UNIVERSITÀ DI PISA

LARGE-SCALE AND MULTI-STRUCT DATABASES PROJECT

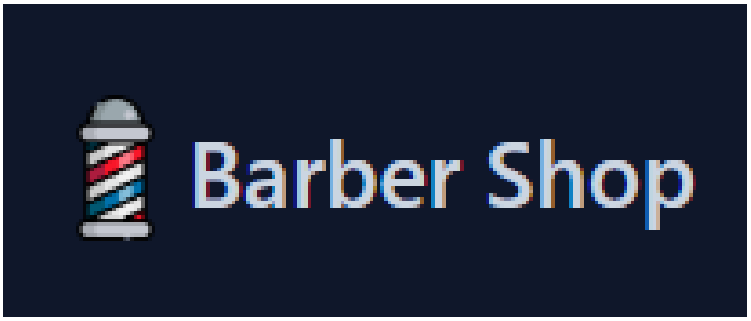
BARBERSHOP

Andrea Bedini
Edoardo Geraci
Alessandro Versari

Index

1. Introduction **4**
2. Feasibility Study **5**
2.1. Data Sources 5
3. Design **7**
3.1. Main Actors 7
3.2. Functional Requirements 8
3.3. Non-Functional Requirements 9
3.4. CAP Theorem Issue 10
3.5. Use Cases 11
3.6. Database Design UML 12
3.7. Data Model 13
3.7.1. MongoDB Collections 14
3.7.2. Redis Namespace and Keys 17
3.8. Database Replication 18
3.9. Sharding Proposal 20
3.10. Platform, Technologies and Architecture 21
4. Implementation **23**
4.1. Frontend 24
4.1.1 Code Organization 24
4.2. Backend 27
4.2.1. File List 27
4.2.2. Packages and Architecture 32
4.2.3. External Packages/Libraries 37
4.2.4. Unit Tests 38
4.3. Scraper and Importer 39
4.4. Docker Compose 40
4.5. Database CRUD Operations 41
4.5.1. Users 41
4.5.2. Shops 45
4.5.3. Appointments 48
4.5.4. ShopViews 50
4.5.5. Reviews 51
4.6. Key-Value Database Operations 56
4.7. DocumentDB Aggregations 60
4.8. MongoDB Indexes and Performance Analysis 89
5. Conclusion **92**
5.1 Closing Words 92
5.2 Expandability 92
6. User Manual **93**

1. Introduction



The application's logo

BarberShop, as the name implies, is a platform born from the need to connect users to barber shops. Most of the time, it is hard to find and properly compare barbers near oneself. Reviews are scattered, addresses are not always consistent, and booking appointments is seldomly easy.
BarberShop offers a centralized hub to solve all these issues. It lets **Users discover barber shops**, **review** them, and even **book appointments** through a simple and streamlined UI.
Barbers can manage their **shop** info and **appointments**, while getting useful data in the form of **analytics**.
All of this is wrapped in a webapp with a sleek and modern look, thus being available on any computer without the need to install anything.

Useful Links:

- The live **homepage** of the site is available at http://172.16.5.42/
- The **endpoint documentation** can be found at http://172.16.5.42/api/swagger/index.html
- The **repository** of the project's code can be found at https://github.com/just-hms/large-scale-multistructure-db

2. Feasibility Study

Before starting to work on this project, a Feasibility Study was conducted in order to identify the **Data Sources** that would be needed, and how to best use them.

2.1. Data Sources

The data for this project comes from 2 different sources, in order to guarantee higher coverage and availability:

- **Google Maps/Places**
- **Yelp**

At first, it was considered the usage of a **web scraper**, but the idea was soon discarded due to anti-scraping measures (rate-limiting and dynamic CSS), and a possible violation of ToS.

The approach that was ultimately used is querying the sites' respective **Developer's API**. Both of these sites offer a free/trial tier of API access once an account has been created. The provided free API calls were enough to fetch all the necessary data necessary to build the Databases.

The data that was obtained from the APIs is:

- **Usernames**
- **Barbershops' data**
- **Reviews**

Both sources produced data in JSON format, albeit with different field names and structure. Thus, a **script** is needed in order to merge and homogenize the information.

An example of the scraped data from a Shop

Lastly, an **importer script** executes a number of miscellaneous tasks:

- **Producing data** that cannot be scraped (**ShopViews**, **Appointments**, Reviews' **Up/DownVotes**, **User Account Info**) due to it being very application specific. This is accomplished thanks to **Faker**, a library that generates some believable "fake" data.
- **Making adjustments** to fields where necessary (eg. preparing coordinates for MongoDB's Georadius)
- Creating the necessary MongoDB's **Collections** and **Indexes**
- **Importing** all the scraped and adjusted data into MongoDB

More informations on the process and the relevant code is available further in this document in the Chapter 4.3

3. Design

3.1. Main Actors

The application has three main actors:

- **User**
- **Barber**
- **Admin**

User: this is a basic user without any additional permission. At least a **User** account is needed in order for clients to use the website. A **User** can use this application to browse **Shops**, read and write **Reviews**, **Up/Downvote** them, and book **Appointments** at any **Shop** they want.

Barber: this user has the ability to administer one or more **Shop** pages. Each **Barber** can edit the information of any of their **OwnedShops**, see and manage the lists of **Appointments**, and get various types of Analytical data on their **Shops**.

