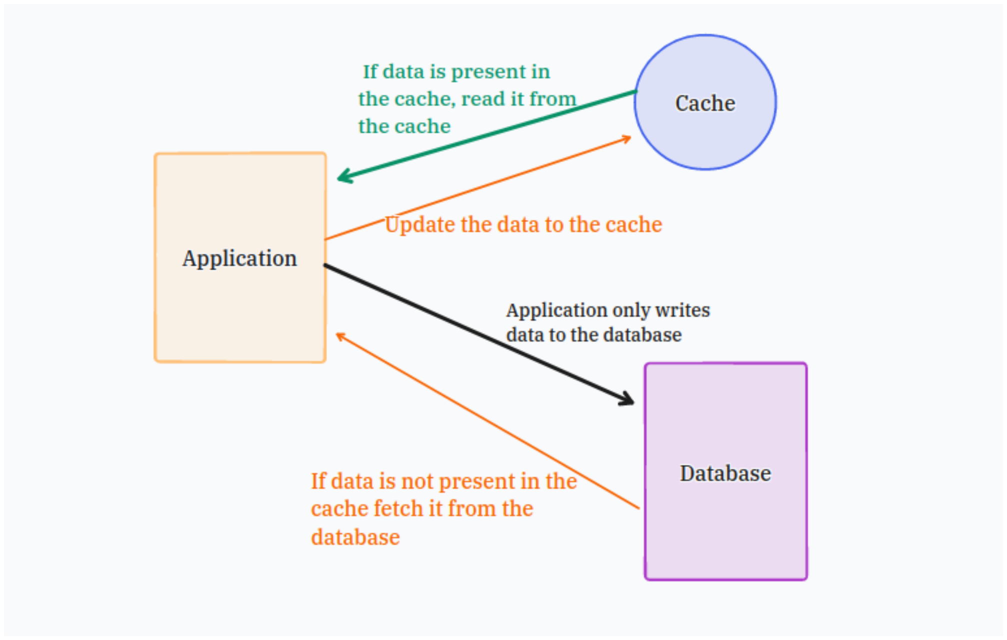
If data is present in the cache, read it from the cache
Cache
Update the data to the cache
Application
Application only writes data to the database
Database
If data is not present in the cache fetch it from the database

Figure 4.5: Model of Write-Around Pattern to explain common database caching strategies [14].

The difference between the write-around strategy and the cache-aside strategy is that when there is a write request, only the data in the database is updated. The data in the cache is only updated when there is a cache miss from a read request. This way, the cache is not overflooded with unnecessary data. The disadvantage of this strategy is that recently written data will always result in a cache miss. This is why the write-around strategy is suitable when the application has a lot of read requests and data is rarely updated.

4.3 Redis

Until now, we have learned the types of database caching strategies. But how are these strategies actually implemented? Redis is a widely used open-source data structure store that makes these caching patterns practical at scale.

Redis works entirely in memory, which gives it extremely fast read and write speeds. Most databases store data on disk, so every query involves slower I/O operations. Redis avoids this by keeping data in RAM, allowing applications to access cached values in microseconds. This speed makes it perfect for handling database caching strategies [5].

Redis also supports a wide range of data structures—strings, hashes, lists, sets, sorted sets, and more. These structures map naturally to different caching scenarios. For example, strings work well for caching simple query results, hashes can store user profile objects, and sorted sets can maintain real-time rankings or leaderboards. Because Redis supports many data structures, it is not limited to simple key–value caching. It can also function as a fast and lightweight data engine that handles situations that require complex caching.

Reliability is another reason Redis is chosen for database caching. Redis provides persistence options (RDB snapshots and AOF logs) so cached data or application-critical information can survive server restarts. It also supports replication and clustering, which distribute data across multiple nodes to achieve both high availability and horizontal scalability. These features ensure that the caching layer does not become a single point of failure.

In summary, Redis transforms abstract caching strategies into practical and high-performance solutions. Its low latency, adaptable data model, and robust operational features make it a dependable component in web applications. These are the reasons why Redis will be used to implement database caching strategies in this project.

4.4 Client Side Caching

People find it annoying when it takes too long or laggy to load a website they want to access. This is where client side caching could be beneficial. Client side caching is a technique where data is cached inside the client's device unlike the application level caching. Frequently used static data are normally cached on the client side to improve the performance and efficiency of web applications. The reduced time to load the webpages result in enhanced user's satisfaction and the overall experience. Static data could also be available when the user's device is offline, which could be useful for some mobile and web applications.

There are different types of client side caching that could be used, but the most common ones are local storage, session storage, and browser cache. Local storage and session storage are types of web storage APIs, where developers can store data on the client side that is not automatically cached by the browser. This could include saving users' preferences such as theme or language, or simple offline data. The difference between local storage and Session storage is the scope of the browser. Local storage is shared among all tabs and windows of the same origin, but session storage is specific to a single tab. For session storage, they will be deleted once a tab is closed. However, local storage will be saved even after the tab and the browser are closed.

Browser cache is where most of the static elements of the web application, such as HTML, CSS, JavaScript, and images, are automatically cached. The way how and where these caches are stored depend on the browser the user is using, but the core concepts are the same. Although browser cache is automatically stored, there are some parts that could be controlled within the browser cache, which is called cache control. Cache control is an HTTP response header that gives instructions to the browser on how to cache resources. There are 15 total cache control techniques, and each of these techniques serves different purposes, depending on how often the data changes and how much freshness the application requires, but we will be discussing the most common ones.

4.5 Caching Invalidation

After a cache is stored whether in the client or the server side, it is crucial to know when the cache should be validated. For example, let's assume the user's profile is cached in the browser. Without any caching invalidation, even after the user's profile is updated, the user will not be able to see the changes if the cache still exists in the browser. This could lead to crucial data inconsistency as the importance of the data increases such as sensitive security data. Caching invalidation strategies are needed to set when the cache should be stored, updated, or deleted. Choosing the right caching invalidation strategy can lead to better performance of the application, freeing up the memory space and increasing cache-hit rates.

One of the most common types of cache invalidation is time-based invalidation. This type of technique invalidates cached data after a certain time period has passed, which could be decided by the developers [11]. Although it is simple and widely used, it also has limitations because it may invalidate data too early or too late depending on the situation. In some systems, a timeout value can be added directly in a cache configuration file to specify how long the entry should stay valid. This value can also be customized to fit the needs of the application. Some examples of time-based invalidation could be a recipe website may refresh its cache daily to show the latest trending dishes, or a movie streaming service that could be refreshed where there are new releases every few hours.

Another type of invalidation is event-based invalidation, which happens when a specific event is triggered in the system. This is useful when the cached data is tied closely to an event or state change. For example, when a blog post is updated, the old cached version of the post should be removed so that users can see the most recent version right away. This ensures that the content stays consistent with changes happening inside the system.

Another type is group-based invalidation, which removes cache based on a larger group or category. This is helpful when many cache entries belong to the same section of the application, and clearing them one by one would be inefficient [11]. This type of cache invalidation technique is commonly used when a larger group of data must be invalidated at once. For example, on a news website, if the politics section is updated, all cached articles under that section should be invalidated to make sure the latest content is displayed. In eCommerce systems, updating a product category may require invalidating all caches related to that category.

Lastly, there is validation-based invalidation. This cache invalidation technique relies on the browser checking with the server to see if the cached data is still valid instead of removing it after a fixed time or when an event occurs [11]. A common technique for this is using ETags, which are small identifiers generated by the server to represent a specific version of a resource. When the browser stores a cached file, it also keeps the ETag value and later sends it back to the server to confirm whether the resource has changed. If it has not changed, the server replies with a "Not Modified" response, which allows the browser to continue using the cached version without downloading it again. This approach helps reduce unnecessary network usage while still ensuring that users receive the most up-to-date content, making it useful for situations where data changes unpredictably.

Not all of these caching strategies could be used both in the server side and the client side. Each technique serves a different purpose, and some are only effective when controlled by the system that manages the data. Time-based invalidation is mostly handled on the server side through cache headers, while event-based and group-based invalidation occur exclusively on the server since they depend on changes happening within the application itself [11]. On the other hand, validation-based methods such as ETags rely on collaboration between the browser and the server, where the server generates validation tokens and the client uses them to check freshness. Understanding where each technique is applied helps ensure that caching remains efficient, consistent, and aligned with the needs of the application.

CHAPTER 5

Implementation

As discussed in chapter 2 2, chapter 3 3, and chapter 4 4, numerous design choices must be made when developing a software. This chapter outlines the design choices in the backend and the frontend side of the project. The design choices made in the backend will focus on the reasons why certain decisions were made in the software architecture, database design choices, and database caching systems. The frontend section examines the decisions regarding frontend caching in detail, along with other relevant implementation considerations.

5.1 Backend Design

Multiple options are available when selecting a web framework for backend development. Java Spring Boot is chosen for this project for two reasons. First, this project provided an opportunity to explore a backend framework beyond Python Flask, which was the only framework I have previously used. When I was choosing my framework at the beginning of the project, I planned to go back to South Korea after graduation. I wanted to choose a framework commonly used in the software industry in South Korea, which was Java Spring Boot. Other backend web frameworks, such as Django and NestJS, are used in South Korea, but Java is the most dominant, making it practical and relevant to the industry [4].