Admin: this user is able to create the **Shop** pages and assign them to a **Barber**. The Admin can also manage user and shop profiles. Lastly, an **Admin** can view **Analytics** about the usage of the website, and have insights about the general state of the application.

3.2. Functional Requirements

The application will allow the users to perform the following operations:

Any type of user

- account related operation
    - signup
    - login
    - password recovery
- find shops
- view shop profile
    - review shop
    - up/downvote review

User

- view profile info
    - view current appointment
    - delete current appointment
- delete account
- view shop profile
    - book appointment

Barber

- view profile info
    - delete account
    - browse owned shops
        - view booked appointments
        - view shop analytics

Admin

- browse users
    - find user
    - view user
        - delete user
        - edit barber shop ownership
- browse shops
    - edit shop information
- create shops
- view app analytics

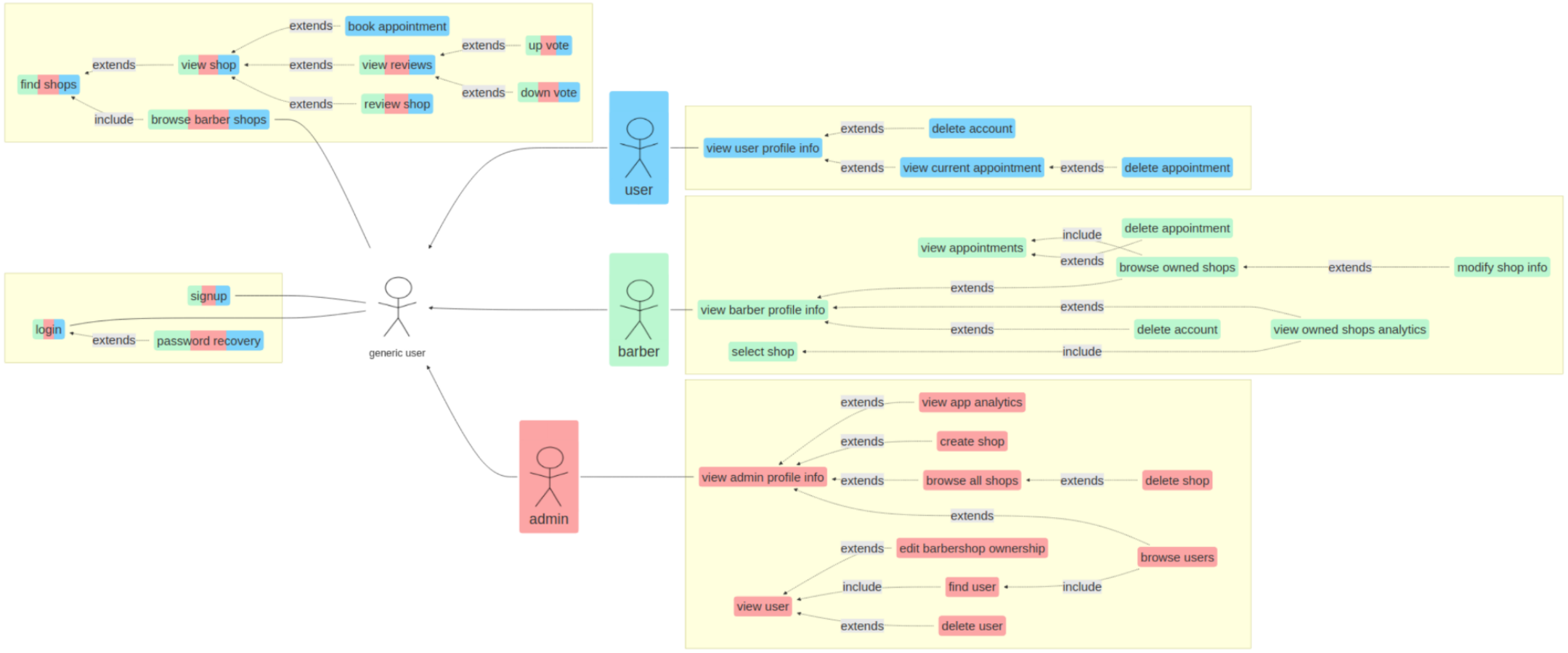
3.3. Non-Functional Requirements

- Retrieve the list of the **nearest barber shops** in a **reasonable amount of time**
- Ensure **availability** of the data and **partition tolerance**
- **Cache** the appointment's **calendar** of every barbershop to ensure **access in a short amount of time**
- Allow the user to search for places with a "**human-readable**" address/location
- Make meaningful **analytics** that might help identify issues in a shop's management
- Separate data in different collections in order to **minimize** both **data loaded** from the database and **data sent** over the network
- Produce an **easy and simple to use UI** in order to allow every type of user to easily use the application