The second reason for selecting Java Spring Boot emerged during the development process. It became clear that Spring Boot integrates effectively with the chosen software architecture, particularly in terms of modular design. Java's structured organization around pacakges, classes, and interfaces aligns naturallly with modular boundaries. Each domain (cuisine, region, reviews) can be encapsulated within its own package hierarchy, promoting high cohesion within modules and reduced coupling between them. This architectural compatibility is discussed in greater detail in a later subsection.

Beyond the backend framework, additional technologies were selected to support development efficiency and scalability. Supabase was chosen as the database hosting solution to leverage a managed cloud service, allowing the project to focus on application logic rather than data and infrastructure management. Redis was selected as the caching solution due to its widespread industry adoption, proven performance, and suitability for implementing efficient caching strategies within the application.

5.1.0.1 Software Architecture

Monolithic vs Distributed

Multiple design approaches were considered when deciding whether the local cuisine application should adopt a monolithic architecture or a distributed system architecture. Until now, all the applications that I have created were monolithic applications. As a student, the scope of solo projects were small enough that monolithic architectures were appropriate, and all software developed or contributed to during internships also employed a monolithic architecture.

For this reason, I wanted to create an application that has a distributed system architecture in the beginning. I wanted to try something that I have not tried, and also gain experience using this architectural pattern. Developing an application using both monolithic and distributed architectural styles provides valuable perspective when selecting an appropriate software architecture, as it enables a deeper understanding of the trade-offs involved in architectural decision-making.

However, as I continued exploring architectural style for the local cuisine application, the monolithic architecture seemed to be a more compelling option. I plan to keep maintaining and expanding on this project even after graduation and the final submission of the independent study. Through the study of software architecture theory, it became clear that adopting a distributed system architecture at an early stage could introduce unnecessary complexity, which may not be justified for a system of this scale. The benefits that I would gain by setting up the distributed system architecture outweigh the drawbacks.

First, Modular monolithic architecture is appropriate for the local cuisine application compared to the distributed system architecture in terms of scaling. Scalability is one of the primary advantages of a distributed system architecture. It is possible that the local cuisine application could grow significantly and attract thousands of daily users, and a distributed architecture could provide clear benefits in handling increased traffic and workload. However, in the current stage of development where the aim of average users is less than thousands of users, the modular monolithic architecture is easier to maintain the application.

Furthermore, as the local cuisine application is still relatively small, it would not benefit from the loose coupling that the distributed system architecture has. It is possible to divide each part of the application into different services, such as having an individual service for cuisines, regions, and reviews. However, introducing this level of separation would be unnecessary at the current stage unless the applications scales. The distributed architecture approach becomes more valuable only when the application scales significantly and requires independently evolving services. For the present scope of the local cuisine application, the advantages of loose coupling provided by a distributed architecture do not outweigh the added complexity such as inter-service communication or network latency.

Another big reason the monolithic architecture was chosen was the cost. Although a distributed system architecture offers several advantages, the associated infrastructure costs are critical. Deploying and maintaining multiple servers to support distributed services can rapidly increase the cost, that exceeds the scope of a personal project. On the other hand, the monolithic architecture only needs one server, which is cost-effective and easier to maintain over the long term.

Finally, the modular monolithic architecture was chosen since it preserves the possibility of futer evolution into a distributed system architecture. In case the local cuisine application needs to scale, the existing modular structure provides two options: maintain the current monolithic deployjment, or incrementally refactoring modules into independent services. The clear separation of concerns within the modular monolith allows potential transition, as the well-defined module boundaries can serve as a foundation for service extraction. This flexibility allows architectural decisions to be revisited based on observed system growth, performance requirements, and usage patterns, rather than being determined from the beginning.

Fault tolerance was the main trade-off considered in the architectural decision-making process. While a distributed system architecture provides stronger fault tolerance, the overall requirements and constraints of the project led to the selection of a monolithic architecture for the local cuisine application.

Layered or Modular Monolithic

There are two monolithic architectures introduced in the theory section: the Layered architecture and the Modular Monolithic architecture. For the local cuisine application, I decided to use the modular monolithic approach. The main reason behind this decision is the clear separation of concerns it provides between different parts of the application.

The application is composed of three main components: regions, which allow users to explore the geographic areas supported by the local cuisine application; cuisines, which provide detailed information about the culinary offerings within a selected region; and reviews, which enable users to interact with one another by submitting and viewing feedback on specific cuisines.

The modular architecture is well-suited to this application, since the local cuisine application has a clear separation of functional domains: region, cuisine, and reviews. If this were implemented using a layered architecture, the application would be organized strictly by technical layers (such as controller, service, and repository) rather than being structured around the distinct functional domains of the system. This structure not only mirrors the real-world organization of the application but also improves maintainability by isolating changes to individual modules and reducing the risk of unintended side effects across the system.

Furthermore, the modular monolithic architecture could also retain the organizational benefits of a layered architecture. For example, Within each module, technical layers—such as presentation, workflow, persistence, and database—can be implemented independently, ensuring clear separation of concerns and consistent layering practices across the system. Each module could contain these layers within its own module, which allows modular monolithic inherit the advantages of the layered architecture. Here is the structure of the modular monolithic architecture for the local cuisine application.

The figure above shows three modules in the local cuisine application: the region, cuisine, and reviews. Within each module, the API, service, and domain packages are included. The api package is a part of the presentation layer, where information is organized and sent to the front end. The service package is the workflow layer where all the business logic is included. Finally, the domain package is the persistent layer where entities are mapped persistently to the relational database.

This architecture of the modular monolithic structure allows separation of modules and layers, which makes it easier to maintain the code. The separation is also beneficial when the application would be developed collaboratively in the future as well.

However, it is possible that the decision to implement the modular monolithic architecture for this project could be viewed as excessive. There are only three modules for the local cuisine application, and it is possible to just have all of these modules under the layers of the layered monolithic architecture. Although it is currently a small-sized application, the decision to adopt a modular monolith was guided by the potential for future scalability, allowing the system to evolve without requiring a major architectural overhaul. Moreover, I have already experienced the layered monolithic architecture throughout class projects in the past, and I wanted to explore different structures to enhance decision-making skills. Practical considerations, including personal learning goals and the desire to experiment with modular design, also informed this decision.

5.1.0.2 Database Systems

ERD Diagram and Normalization

The database for the local cuisine application is designed to store three main pieces of information: the region, the cuisine, and the reviews. Accordingly, there are three main tables: the regions table, the cuisines table, and the reviews table. As shown in the entity relationship diagram, the regions table has a one-to-many relationship with the cuisines table, since one region can contain multiple cuisines, while each cuisine belongs to a single region. The cuisines table also has a similar relationship with the reviews table. Each cuisine will have multiple reviews, but each review is associated with one cuisine. These relationships are implemented using foreign keys. The region_id in the cuisines table references the regions table, and the cuisine_id in the reviews table references the cuisines table. The regions table includes information about the area, and has attributes id (primary key), country, name, and region. The cuisines table contains details about individual cuisines, including id, cuisineName, and description, and references the

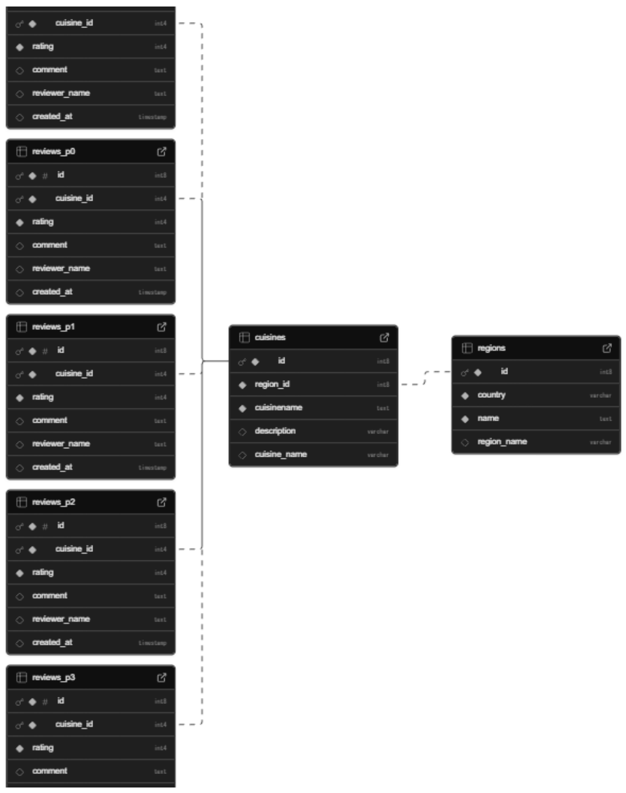

Figure 5.1: Figure of the entity relationship diagram for the local cuisine application.

associated region through region_id. The reviews table stores reviews for each cuisine and includes the attributes id, rating, comment, reviewer_name, and created_at.

To enforce the integrity of databases, normalization the data is crutial. In the local cuisine database, each table satisfies the requirements of 3NF. All non-key attributes in the regions table depend solely on the primary key region_id. Similarly, in the cuisines table, attributes such as cuisine name and description depend only on cuisine_id, while the region_id is used as a foreign key to represent relationships rather than introducing transitive dependencies. Finally, in the reviews table, attributes related to the reviews (such as rating and comment) depend only on the primary key review_id.