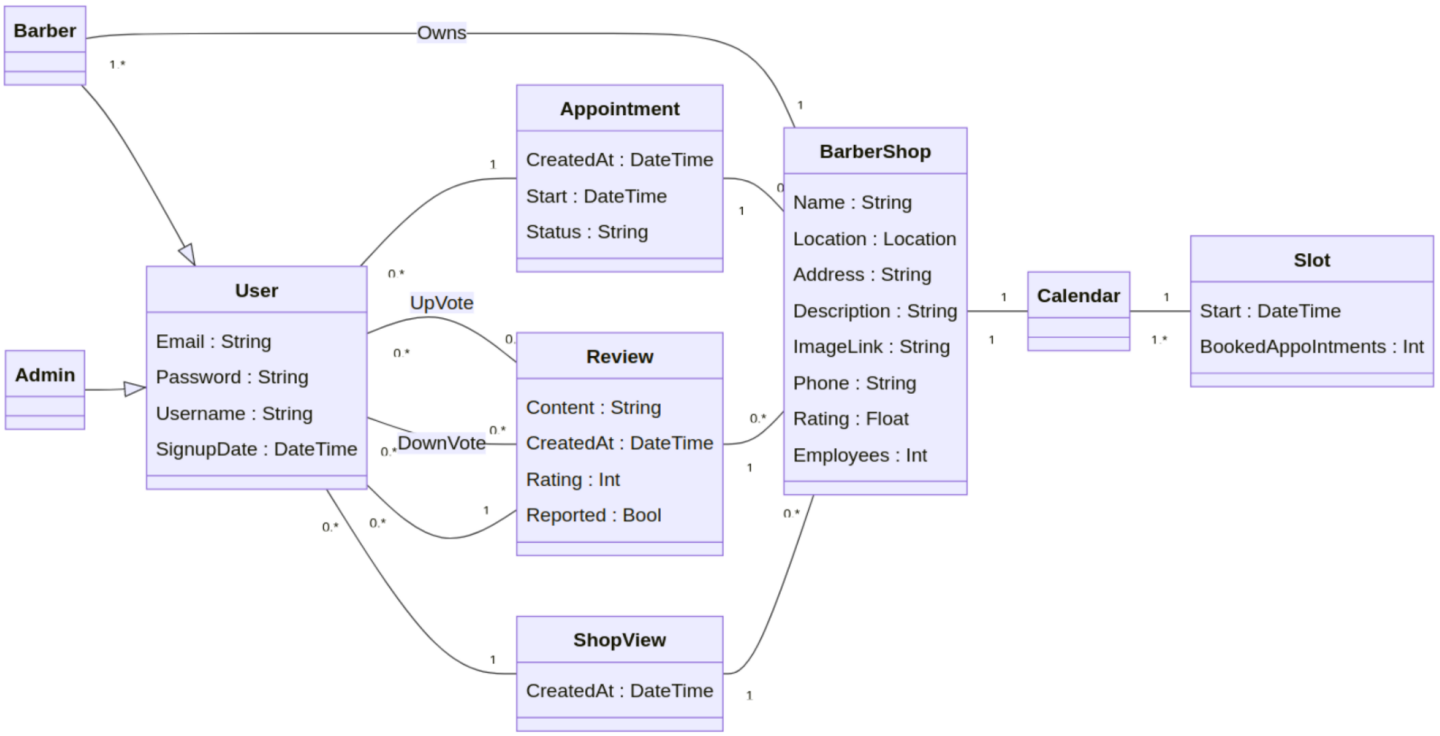
3.4. CAP Theorem Issue

To ensure optimal performance for the expected influx of read operations, it is imperative to prioritize both **high availability** and **low latency** during the application's design. Moreover, maintaining functionality even in the face of partitioning is crucial. Following the principles of the **CAP** theorem, this application's design should prioritize **Availability (A)** and **Partition Tolerance (P)** over Consistency (C). This implies that the application concentrates on ensuring continued system access and the ability to handle partitioning, rather than guaranteeing absolute data consistency.

3.5. Use Cases


extends
book appointment
extends
up vote
extends
view shop
extends
view reviews
extends
down vote
find shops
extends
review shop
include
browse barber shops
user
view user profile info
extends
delete account
extends
view current appointment
extends
delete appointment
signup
login
extends
password recovery
generic user
barber
view appointments
include
delete appointment
extends
browse owned shops
extends
modify shop info
extends
view barber profile info
extends
extends
delete account
view owned shops analytics
select shop
include
admin
extends
view app analytics
extends
create shop
view admin profile info
extends
browse all shops
extends
delete shop
extends
browse users
extends
edit barbershop ownership
include
find user
include
view user
extends
delete user

3.6. Database Design UML



3.7. Data Model

BarberShop uses 2 different DBMSes:

- **MongoDB**
- **Redis**

MongoDB is a document DB, and acts as the primary database for the platform. On it is stored all the “permanent” data used by the application. It also serves as the **Geolocation** provider.

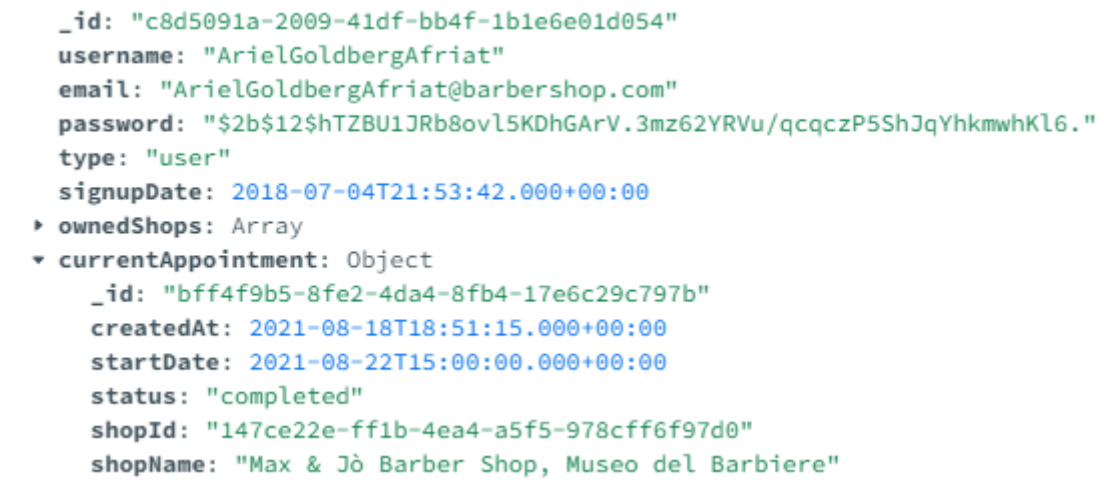
Redis is instead a key-value database. It acts as the **cache** for the **Appointments’ Calendar**. This task is well suited as:

1) **No complex data** is needed
2) It **doesn’t crossover** with other data in the other DB, unless a new Appointment has been booked
3) It allows **fast reads of a high volume of data**, which is necessary to offer a snappy and responsive experience to the user
4) It offers a **built-in expiry mechanism**, useful to automatically dispose of data no longer needed once an Appointment date has passed

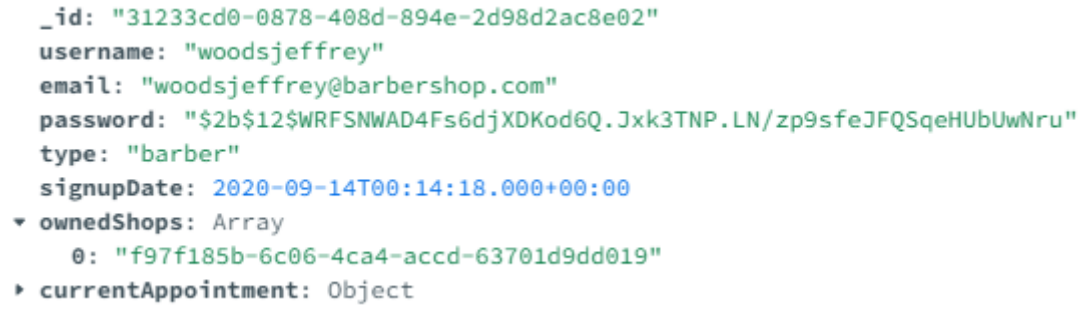
3.7.1. MongoDB Collections

BarberShop uses 5 different collections. This was done in order to **reduce the amount of unneeded fetched data**, and to better apply the necessary **Aggregations**. Data has been **replicated** in order to reduce/remove the need of multi-collection queries/Aggregations wherever was deemed necessary.

- **Users**: BarberShop has 3 different categories of users, which are distinguished by the “type” field. Each type of user has slightly different fields:



A regular User does not have any ownedShops, but may have a currentAppointment, which is replicated data of their last booked Appointment



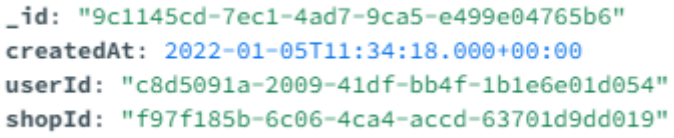
A Barber user has a list of ownedShops, which is a list of the ids of the Shops he manages

An Admin usually does not have neither ownedShops nor a currentAppointment

- **Barbershops**: the data of each Shop in the database. It comes with a link to its **profile picture**, and **Geolocation** data:

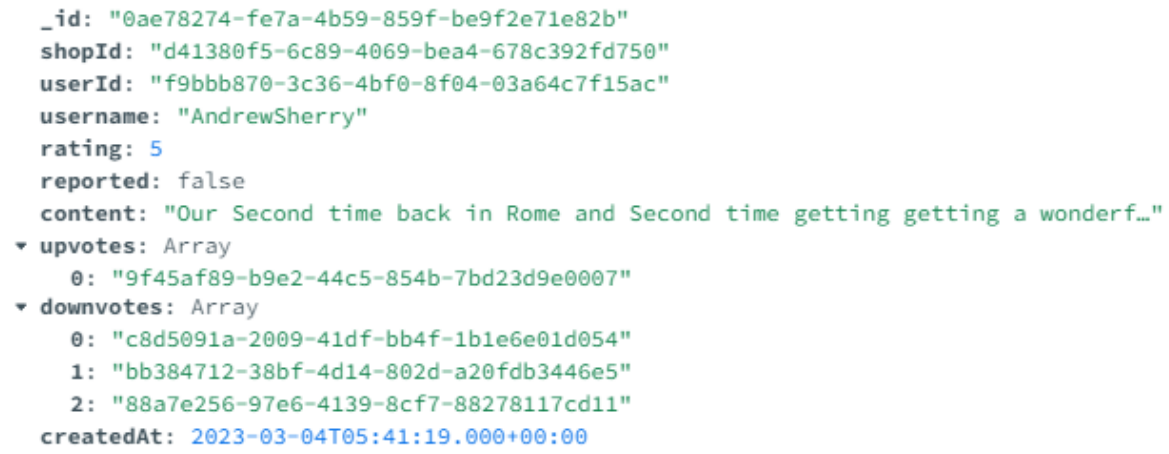
The stored data of a typical Shop

- **Shopviews**: whenever **a user opens the profile of a Shop**, a Shopview is stored in the database. This is very useful to make some **Analytics** and gauge the interest of Users on a Shop:



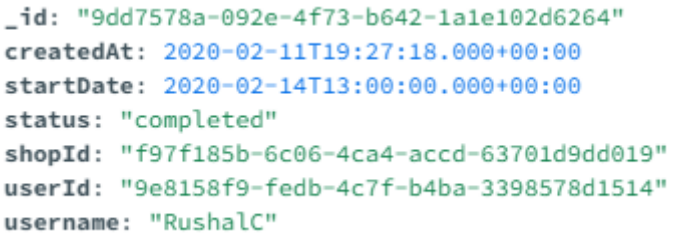
A ShopView, in which info on both the User and the visited Shop are stored

- **Reviews**: the data of each Review, across all Shops. It also contains a list of UserIDs that have **Upvoted** or **Downvoted** it. A User cannot both Upvote and Downvote a single Review:



The stored data of a typical Review, with its lists of UserIDs who have engaged with it

- **Appointments**: the data of each Appointment, across all Shops. Each appointment can be in **one of 3 statuses**: **pending**, **completed**, **canceled**. This information is useful to make **Analytics** regarding the quality of the service of each Shop:



The stored data of a completed Appointment. Notice the fact that it contains a replicated Username for display purposes.

3.7.2. Redis Namespace and Keys

Redis is used as a **cache** for the Appointments. In order to correctly display the Slots in the **Appointments' calendar**, it is necessary to have stored some data that can be quickly accessed.
Each **Redis** entry is set to **expire** a day after the Datetime associated with the Slot, as it is no longer useful.

Each **Redis key** has the format **"barbershop:<shopID>:slots:<time>"**, and is created using the following function:

```go
func key(barberShopID string, date time.Time) string {
    key := fmt.Sprintf("barbershop:%s:slots:%d", barberShopID, date.Unix())
    return key
}
```

The Golang code responsible for generating the Redis keys

Each **Redis value** consists of a small object/document. The choice of using a single key with a document instead of creating different keys for each field/property was made due to the fact that all the data of the object is usually used together.
Each **value** has the following structure:

```go
type Slot struct {
    Start             time.Time
    BookedAppointments int
    Employees         int
}
```

The Golang representation of the documents stored in Redis

3.8. Database Replication

To ensure maximum **Availability**, both **MongoDB** and **Redis** use **Replication** to have a **backup** of all data in case of a failure.

MongoDB is configured to have 3 servers, organized in a **ReplicaSet**. In this configuration, a server is elected as **Primary**, which handles read/write requests, while the other 2 instances work as **Secondaries** to store a replica of the data. The **ReplicaSet** must be configured and started via the **Mongosh CLI** once all the servers are live.

```
db:
  container_name: db
  restart: "no"
  image: mongo
  depends_on:
    mongodb1:
      condition: service_healthy
  command: >
    mongosh --host mongodb1:27017 --eval
    '
    db = (new Mongo("172.16.5.42:27017")).getDB("test");
    db = (new Mongo("172.16.5.43:27017")).getDB("test");
    db = (new Mongo("172.16.5.47:27017")).getDB("test");
    config = {
    "_id" : "barberReplSet",
    "members" : [
      {
        "_id" : 0,
        "host" : "172.16.5.42:27017",
        "priority" : 5
      },
      {
        "_id" : 1,
        "host" : "172.16.5.43:27017",
        "priority" : 3
      },
      {
        "_id" : 2,
        "host" : "172.16.5.47:27017",
        "priority" : 1
      }
    ]
    };
    rs.initiate(config);
    '
```

The Docker Compose code responsible for spinning up the container used to configure the Replica Set

Redis uses a similar replication scheme, where one server handles read/write operations, while the other 2 instances are configured as **Slaves**, storing a copy of the data.

```
cache-slave:
  restart: always
  container_name: cache-slave
  image : redis
  ports:
    - 0.0.0.0:6379:6379
  command: redis-server --slaveof 172.16.5.42 6379

mongodb-replica:
  container_name: mongodb-replica
  image: mongo
  ports:
    - 0.0.0.0:27017:27017
  restart: always
  command: mongod --replSet barberReplSet
```

The Docker Compose code responsible for instantiating replication container on the Secondary Servers

As an added bonus, due to the fact that this project uses **Docker** technology, it is possible to **quickly spin up new instances of the servers on-demand, on any type of platform**.

3.9. Sharding Proposal

In order to achieve **faster response times** and a **higher degree of scalability**, it could be considered to implement **Data Sharding**.
Data Sharding can work alongside **Data Replication**, in order to maintain fault tolerance.
When implementing Data Sharding, 2 factors must be considered: the **Sharding Key** and the **Partitioning Method**.
The Sharding Key is the field used to decide which server should handle each document, while the Partitioning Method is the algorithm that specifies how this field is interpreted.

Due to the inherently **geographical** nature of the application, it would make sense to shard data on a **Regional** basis, as it is way more probable that a user accesses data of the region they find themselves in. To achieve this, a "**Region**" field could be added to each document of every collection, and then used as the **Sharding Key**. The **Region** field could be decided based on a couple of simple rules:

- **Users**: the Region they are registering from
- **Barbershops**: the Region they are in based on their coordinates
- **Reviews**, **ShopViews**, **Appointments**: the same Region of the Barbershop they refer to

Once a **Region** has been decided, a simple **List Sharding Method** could distribute the data on the geographically closest server.

3.10. Platform, Technologies and Architecture

BarberShop is a webapp built with enterprise-level technologies. The whole application can be divided into 3 different parts:

- **Frontend**: built on top of the **Next.js** and **React** frameworks. It is written in the **javascript** and **typescript** languages. It uses the **Tailwind** library for its graphics.