As a result, the database structure avoids transitive dependencies, minimizes redundancy, and ensures data integrity. This fulfills the requirements of the third normal form. The third normalization form for the database ensures that redundant storage of region or cuisine data across multiple tables to reduce the risk of inconsistent data, and allows the database to scale without adding or restructuring existing tables. This database design also goes well with the modular design of the application since each module (region, cuisine, and review) is clearly separated.

Currently, the region and cuisines table are filled with data from the Wikipedia API. The data was gathered through scraping using Claude AI. There were other options when choosing the information for these tables such as external APIs or datasets. However, they did not include all the information that the local cuisine application requires. For example, some APIs and datasets did not include the original region of the cuisine, or the description of the cuisine. For the following reasons, the Wikepedia API was chosen since it included all the information needed for the local cuisine application, and was able to gather information efficiently. The reviews table includes sample reviews generated by artificial intelligence for testing purposes, but will be filled with human written reviews in the future.

Optimization

When testing the API endpoints of the local cuisine application, there weren't any significant performance issues when tested locally. However, considering that the database will grow and there will be multiple clients connecting to the API simultaneously in the future, a few database optimization techniques were implemented.

The local cuisine application is designed that the cuisines and the reviews are searched the most and make database queries. When the user searches for the cuisines, the cuisines table is accessed and the reviews of each cuisine are also loaded from the reviews table. Furthermore, the cuisines table currently has only about 30 rows, but will expand in the future. The reviews table will also grow exponentially as the number of cuisines and the active users increase over time. This is why the cuisines table and the reviews table need to be optimized.

Among the database optimization and the evolution techniques discussed in the theory section, indexing is used for the cuisines table. Indexing is applied to the cuisines table to improve search performance and reduce query execution time. One of the core features of the system is retrieving all cuisines belonging to a specific region, and this requires frequent filtering using the region_id attribute.

Without indexing, queries that filter cuisines by region would require a full table scan, resulting in increased response time as the number of records grows. This is why an index was created on the region_id column of the cuisines table. This allows the database to efficiently locate relevant rows using an index scan instead of scanning the entire table.

The index was created using the following SQL statement:

```
CREATE INDEX idx_cuisines_region_id
ON cuisines (region_id);
```

By indexing the foreign key attribute region_id, the database can quickly retrieve all cuisines associated with a given region and significantly improve performance for read-heavy operations. This optimization is effective for the application, since cuisine data is queried frequently but updated infrequently. As a result, the application has minimal write overhead while providing faster search and retrieval times. As the dataset scales, this indexing strategy helps maintain consistent performance and improves the overall responsiveness of the application.

Another optimization done in the local cuisine application is avoiding the N+1 prohblem. As mentioned in the query section, the N+1 query problem is a common performance issue in ORM-based applications, and this application uses the Hibernate ORM. When fetching the cuisines, the cuisine repository uses the following gmethods.

```
public interface CuisineRepository extends JpaRepository<Cuisine, Long> {
    List<Cuisine> findByRegion_RegionName(String regionName);
    List<Cuisine> findByRegion_Country(String country);
}
```

These methods generate SQL queries similar to

```
-- Example for findByRegion_RegionName
SELECT c.id, c.cuisinename, c.description, c.region_id
FROM cuisines c
JOIN regions r ON c.region_id = r.id
WHERE r.region_name = ?;


-- Example for findByRegion_Country
SELECT c.id, c.cuisinename, c.description, c.region_id
FROM cuisines c
JOIN regions r ON c.region_id = r.id
WHERE r.country = ?;
```

If the application were to access the region for each cuisine individually, such as calling `cuisine.getRegion().getRegionName(`
this could trigger one additional query per cuisine, resulting in the classical N+1 problem. However, the local cuisine application avoids this entirely by performing all filtering and joining at the repository level.

For example, the service method:

```
public List<String> getCuisinesByRegion(String regionName) {
    List<Cuisine> cuisines = cuisineRepository.findByRegion_RegionName(regionName);
    return cuisines.stream()
                   .map(Cuisine::getCuisineName)
                   .collect(Collectors.toList());
}
```

only accesses the cuisineName field of each Cuisine entity. The findByRegion_RegionName is a repository-level method that performs filtering and joining directly in the database, returning only the relevant Cuisine records. Since the related Region entity is never traversed in application code, no additional queries are executed per cuisine. This ensures that the number of executed queries remains constant regardless of the number of cuisines returned and effectively preventing the N+1 problem.

Furthermore, lazy loading is used for the region association in the Cuisine entity:

```
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = ”region_id”, nullable = false)
private Region region;
```

When a Cuisine is fetched, only the region_id foreign key is retrieved from the database, and the full Region entity is loaded only if it is explicitly accessed in the application.

Of course, eager loading could be used instead of lazy loading to prevent any potential N+1 problems, but this approach would be inefficient because all region data would be loaded for every cuisine, even if the region information is not needed for the operation.

The combination of lazy loading and repository-level filtering allows the application to retrieve only the necessary data, which makes sure that the performance is efficient while still avoiding N+1 queries. The repository query returns all relevant Cuisine records in a single SQL statement, and the service layer only accesses the fields it needs. This will never traverse the Region entity unless explicitly required and avoid the N+1 problem.

Batch fetching was also not necessary in this design because the repository query already fetches all required data in a single call. The total size of the data (number of cuisines and regions) is small enough that performance is not a concern as of now.

Evolution

To prepare for scaling the application, partitioning is applied to the reviews table. The reviews table is partitioned through the partitioning by key-value pair technique discussed in the theory section. The reviews table has the potential to have the most rows in the future, so scalability is crucial to this specific table. Partitioning the reviews table will improve scalability and query performances by distributing data and query load across multiple nodes. Most queries in the local cuisine application are centered around a single entity, such as retrieving cuisines by region or reviews by cuisine. Partitioning by key-value allows requests to be routed directly to the partition that owns the relevant data.

Here is the query that is used to partition the reviews table.

```
CREATE TABLE reviews_p0 PARTITION OF reviews
    FOR VALUES WITH (MODULUS 4, REMAINDER 0);

CREATE TABLE reviews_p1 PARTITION OF reviews
    FOR VALUES WITH (MODULUS 4, REMAINDER 1);

CREATE TABLE reviews_p2 PARTITION OF reviews
    FOR VALUES WITH (MODULUS 4, REMAINDER 2);

CREATE TABLE reviews_p3 PARTITION OF reviews
    FOR VALUES WITH (MODULUS 4, REMAINDER 3);
```

The reviews table is partitioned using hash-based partitioning on the partition key. The `MODULUS 4` clause specifies that the table is divided into four partitions, while the `REMAINDER` value determines which partition a row belongs to based on the hash value of the partition key modulo 4. As a result, each review record is assigned to one of the four partitions as shown in the ERD diagram.

This will distribute the reviews data evenly across each partition, and also reduce the probability of hot spots. This way of partitioning will improve scalability as data increases. Hash-based partitioning also aligns with how the application accesses data in the database, since most queries retrieve reviews by a specific cuisine identifier. By distributing reviews across partitions, query can be processed parallel, and improve query performance as well.

At the current stage of development, secondary indexes were not introdueced into the database design. Although secondary indexes can significantly improve query performance when filtering or sorting data by multiple attributes, they also add complexity to partitioned systems, particularly in terms of request routing and index maintenance. In this application, reviews are mostly accessed through their associated cuisine, and there is no requirement to sort or filter reviews by multiple criteria such as rating and reviewer name, or creation time and rating.

Since queries do not involve complex search conditions across multiple attributes, the benefits of secondary indexes are limited in the current use case. Avoiding secondary indexes simplifies the database design and reduces overhead for write operations. However, this does not mean that secondary indexes will not be implemented in the future. If the application scales and requires filtering reviews by multiple criteria, secondary index could be incorporated to optimize read performance. This approach allows the system to remain efficient and easy to maintain while still being open for future optimization.

In this system, Supabase's managed cloud database (PostgreSQL) is responsible for handling replication across database instances. Supabase supports read replicas, which are additional databases kept in sync with the primary database using PostgreSQL's native replication mechanisms. Read replicas are asynchronously replicated from the primary database. This means that changes on the primary are streamed to replicas without blocking writes, which helps avoid slowing down the primary for scalability while still providing consistent read copies of the data. This approach aligns with the leader–follower pattern discussed earlier, where the primary node handles writes and replicas handle read queries, improving performance and load distribution without introducing the complexity of multi-leader conflict resolution. Supabase provides read replicas that remain read-only and reduce load on the primary to enable low-latency reads across regions if multiple replicas are deployed [13].

5.1.0.3 Caching

Caching is implemented in the local cuisine application to enhance performance and reduce latency for frequently accessed data.

Caching in the local cuisine application improves performance by reducing database access for frequently requested data. By storing frequently accessed information in memory, such as cuisine names by region, the application can serve responses faster, leading to a smoother user experience and reduced database load. This is particularly effective for operations that are read-heavy, such as displaying lists of cuisines when users browse popular regions.

Cuisine data and the reviews data implemented the caching strategy in the local cuisine application. Among the caching strategy discussed in the theory section, the cache-aside pattern for the cuisine data. As the cuisine data won't be written or updated by the users, all the caching strategies that involved writting was not applicable, and the cuisine data is primarily read-heavy. Unlike read-through caching, where the cache automatically loads data on a miss, cache-aside allows the service to explicitly control when and how data is loaded and stored in the cache. This approach provides flexibility for handling cache misses and controlling caching logic, which is ideal for our application where writes are rare and controlled by the system. Write-through or write-back, and write-around techniques were not applicable, as users do not modify cuisine data directly.