- **Backend**: offers a **REST API** to the Frontend. It is written in **Golang**. It uses the **Gin** framework as its router to handle requests. Transmits its data in **JSON** format.

- **Databases**: the databases of choice, as stated above, are **MongoDB** and **Redis**. Each one of these is replicated 2 times, in order to achieve 3 live copies of the database data. Each database instance lives in a dedicated **Docker** container, in order to simplify database deployment and render it platform-agnostic.

The **Backend** was written with **Clean architecture** and **Dependency injection** principles in mind:

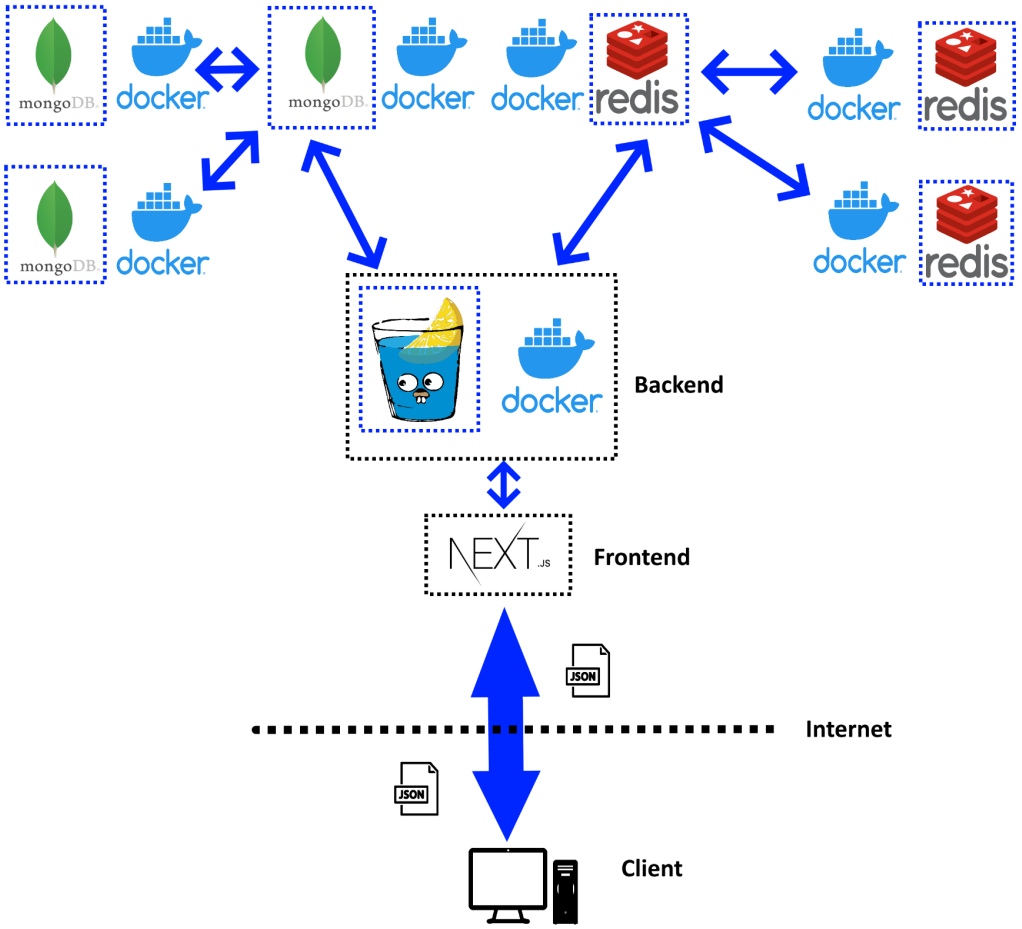
Clean Architecture emphasizes separating concerns, achieving independence from frameworks, and enabling easy testing.

By employing a layered approach, it provides clarity and flexibility, ensuring the core business logic remains decoupled from external dependencies.

On the other hand, **Dependency Injection** complements **Clean Architecture** by managing component dependencies through injection rather than direct instantiation. This decouples components from concrete implementations, enhancing modularity, testability, and extensibility.

The advantages of using these principles can be found in the source code which is extensively tested and loosely coupled.

Here is an overview of the application as a whole:

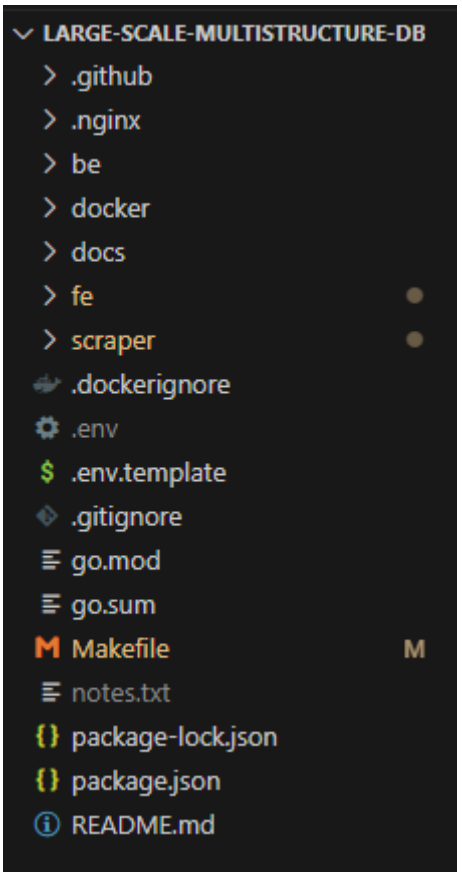


The architecture diagram of interaction between the platform's components

4. Implementation

All the code of this project is available in a **Git** repository at the following address: https://github.com/just-hms/large-scale-multistructure-db

The repository of the project is organized as follows:



The VSCode screenshot of the repo's **root** folder

- **be**: contains all the code of the Backend
- **docker**: contains the Docker files necessary to properly create the containers used by the application
- **docs**: has a collection of some of the pictures used in this documentation and some useful reference doc
- **fe**: contains all the code of the Frontend
- **scraper**: contains the various scrapers and the importer
- **Makefile**: a makefile was used in order to shorten long commands that are frequently used. It is NOT used to build the application or parts of it.