For cache invalidation, a time-based expiration strategy was used. Cached entries are set to expire after a fixed duration (e.g., 1 hour) to ensure that the data remains reasonably fresh while still benefiting from caching. Event-based or validation-based invalidation strategies were not used because they require active notifications or triggers when data changes. In this application, cuisine and region data are rarely updated, so implementing event-based or validation-based invalidation would introduce unnecessary complexity without significant benefit.

The cache-aside strategy described above is implemented in the CuisineService class. The service checks the cache before querying the database and stores the result in the cache on a cache miss. This ensures that subsequent requests for the same region can be served directly from memory.

```
public List<String> getCuisinesByRegion(String regionName) {

    String cacheKey = ”region:” + regionName + ”:cuisineNames”;

    // 1. Check cache
    Object cached = cacheService.get(cacheKey);
    if (cached != null) {
        return (List<String>) cached;
    }

    // 2. Cache miss → query database
    List<Cuisine> cuisines =
            cuisineRepository.findByRegion_RegionName(regionName);

    List<String> cuisineNames = cuisines.stream()
            .map(Cuisine::getCuisineName)
            .collect(Collectors.toList());

    // 3. Store result in cache (1 hour TTL)
    cacheService.put(cacheKey, cuisineNames, 3600);

    return cuisineNames;
}
```

The caching process follows the cache-aside pattern and is implemented in three steps. First, a unique cache key is generated using the requested region name. This ensures that each region’s cuisine list is cached independently.

Next, the application queries the cache using the generated key. If the data exists (cache hit), the cached list of cuisine names is returned immediately, bypassing the database entirely. This significantly reduces response time and database load.

If the data is not found in the cache (cache miss), the service queries the database using the CuisineRepository. The retrieved cuisine entities are then transformed into a list of cuisine names. Finally, the result is stored in the cache with time based invalidation of one hour before being returned to the client.

This explicit control over cache access and population reflects the cache-aside strategy, allowing the application to balance performance improvements with data consistency while keeping the caching logic simple and maintainable. The same caching approach is also applied to the getCuisinesByName method, which follows a similar structure and purpose. This also ensures consistent performance improvements when accessing across different cuisines by the cuisine name.

For the review data, initially, a write-through caching strategy was considered. Because reviews are created and deleted by users, it seemed beneficial to keep the cache fully synchronized with the database on every write operation. Under this approach, each write operation would immediately update the cached review list, ensuring strong consistency between the cache and the database. Here is the original code for the write-through caching strategy.

```
public ReviewResponse createReview(CreateReviewRequest request) {

    Cuisine cuisine = cuisineRepository.findById(request.getCuisineId())
            .orElseThrow(() -> new IllegalArgumentException("Cuisine not found"));

    Review review = new Review(
            cuisine,
            request.getRating(),
            request.getComment()
    );

    Review saved = reviewRepository.save(review);

    String cacheKey = "cuisine:" + cuisine.getId() + ":reviews";
    List<ReviewResponse> updatedReviews =
            reviewRepository.findByCuisineId(cuisine.getId())
                    .stream()
                    .map(r -> new ReviewResponse(
                            r.getId(),
                            r.getRating(),
                            r.getComment(),
                            r.getCreatedAt()
                    ))
                    .collect(Collectors.toList());

    cacheService.put(cacheKey, updatedReviews, 300);

    return new ReviewResponse(
            saved.getId(),
            saved.getRating(),
            saved.getComment(),
            saved.getCreatedAt()
    );
}
```

We can see from that a created review is both written in the cache and the database. This approach ensured that cached data remained fully up to date, but it introduced significant overhead. Each write operation required an additional database query to rebuild the cached review list, followed by a cache update. As review activity increased for popular cuisines, I thought that this design could become write-heavy and inefficient.

To address these issues, the caching design was changed to use the cache-aside pattern. Instead of updating the cache on every write, the cache is now updated whenever there is a read request.

```
public ReviewResponse createReview(CreateReviewRequest request) {

    Cuisine cuisine = cuisineRepository.findById(request.getCuisineId())
            .orElseThrow(() -> new IllegalArgumentException("Cuisine not found"));

    Review review = new Review(
            cuisine,
            request.getRating(),
            request.getComment()
    );

    Review saved = reviewRepository.save(review);

    String cacheKey = ”cuisine:” + cuisine.getId() + ”:reviews”;
    cacheService.evict(cacheKey);

    return new ReviewResponse(
            saved.getId(),
            saved.getRating(),
            saved.getComment(),
            saved.getCreatedAt()
    );
}
```

The difference from the write through database caching method is that new reviews are now only written in the database, and the existing cache is not evicted. In this scope, evicted means that the original cache is deleted. With this approach, write operations remain lightweight, consisting only of the database transaction and cache eviction. The cache is repopulated only when a read request occurs:

```
public List<ReviewResponse> getReviewsByCuisine(Long cuisineId) {

    String cacheKey = ”cuisine:” + cuisineId + ”:reviews”;

    Object cached = cacheService.get(cacheKey);
    if (cached != null) {
        return (List<ReviewResponse>) cached;
    }

    List<ReviewResponse> responses =
            reviewRepository.findByCuisineId(cuisineId)
                    .stream()
                    .map(r -> new ReviewResponse(
                            r.getId(),
                            r.getRating(),
                            r.getComment(),
                            r.getCreatedAt()
                    ))
                    .collect(Collectors.toList());

    cacheService.put(cacheKey, responses, 300);
    return responses;
}
```

This cache-aside design reduces write overheads, improves scalability, and aligns more effectively with the read-heavy access patterns of the application. By combining explicit cache eviction on writes with time-based expiration, the system maintains acceptable data freshness while significantly improving performance.

Event-based cache invalidation was also considered during the design of the review caching strategy. In an event-based approach, cache entries are refreshed or invalidated in response to domain events published by the system. In the case of the local cuisine application the only meaningful domain event related to reviews occurs when a new review is created for a cuisine.

An example of an event-based cache invalidation approach would involve publishing a ReviewCreatedEvent whenever a review is persisted, and having a separate event listener respond by evicting or updating the cached reviews for the affected cuisine.

For example, a review creation event could be created as following:

```
public class ReviewCreatedEvent {
    private final Long cuisineId;

    public ReviewCreatedEvent(Long cuisineId) {
        this.cuisineId = cuisineId;
    }

    public Long getCuisineId() {
        return cuisineId;
    }
}
```

And a corresponding event listener could invalidate the cache entry:

```
@Component
public class ReviewEventListener {

    @CacheEvict(value = "reviews", key = "#event.cuisineId")
    @EventListener
    public void handleReviewCreated(ReviewCreatedEvent event) {
        // Cache is evicted in response to the event
    }
}
```

However, in this system the cache eviction already occurs directly within the review creation workflow. Because the system has a single, well-defined write path for reviews, introducing a full event-based validation mechanism would add unnecessary complexity to the application. The chosen approach achieves the same consistency guarantees while remaining simpler and easier to maintain.

5.1.1 Backend Request Workflow

Now that it is clear which decisions were made when building the backend of the project, this section introduces the backend workflow that is executed when a user makes a request to the system. It outlines the sequence of interactions between the client and the backend components, including the controller, service, repository, caching, and database layers.

The following sequence diagram shows the lifecycle of a backend request starting from a user. It shows the order in which the client interacts with the backend components and how the request processes through the controller, service, caching, repository, and database layers. This diagram focuses on a single request of a cuisine search, and highlights the responsibilities of each component involved. While the example shown uses the cuisine module, the same request flow and architectural pattern are consistently applied across other modules in the system, such as regions and reviews.

When a user initiates a cuisine search, the backend processes the request in the following order:

1. The client sends an HTTP request to the backend API.
2. The request is received by the corresponding controller.
3. The controller delegates the request to the service layer.
4. The service layer checks the cache for existing data related to the request.
5. If the requested data is found in the cache, it is returned immediately.

6. If the data is not found or the cache entry is invalid, the service queries the repository layer.
7. The repository retrieves the required data from the database.
8. The service processes the retrieved entities and maps them.
9. The processed result is stored in the cache for future requests.
10. The controller returns the final response to the client.

As shown in the sequential diagram above, the workflow begins when the user initiates a search action from the frontend interface. In this example, the user searches for “Hakata ramen”.

The frontend translates this action into an HTTP request and sends it to the backend API.

```
GET /api/cuisines/Hakata ramen
```

The frontend is responsible only for user interaction and request initiation, while all business logic is handled by the backend. The request is received by the CuisineController, which serves as the entry point to the backend. The controller extracts request parameters and sends the request to the service layer.

```
@RestController
@RequestMapping(”/api/cuisines”)
public class CuisineController {

    private final CuisineService cuisineService;

    public CuisineController(CuisineService cuisineService) {
        this.cuisineService = cuisineService;
    }

    @GetMapping(”/byRegion/{regionName}”)
    public List<String> getByRegion(@PathVariable String regionName) {
        return cuisineService.getCuisinesByRegion(regionName);
    }
    @GetMapping(”/{cuisineName}”)
   public ResponseEntity<CuisineResponse> getByCuisineName(@PathVariable String cuisineName) {
        return ResponseEntity.ok(cuisineService.getCuisineByName(cuisineName));
    }
}
```

The controller contains no business logic, ensuring a clear separation of the layers.

The CuisineService contains the business logic for retrieving a cuisine by name. It first checks the cache using the cache-aside pattern.

```
public CuisineResponse getCuisineByName(String cuisineName) {

    String cacheKey = ”cuisine:name:” + cuisineName.toLowerCase();

    // 1. Check cache
    Object cached = cacheService.get(cacheKey);
    if (cached != null) {
        CuisineResponse response =
            objectMapper.convertValue(cached, CuisineResponse.class);
        System.out.println(”cache hit”);
        return response;
    }

    System.out.println(”cache miss”);
}
```

If the cache contains a valid entry, it is converted into a CuisineResponse using ObjectMapper, which is a uitility class of a Java library used for mapping Java objects and JSON-compatible structures. In this contex, it transforms the cached generic object retrieved from Redis back to the CuisineRepsonse type. This will be explained more in detailed in the next step. This avoids unnecessary database access and reduces response time.

If the cache does not contain the requested cuisine, the service queries the database through the repository:

```
Cuisine cuisine = cuisineRepository
        .findByCuisineNameIgnoreCase(cuisineName)
        .orElseThrow(() -> new RuntimeException(”Cuisine not found”));

CuisineResponse response = CuisineResponse.from(cuisine);
```

Here, the repository retrieves the Cuisine entity by name. If there are no matching entity, an exception is thrown and and propagated back to the controller. The controller handles the exception and returns an appropriate HTTP error response to the frontend, typically a 404 Not Found status with a descriptive error message. The error message is sent to the frontend, and the user is informed them that the requested cuisine could not be found. If the cuisine data is found in the database that matches the cuisine name, the response is converted into a CuisineResponse object, which is an object that is responsible for transferring data to the frontend.

```
cacheService.put(cacheKey, response, 3600);

return response;
```

The response is cached with a time invalidation of 3600 seconds (1 hour) in Redis. Finally, the controller returns the CuisineResponse to the frontend, which displays it to the user. Below is the full FindCuisineByName method.

```
public CuisineResponse getCuisineByName(String cuisineName) {

    String cacheKey = ”cuisine:name:” + cuisineName.toLowerCase();

    // 1. Check cache
    Object cached = cacheService.get(cacheKey);
    if (cached != null) {
        CuisineResponse response =
            objectMapper.convertValue(cached, CuisineResponse.class);
    System.out.println(”cache hit”);
        return response;
    }

    System.out.println(”cache miss”);

    // 2. Cache miss → original logic
    Cuisine cuisine = cuisineRepository
            .findByCuisineNameIgnoreCase(cuisineName)
            .orElseThrow(() -> new RuntimeException(”Cuisine not found”));

    CuisineResponse response = CuisineResponse.from(cuisine);
    // 3. Store result in cache
    cacheService.put(cacheKey, response, 3600);

    return response;
}
```

5.1.2 Modular Architecture Overview

In the previous subsection, we explored the workflow for a backend request. This section will introduce indepth what happens how the Java classes interact with each other to enable those workflows as a part of the modular monolithic architecture.

The following UML class diagram provides a visual representation of the structural organization of the local cuisine application. It illustrates how classes are organized within the system and how they interact with one another. The diagram highlights the relationships between different classes and modules, and clarify the design and the flow of responsibilities across the application.

Different types of arrows in the diagram represent specific relationships between classes. The six primary UML relationships and their corresponding arrow notations are as follows:

- **Inheritance (Generalization):** Solid line with a hollow triangle pointing to the parent class.
- **Realization (Implementation):** Dashed line with a hollow triangle pointing to the interface.
- **Composition:** Solid line with a filled diamond at the owning class. Represents a ”has-a” relationship with strong ownership, where the contained object’s lifecycle depends on the owning class.
- **Aggregation:** Solid line with a hollow diamond at the owning class. Indicates a weaker ”has-a” relationship.
- **Association:** Solid line connecting two classes, representing a structural relationship.
- **Dependency:** Dashed arrow pointing from the dependent class to the class it relies on, indicating a temporary or unidirectional usage.

In this explanation, the meaning of these relationships will be clarified further in context as each module and its layered structure is discussed.

Overall System Structure

The local cuisine application is organized into three main modules, and each of them are responsible for a distinct aspect of the system. Although these modules are logically separated, they are implemented within a single deployable Spring Boot application, forming a modular monolith.

Each module follows a layered architecture, which promotes separation of concerns and maintainability. This design allows the modules to evolve independently while sharing common resources, such as the database and system configuration. The layered structure ensures a clear flow of responsibilities, making the system easier to understand and maintain.

The layered architecture within each module organizes classes into four primary layers: Controller, Service, Repository, and Entity. These layers clearly separate responsibilities and facilitate maintainability. The UML class diagram illustrates these layers and the relationships between the classes, with arrows indicating dependencies and structural associations.

Controller Layer

The controller layer handles HTTP requests and maps endpoints to the appropriate service methods Each controller is composed of service objects, which it depends on to execute business logic. In UML, this relationship is represented by a composition arrow (`*-`) pointing from the controller to the service class.

For example, in the Review module, the ReviewController has a composition relationship with ReviewService. This is implemented in Java via constructor injection: the controller receives a reference to a ReviewService instance when it is created. This means that the controllers owns the reference to their corresponding service object for its lifetime and relies on it to handle all requests.

```
// ReviewController.java

@RestController
@RequestMapping(”/api/reviews”)
public class ReviewController {

    private final ReviewService reviewService;

    // Constructor injection
    public ReviewController(ReviewService reviewService) {
        this.reviewService = reviewService;
    }
    @GetMapping("/cuisine/{cuisineId}")
    public List<ReviewResponse> getReviewsByCuisine(@PathVariable Long cuisineId) {
        return reviewService.getReviewsByCuisine(cuisineId);
    }
}
```

For instance, when a user makes a GET request to view reviews for a certain cuisine, the ReviewController invokes the getReviewsByCuisine() method on its ReviewService object. The controller does not contain any business logic itself; it passes all such responsibilities to the service. Because the service object is initialized along with the controller and maintained as a private final field, the UML composition arrow accurately reflects the "has-a" relationship between the controller and the service.

For the reviews module, the ReviewController interacts with data transfer objects (DTOs) for handling input and output. DTOs are simple objects that carry data between layers of the application without containing business logic. They are necessary to decouple the internal domain entities from external API representations, ensuring that changes to the database model do not directly affect the API contract.

For example, a DTO used in the Review module is the CreateReviewRequest, which encapsulates the data needed to create a new review:

```
// CreateReviewRequest.java

public class CreateReviewRequest {

    @NotNull
    private Long cuisineId;

    @NotNull
    @Min(1)
    @Max(5)
    private Integer rating;

    @Size(max = 1000)
    private String comment;

    public Long getCuisineId() { return cuisineId; }
    public void setCuisineId(Long cuisineId) { this.cuisineId = cuisineId; }


    public Integer getRating() { return rating; }
    public void setRating(Integer rating) { this.rating = rating; }


    public String getComment() { return comment; }
    public void setComment(String comment) { this.comment = comment; }
}
```

This DTO is used by the controller to receive data from the client in a structured format. It ensures that only valid and necessary data is passed to the service layer, allowing the internal entity objects, such as the Review entity, to remain decoupled from the API contract. For example, the @NotNull annotation ensures that none of the fields of the submitted data isn't empty.

This class does not contain any business logic. It simply holds the data returned by the service layer. It is used by the ReviewController to send responses back to clients.

In the Review module, the ReviewController depends on two DTOs: CreateReviewRequest and ReviewResponse. When a client sends a POST request to create a new review, the incoming JSON payload is mapped to a CreateReviewRequest object. The controller then passes this object to the ReviewService for processing. After the service completes the operation, it returns a ReviewResponse object, which the controller sends back to the client.

```
// ReviewController.java - POST example

@PostMapping
public ReviewResponse createReview(@RequestBody @Valid CreateReviewRequest request) {
    // The controller assigns the processing to the service,
    // passing the DTO object along.
    return reviewService.createReview(request);
}
```

This dependency is short-lived. The controller uses the DTO objects only during the handling of each request. Because the controller does not own these objects for its entire lifetime, UML represents this relationship with a dependency arrow rather than a composition arrow.

Service Layer

The service layer contains the core business logic of the application. Services process requests from controllers, enforce business rules, and prepare data to be returned. All three modules' services maintain composition relationships with their respective repository interfaces to perform database operations.

```
// CuisineService.java - composition with repositories

@Service
public class CuisineService {

    private final CuisineRepository cuisineRepository;
    private final CacheService cacheService;
    private final ObjectMapper objectMapper;

    // Constructor injection establishes the composition relationship
    public CuisineService(CacheService cacheService,
                          CuisineRepository cuisineRepository,
                          ObjectMapper objectMapper) {
        this.cuisineRepository = cuisineRepository;
        this.cacheService = cacheService;
        this.objectMapper = objectMapper;
    }
}
```

In this example, CuisineService maintains a composition relationship with CuisineRepository This means that the service objects owns the reference to the corosponding repository object for its lifetime and relies on it for any database interactions. These objects are injected via the constructor and exist for the lifetime of the service, reflecting strong ownership in UML. Additionally, CuisineService and the ReviewService also have a composition relationship with CacheService to manage caching of frequently accessed data.