4.1. Frontend

The **Frontend** of the application is built using **Next.js**, a powerful **React** framework that seamlessly integrates with popular tools. **Next.js** is an excellent choice for developing high-performance front-end applications.

4.1.1 Code Organization

The code in the **Frontend** follows the standard **Next.js** project structure, where **pages** represent the **endpoints** accessible to the users. In this project, there are nine **endpoints** available, each of which requires **authenticated requests** to the **Backend** server for access control. The available endpoints are as follows:

- admin
- barber
- home
- password_recovery
- search
- shop
- signup
- user

To maintain a modular and reusable codebase, all **Next components** used in the pages are stored in the **components** folder. This allows for easier management and maintenance of these components throughout the project.

In addition, the **lib** folder contains utility functions that are utilized by various **Next** components to communicate with the backend. These functions primarily focus on fetching data from backend APIs to provide dynamic content and enhance the user experience.

By organizing the code in this manner, we ensure a **structured** and **maintainable** frontend implementation for our **Next.js** application.

Every component is styled using **Tailwind CSS** which is a highly customizable utility-first CSS framework that provides a wide range of pre-built components and utility classes.

The most interesting implementation here is probably the **Slot** handling in the **Calendar**:
Implemented in ***/fe/components/shop_component/calendar.tsx***, it fills the **Calendar**

with a monthful of **Slots**, checking that there's at least one **employee** free to take care of that **Slot**.
The implementation is as it follows:

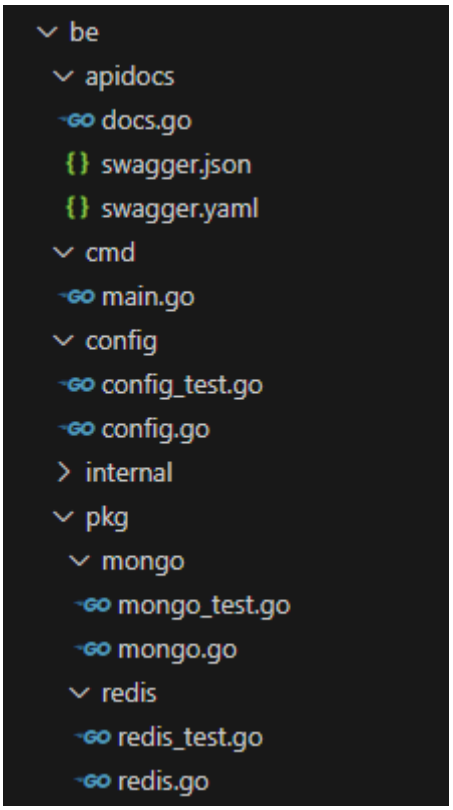
```
[...]
export const craftEventObject = (index:any,calendar:any) =>{
        const date = moment().add(30*index, 'm')
        if(date.hours() > 20|| date.hours() < 7 ){
        return null
        }
        if (date.minutes() >= 30){
        date.set('hour',date.hours()+1)
        date.set('minute',0)
        }else{
        date.set('minute',30)
        }
        date.set('second',0)
        date.set('millisecond',0)
        for(var i in calendar){
        if((new Date(calendar[i].Start)).getTime() === date.toDate().getTime()){
        if(calendar[i].BookedAppointments == calendar[i].Employees){
        return null
        }
        }
        }
        return {
        'title' : "Slot Disponibile",
        'start' : date.toDate().toISOString(),
        'end'   : moment(date).add(30, 'm').toDate().toISOString(),
        }
}

export const fillCalendar = (calendar:any, employees:any)=>{
        let events = []
        if(employees != 0){
        for(let index = 0; index < 48*30; index += 1){
        var event = craftEventObject(index,calendar)
        if(event){
        events.push(event)
        }
        }
        }
        return events
}
[...]
```

4.2. Backend

As it was previously mentioned, the **Backend** implements a **REST Api** that handles all the operations between users and the databases. The **Backend** is written in **Golang**, and uses the **Gin** library as its router.