The service layer also interacts with DTOs for input and output, similar to how controllers depend on request DTOs. For example, just as ReviewController depends on the CreateReviewRequest DTO to temporarily handle incoming data, services depend on output DTOs to return structured data to the controller. Here is a code example that demonstrates these relationships.

```
// CuisineService.java - dependency on DTO
public CuisineResponse getCuisineByName(String cuisineName) {
    Cuisine cuisine = cuisineRepository
            .findByCuisineNameIgnoreCase(cuisineName)
            .orElseThrow(() -> new RuntimeException(”Cuisine not found”));

    // Convert entity to DTO
    return CuisineResponse.from(cuisine);
}

// CuisineResponse.java - output DTO

public class CuisineResponse {

    private Long id;
    private String cuisineName;
    private String description;
    private String regionName;
    private String country;

    public CuisineResponse(Long id, String cuisineName, String description,
                           String regionName, String country) {
        this.id = id;
        this.cuisineName = cuisineName;
        this.description = description;
        this.regionName = regionName;
        this.country = country;
    }

    public static CuisineResponse from(Cuisine cuisine) {
        return new CuisineResponse(
                cuisine.getId(),
                cuisine.getCuisineName(),
                cuisine.getDescription(),
                cuisine.getRegion().getRegionName(),
                cuisine.getRegion().getCountry()
        );
    }

    public Long getId() { return id; }
    public String getCuisineName() { return cuisineName; }
    public String getDescription() { return description; }
    public String getRegionName() { return regionName; }
    public String getCountry() { return country; }
}
```

Unlike composition, the service does not maintain a permanent reference to the DTO object. It only uses the DTO briefly during method execution. In this example, the DTO is only used during when there is a get request from a client to get a specific cuisine by name from the controller. These relationships are represented in UML by dependency arrows, indicating temporary or usage-based relationships rather than ownership.

The CuisineResponse DTO is specifically responsible for carrying cuisine data from the service layer to the controller layer. It encapsulates all the fields the API should expose: id, cuisineName, description, regionName, and country. This allows the service to return structured data to the controller without exposing the internal Cuisine entity directly.

The CuisineResponse DTO uses the Cuisine entity to send a respond back to the Cuisine Controller. The static from() method in CuisineResponse takes a Cuisine entity as input, reads its fields (id, cuisineName, description, and region), and constructs a new DTO object. Once the DTO is created and returned to the service or controller, the service no longer maintains a reference to the entity. Because CuisineResponse only temporarily depends on the Cuisine entity during object creation, UML represents this relationship with a dependency arrow rather than a composition.

Repository Layer

The repository layer is responsible for handling database access. Repositories provide query methods for retrieving and storing entity objects, but they do not contain business logic. Their sole responsibility is to abstract data access from the service layer. The CuisineRepository defines several query methods for retrieving Cuisine entities from the database:

```
// CuisineRepository.java

public interface CuisineRepository extends JpaRepository<Cuisine, Long> {
    List<Cuisine> findByRegion_RegionName(String regionName);
    List<Cuisine> findByRegion_Country(String country);

    Optional<Cuisine> findByCuisineNameIgnoreCase(String cuisineName);

}
```

These repository methods are invoked within the service layer to implement business logic. For example, the getCuisinesByRegion() method in CuisineService calls findByRegion_RegionName() to retrieve the corresponding entities:

```
// CuisineService.java


public List<String> getCuisinesByRegion(String regionName) {


    List<Cuisine> cuisines =

        cuisineRepository.findByRegion_RegionName(regionName);


     List<String> cuisineNames = cuisines.stream()

         .map(Cuisine::getCuisineName)

         .collect(Collectors.toList());


    return cuisineNames;


}
```

We can see that the findByRegion_RegionName method in the CuisineRepository is invoked within the Cuisine-Service as part of the composition relationship between the two classes. This demonstrates how the service depends on the repository to access persistent data, while still encapsulating the business logic and data transformation.

Similarly, ReviewService also has a composition relationship with CuisineRepository. This is because creating or retrieving reviews requires access to the associated Cuisine entity. For instance, when a new review is created, the service must first retrieve the corresponding cuisine from the database before associating it with the review. Therefore, both services depend on the same repository, but the repository itself remains independent of them.

Entities

The entity layer defines the application's persistent data model. Each entity maps directly to a database table and specifies the fields, relationships, and constraints that determine how data is stored and structured.

In the UML diagram, the entities are connected to reflect their relationships within the domain model. The Region class has an aggregation relationship with the Cuisine class, indicating that a region may contain multiple cuisines. However, the cuisines can conceptually exist independently of the region's lifecycle. The multiplicity 1 near Region and 0..* near Cuisine in the UML diagram shows that each cuisine must belong to exactly one region, while a region may have zero or more cuisines. Since a cuisine can exist without a region, there this is aggregation relationship.

In the code, this is implemented using the `@OneToMany` annotation in the Region entity:

```
// Region.java
@OneToMany(mappedBy = "region", cascade = CascadeType.ALL, fetch = FetchType.LAZY)
private List<Cuisine> cuisines;
```

This code reflects the UML diagram's relationships, showing how the Region entity contains multiple Cuisine entities while each Cuisine references its owning Region.

The Cuisine entity represents a specific type of food and maps to the cuisines table. Each cuisine has an id, cuisineName, description, and a reference to its associated region:

```
// Cuisine.java
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "region_id", nullable = false)
private Region region;
```

This establishes the many-to-one side of the region-cuisine relationship, allowing the cuisine entity to reference the region entity. In the UML diagram, the Cuisine entity has a composition relationship with the Review entity. Although the Cuisine entity does not explicitly maintain a collection of review objects, the relationship reflects a strong lifecycle dependency. Each review must be associated with exactly one cuisine, and a review cannot exist independently. The multiplicity 1 near Cuisine and 0..* near Review indicates that a cuisine may have zero or more reviews, while each review belongs to exactly one cuisine. This representation captures the ownership and domain-level dependency between cuisines and reviews.

The Review entity saves user feedback on a cuisine and maps to the reviews table. Each review has an id, a rating, a comment, and a timestamp. Each review is linked to a specific cuisine:

```
// Review.java
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "cuisine_id", nullable = false)
private Cuisine cuisine;
```

Together, these entities form a hierarchical structure in which a Region can contain multiple Cuisine objects, representing a one-to-many aggregation. Each Cuisine belongs to a single Region and form a many-to-one composition relationship. Cuisines can also have multiple Review objects, indicating another one-to-many composition. Each Review, belongs to a single Cuisine, and has the many-to-one composition relationship. These relationships maintain integrity in the database and are accurately reflected in the UML diagram through the use of composition and aggregation arrows.

Overall, the UML class diagram provides a clear visualization of the entire backend architecture. It shows how controllers, services, repositories, DTOs, and entities interact with each other. It also highlights the ownership and dependency relationships between layers, such as the composition between services and repositories, the temporary dependencies on DTOs, and the aggregation and composition among entities. By examining the diagram with the code examples, it becomes easier to understand the flow of data through the application, the separation of concerns across layers, and how each component contributes to the overall functionality of the local cuisine system.

5.2 Frontend Design

The frontend of the local cuisine application is responsible for accurately displaying the data sent from the backend. It is only responsible for presenting the data and managing user interactions, while all the business logic and data persistence are handled by the backend. The tech stack used for the frontend development are Next.JS and Tailwind css.

Next.js was selected as the primary frontend framework primarily for its strong support of modular, component-based development. As a React-based framework, it enables the application to be structured as a collection of independent, reusable components, each responsible for a specific portion of the user interface. This modular structure improves maintainability, as changes to one component can be made with minimal impact on others, and enhances scalability as new features can be added without restructuring the entire system.

The architectural philosophy of Next.js aligns closely with the modular backend implemented using Spring Boot. Just as the backend is organized into clearly separated modules and layers, the frontend is structured into distinct components and pages with well-defined responsibilities. For example, the frontend is divided into two primary modules: regions and cuisines, each responsible for handling its respective presentation logic and user interactions. This structure reflects the modular monolithic architecture of the backend, where related functionalities are grouped into modules while remaining part of a single deployable system. This design allows the similar domain across both layers, as the application maintains a consistent architectural model throughout the stack.

Both technologies promote separation of concerns and low coupling between modules, resulting in a coherent and maintainable system design. This alignment between frontend and backend architecture enhances overall system clarity and simplifies future extension, testing, and maintenance.

5.2.1 Frontend Architecture

The frontend of the application is organized in a modular structure that mirrors the domain model of the backend. At the top level, the application is divided into two primary modules: regions and cuisines, each corresponding to a distinct functional area of the application. These separate directories in Next.js automatically define the routing structure, simplifying navigation and maintaining a clear correspondence between the file system and URL paths. For example, the folder app/cuisines/[cuisineName]/page.tsx automatically corresponds to the route /cuisines/cuisineName, where cuisineName is a dynamic segment representing the name of a specific cuisine. This structure allows the application to generate pages for each cuisine dynamically while preserving an intuitive and consistent URL hierarchy.

The regions module is responsible for presenting regional data and navigating between different regions, while the cuisines module handles the display of specific cuisines and their associated interactions. All logic related to reviews is included within the cuisines module, as reviews are inherently tied to individual cuisines. This ensures that each review is associated with its respective cuisine, maintaining a coherent modular structure. Consequently, the cuisines module encapsulates both the presentation of cuisine details and the user interactions related to reviews. This is because each review as always associated with a certain cuisine.

Within each module, the application distinguishes between page components and reusable UI components. For example, the cuisines/[cuisineName] folder contains page-level components such as page.tsx, which serves as the base page for rendering the main cuisine information, along with supporting components like ReviewForm.tsx and ReviewList.tsx, which are attached to page.tsx to encapsulate specific UI functionalities. Similarly, the regions/[region] folder contains CuisineList.tsx, a reusable component for displaying a list of cuisines within a region. This separation allows components to be developed and maintained independently, enhancing code readability.

The application also employs a centralized API abstraction through the lib/api.ts file, which handles all communication with the backend REST endpoints. This approach ensures that data fetching logic is decoupled from presentation components, further reinforcing the separation of concerns. Global styling and layout are managed through globals.css and layout.tsx, providing consistent styling and structure across pages without introducing unnecessary complexity in individual components.

5.2.2 API Communication

The frontend communicates with the backend exclusively through RESTful APIs exposed by the Spring Boot application. All HTTP requests are centralized in the lib/api.ts file, which abstracts network communication and ensures that individual components remain focused on presentation and user interaction.

This centralized API abstraction offers several advantages. First, it ensures that page components and reusable UI components do not directly manage network requests; they simply invoke the relevant function from api.ts and receive structured JSON data. Second, it improves maintainability: if a backend endpoint changes, only the corresponding function in api.ts needs to be updated. Third, it facilitates consistent error handling, as all functions check response statuses and log errors appropriately

A concrete example of this abstraction can be seen in the function getCuisineByName(cuisineName) (Listing 5.1), which retrieves detailed information about a specific cuisine:

Listing 5.1: getCuisineByName – fetching a single cuisine by name

```
 1  export async function getCuisineByName(cuisineName: string) {
 2    try {
 3      const res = await
          fetch(‘${API_BASE_URL}/api/cuisines/${encodeURIComponent(cuisineName)}‘,
          { credentials: ”include” });
 4      if (!res.ok) { throw new Error(‘Failed to fetch cuisine:
          ${cuisineName}‘); }
 5      const data = await res.json();
 6      return data;
 7    } catch (error) {
 8      console.error(”Error fetching cuisine:”, error);
 9      return null;
10    }
11  }
```

In this example, the function constructs the URL dynamically using the provided cuisineName parameter (line 3), performs a fetch request including credentials, checks for a successful response (line 4), and returns the JSON data (line 6). Any errors during the request are caught and logged (lines 7–9), ensuring consistent error handling throughout the frontend. The getCuisineByName function is called in the page.tsx in the cuisines module to load data related to a specific cuisine. The data returned includes the cuisine's unique identifier (id), the name of the cuisine (cuisineName), a textual description (description), the region it belongs to (regionName), and the country (country). This information is used by the cuisines module to render the main content of the page, displaying the cuisine’s name, description, and region details to the user. Additionally, the cuisine id (id) is passed to the getReviewsByCuisineId function to fetch all reviews associated with this specific cuisine, ensuring that the displayed reviews are correctly linked to the selected cuisine and maintaining consistency between the main page content and the review components.

Similarly, creating a new review is abstracted through the createReview(data) function (Listing 5.2), which sends a POST request with a JSON payload:

Listing 5.2: createReview – sending a new review to the backend

```
1  export async function createReview(data: { cuisineId: number; rating: number;
       comment: string; }) {
2    const response = await fetch(
3      ‘${API_BASE_URL}/api/reviews‘,
4      {
5        method: ”POST”,
6        headers: { ”Content−Type”: ”application/json” },
7        body: JSON.stringify(data),
8      }
9    );
10   if (!response.ok) { throw new Error(”Failed to create review”); }
11   return response.json();
12 }
```

This function takes a structured data object containing the cuisineId, rating, and comment (line 1), serializes it to JSON (line 5), and sends it to the backend (lines 2–6). It also checks for errors (line 7) and returns the response JSON (line 8). By using these abstractions, the frontend components remain clean and focused on rendering, while all network communication and error handling is handled in a single, maintainable location.

In the application, the createReview function is invoked within the ReviewForm.tsx component of the cuisines module. When a user submits a new review through the form, the component collects the rating and comment values, along with the cuisineId from the currently displayed cuisine, and passes them to createReview. This ensures that the review is correctly associated with the selected cuisine and immediately reflected in the ReviewList component, maintaining a smooth user experience and a consistent connection between the cuisine data and its associated reviews.

This clear separation between data fetching and presentation reinforces the modular architecture of the frontend and maintains consistency with the backend’s modular monolithic structure. By centralizing all network requests within api.ts, page and UI components focus solely on rendering and user interactions, while all communication with the backend and error handling is encapsulated in a single location. This approach enhances maintainability, ensures consistent data usage across components, and aligns the frontend and backend architectures, resulting in a coherent and extensible system design.

5.2.3 Client-Side Caching

As discussed in Section 4.4 (Make reference), client-side caching can significantly improve performance by reducing redundant network requests, and enable offline usability. However, since this application is primarily intended to run in an online environment with continuous communication between the frontend and backend, advanced frontend caching mechanisms were not fully implemented in the current version. Instead, the system relies on default browser caching behavior, while remaining open for future integration of additional client-side caching strategies.

The primary form of caching currently utilized in this project is the browser's automatic caching of static frontend assets. When users access the application for the first time, the compiled Next.js JavaScript bundles (e.g., files under /_next/static/) and the generated CSS stylesheet produced by Tailwind CSS are downloaded and stored in the browser cache. Because these assets are static and shared across multiple pages, they do not need to be re-downloaded during subsequent navigation. For example, when a user navigates from the region listing page to a cuisine detail page, the previously loaded JavaScript bundle and stylesheet are reused from the browser cache. Only the dynamic data retrieved from the backend API needs to be fetched again. This reduces loading time and improves responsiveness without requiring additional manual configuration.

Although local storage was not directly implemented in the current version of the application, the existing frontend architecture allows for its straightforward integration. As described in Section 4.4, local storage enables persistent storage of lightweight, non-sensitive data on the client device across browser sessions. Local storage could be used to retain user interface preferences in the future, such as theme selection, or to store recently viewed cuisines. It could also be applied to cache infrequently changing API responses in order to reduce repeated network requests.

In addition to browser-level caching, HTTP Cache-Control headers could be configured on the backend to provide more control over resource freshness. For example, relatively stable data such as region or cuisine listings could be assigned appropriate caching directives (e.g., max-age) to allow temporary reuse without repeated server requests. However, data that could change frequently such as user reviews would require stricter caching policies to ensure data consistency and accuracy. Although these cache-control strategies were not explicitly configured in the current implementation, the system architecture supports their future integration. This reflects a design consideration that balances performance optimization with data freshness, as discussed in Section 4.4.

5.2.4 User Interface Design Choices

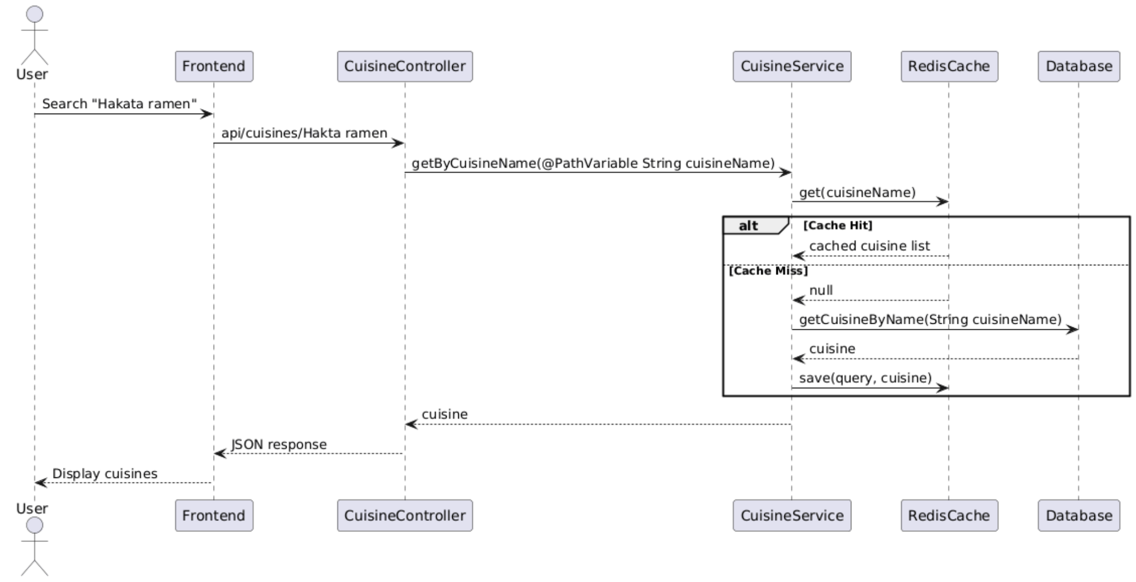

Figure 5.2: Figure of the backend workflow sequential diagram for the local cuisine application.

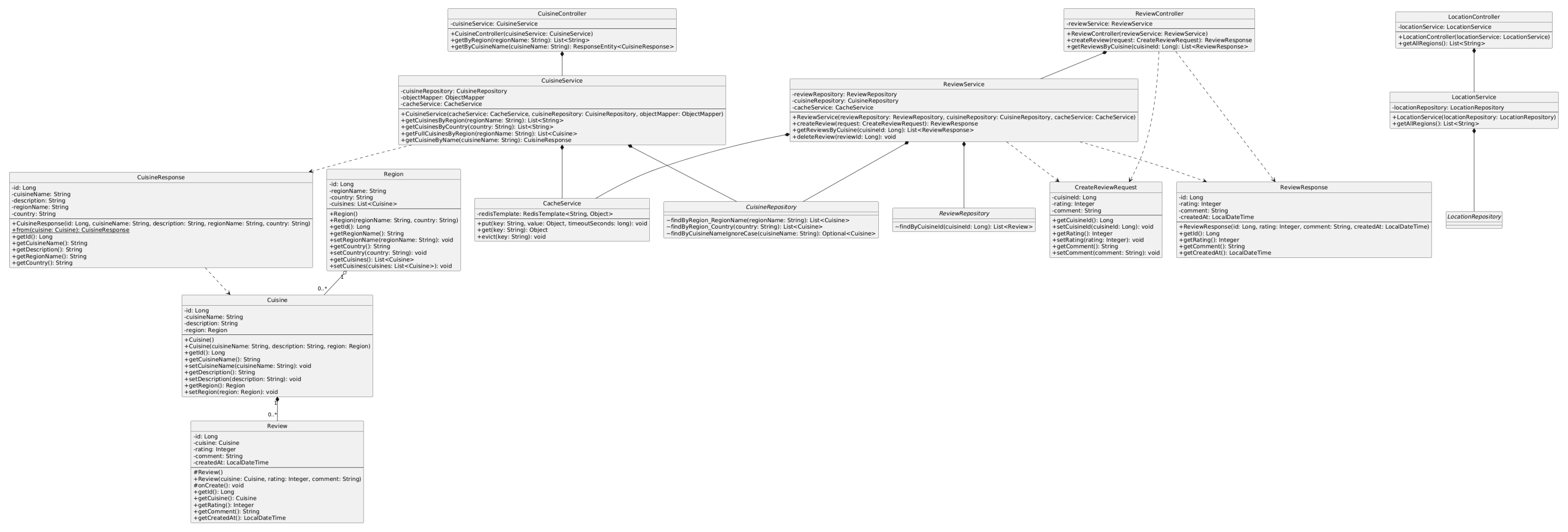
CuisineController
-cuisineService: CuisineService
+CuisineController(cuisineService: CuisineService)
+getByRegion(regionName: String): List<String>
+getByCuisineName(cuisineName: String): ResponseEntity<CuisineResponse>
CuisineService
-cuisineRepository: CuisineRepository
-objectMapper: ObjectMapper
-cacheService: CacheService
+CuisineService(cacheService: CacheService, cuisineRepository: CuisineRepository, objectMapper: ObjectMapper)
+getCuisinesByRegion(regionName: String): List<String>
+getCuisinesByCountry(country: String): List<String>
+getFullCuisinesByRegion(regionName: String): List<Cuisine>
+getCuisineByName(cuisineName: String): CuisineResponse
CuisineResponse
-id: Long
-cuisineName: String
-description: String
-regionName: String
-country: String
+CuisineResponse(id: Long, cuisineName: String, description: String, regionName: String, country: String)
+from(cuisine: Cuisine): CuisineResponse
+getId(): Long
+getCuisineName(): String
+getDescription(): String
+getRegionName(): String
+getCountry(): String
Region
-id: Long
-regionName: String
-country: String
-cuisines: List<Cuisine>
+Region()
+Region(regionName: String, country: String)
+getId(): Long
+getRegionName(): String
+setRegionName(regionName: String): void
+getCountry(): String
+setCountry(country: String): void
+getCuisines(): List<Cuisine>
+setCuisines(cuisines: List<Cuisine>): void
CacheService
-redisTemplate: RedisTemplate<String, Object>
+put(key: String, value: Object, timeoutSeconds: long): void
+get(key: String): Object
+evict(key: String): void
CuisineRepository
~findByRegion_RegionName(regionName: String): List<Cuisine>
~findByRegion_Country(country: String): List<Cuisine>
~findByCuisineNameIgnoreCase(cuisineName: String): Optional<Cuisine>
1
0..*
Cuisine
-id: Long
-cuisineName: String
-description: String
-region: Region
+Cuisine()
+Cuisine(cuisineName: String, description: String, region: Region)
+getId(): Long
+getCuisineName(): String
+setCuisineName(cuisineName: String): void
+getDescription(): String
+setDescription(description: String): void
+getRegion(): Region
+setRegion(region: Region): void
1
0..*
Review
-id: Long
-cuisine: Cuisine
-rating: Integer
-comment: String
-createdAt: LocalDateTime
#Review()
+Review(cuisine: Cuisine, rating: Integer, comment: String)
#onCreate(): void
+getId(): Long
+getCuisine(): Cuisine
+getRating(): Integer
+getComment(): String
+getCreatedAt(): LocalDateTime
ReviewController
-reviewService: ReviewService
+ReviewController(reviewService: ReviewService)
+createReview(request: CreateReviewRequest): ReviewResponse
+getReviewsByCuisine(cuisineId: Long): List<ReviewResponse>
ReviewService
-reviewRepository: ReviewRepository
-cuisineRepository: CuisineRepository
-cacheService: CacheService
+ReviewService(reviewRepository: ReviewRepository, cuisineRepository: CuisineRepository, cacheService: CacheService)
+createReview(request: CreateReviewRequest): ReviewResponse
+getReviewsByCuisine(cuisineId: Long): List<ReviewResponse>
+deleteReview(reviewId: Long): void
ReviewRepository
~findByCuisineId(cuisineId: Long): List<Review>
CreateReviewRequest
-cuisineId: Long
-rating: Integer
-comment: String
+getCuisineId(): Long
+setCuisineId(cuisineId: Long): void
+getRating(): Integer
+setRating(rating: Integer): void
+getComment(): String
+setComment(comment: String): void
ReviewResponse
-id: Long
-rating: Integer
-comment: String
-createdAt: LocalDateTime
+ReviewResponse(id: Long, rating: Integer, comment: String, createdAt: LocalDateTime)
+getId(): Long
+getRating(): Integer
+getComment(): String
+getCreatedAt(): LocalDateTime
LocationController
-locationService: LocationService
+LocationController(locationService: LocationService)
+getAllRegions(): List<String>
LocationService
-locationRepository: LocationRepository
+LocationService(locationRepository: LocationRepository)
+getAllRegions(): List<String>
LocationRepository

Figure 5.3: Figure of the Java class relationship diagram for the local cuisine application.

CHAPTER 6

Results

CHAPTER 7

Challenges and Lessons Learned

7.1 What worked well

7.2 What did not work well

References

[1] Kamran Ahmed. *Backend Developer Roadmap*. 2025. url: https://roadmap.sh/backend (page 1).

[2] Tom Barker. *Intelligent caching*. O'Reilly Media, Inc, 2017.

[3] Domenico Bianculli et al. "Assessing the Impact of Asynchronous Communication on Resilience and Robustness: A Comparative Study of Microservice and Monolithic Architectures". In: *Software Architecture. ECSA 2025 Tracks and Workshops*. Cham: Springer Nature Switzerland, 2026, pp. 171–186. isbn: 978-3-032-04403-7.

[4] Prime Career. *Is Java a Must-Have? 5 Top Backend Job Languages for 2025: Backend Engineer: Prime Career*. url: https://prime-career.com/article/11242 (page 38).

[5] Josiah L. Carlson. *Redis in action*. Manning, 2013 (page 34).

[6] Raju Gandhi, Mark Richards, and Neal Ford. *Head first software architecture*. O'Reilly, 2024 (pages 4–6, 9).

[7] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database systems: The complete book*. Pearson Education Limited, 2014 (page 25).

[8] Jan L. Harrington. *Relational database design and implementation*. Morgan Kaufmann/Elsevier, 2016 (pages 13–14).

[9] Jacky. *Solving the notorious N+1 problem: Optimizing database queries for Java backend developers*. Sept. 2023. url: https://dev.to/jackynote/solving-the-notorious-n1-problem-optimizing-database-queries-for-java-backend-developers-2o0p (page 27).

[10] Martin Kleppmann. *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*. O'Reilly, 2017 (pages 18–21, 23).

[11] Redis. *Cache invalidation*. Aug. 2025. url: https://redis.io/glossary/cache-invalidation/ (pages 36–37).

[12] Amazon Web Services. *Relational vs Nonrelational Databases - Difference Between Types of Databases - AWS*. What's the Difference Between Relational and Non-relational Databases? n.d. url: https://aws.amazon.com/compare/the-difference-between-relational-and-non-relational-databases/ (pages 11–12, 26).

[13] Supabase. *Database replication: Supabase docs*. Feb. 2026. url: https://supabase.com/docs/guides/database/replication (page 47).

[14] Maharani Thiraviyam. *What is caching strategies in DBMS?* July 2025. url: https://www.geeksforgeeks.org/dbms/what-is-caching-strategies-in-dbms/ (pages 31–34).

[15] Alexi Turcotte, Mark W. Aldrich, and Frank Tip. "reformulator: Automated Refactoring of the N+1 Problem in Database-Backed Applications". In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. ASE '22. Rochester, MI, USA: Association for Computing Machinery, 2023. isbn: 9781450394758. doi: 10.1145/3551349.3556911. url: https://doi.org/10.1145/3551349.3556911.