4.2.1. File List

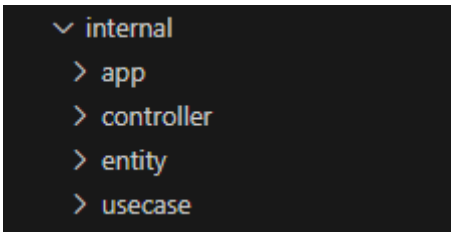
The Backend code of the application is organized in folders depending on the functionality it implements (more on it in Chapter 4.2.2) :



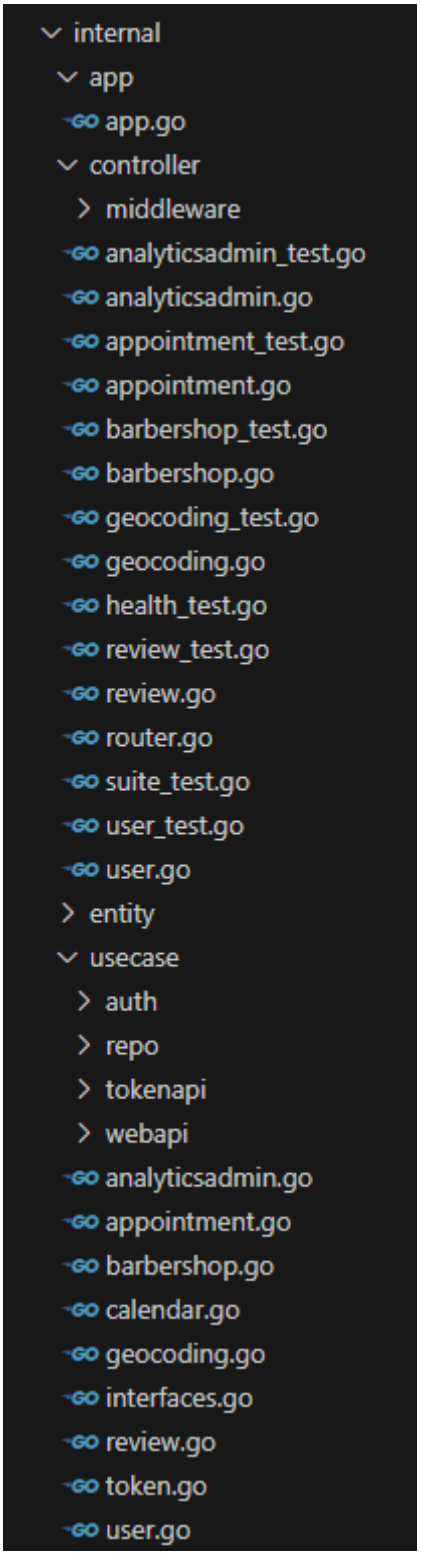
The structure of the files inside the **be** folder

- **apidocs**: contains all the code **documenting the REST Api** endpoints, their parameters, and their behavior. The contents of the folder are auto-generated through the use of **Swagger** annotations in the controller files. The **documentation is available through a Web Interface** once the server is running.
- **cmd**: contains the **entry point** of execution of the **Backend**.
- **config**: contains files that handle the loading of the **.env config file**.
- **internal**: contains the **core** of the **Backend** itself.
- **pkg**: contains files where **util functions** are defined in order to properly setup and interface with the **Databases**.

The **internal** folder is organized as follows:



The structure of the files inside the **internal** folder



A closer look to the files inside the **internal** folder

- **app**: contains the file responsible for initializing connection to the databases and starting the **Gin** router.
- **controller**: contains the controllers, files used in order to handle the http REST requests.
- **controller/middleware**: contains all the necessary **middlewares**. Among these there are the user auth and Barber/Admin auth middlewares.
- **entity**: contains the **entities** used throughout the **Backend**. An entity is a data structure representing a document or element.
- **usecase**: contains the **usecases**, files used by the controllers to query the **repo** functions and do some data processing if needed.
- **usecase/auth**: contains the file password.go, responsible for salting, hashing and verifying the user passwords stored in MongoDB.
- **usecase/repo**: contains the **repos**, files that implement the functions and operations necessary to interface with the databases.
- **usecase/tokenapi**: contains the util files that handle **jwt tokens**. User auth is handled in the db via the creation and verification of jwt token, sent in the http header of every auth-needing request.
- **usecase/webapi**: contains the files used to connect to **Geaoapify**'s service. **Geoapify** is a reverse geolocation service that returns the coordinates of a specified place/address.

4.2.2. Packages and Architecture

The Backend is organized in several Golang packages. A package in Golang is a collection of files “with a common goal/behavior”, not too dissimilar from Java packages, and mostly align with how the folders are structured.

The most relevant packages are:

- **controller package**: contains all the functions used by the **Router** to handle **HTTP requests**. Each **controller** function is prefaced with **Swagger Annotations**, that are used to automatically build the **documentation** available in the apidocs folder and the webserver.

```go
// Show Shows the user informations
//
// @Summary Show user information by ID
// @Description Get user information by ID
// @Tags User
// @Accept json
// @Produce json
// @Param id path string true "User ID"
// @Success 200 {object} map[string]entity.User
// @Failure 401 {object} string "Unauthorized"
// @Failure 404 {object} map[string]string
// @Router /user/self [get]
// @Router /admin/user/{id} [get]
func (ur *UserRoutes) Show(ctx *gin.Context) {

    ID := ctx.Param("id")

    user, err := ur.userUseCase.GetByID(ctx, ID)

    if err != nil {
        ctx.JSON(http.StatusNotFound, gin.H{"error": err.Error()})
        return
    }

    ctx.JSON(http.StatusOK, gin.H{"user": user})
}
```

An example **controller** with its **Swagger** annotations

- **middleware package**: contains several functions used by the **controller**, such as user and role **authentication**.

```go
func (mr *MiddlewareRoutes) RequireAdmin(ctx *gin.Context) {

    token := ExtractTokenFromRequest(ctx)

    tokenID, err := mr.tokenUseCase.ExtractTokenID(token)

    if err != nil {
        ctx.AbortWithStatusJSON(http.StatusUnauthorized, gin.H{"error": "Unauthorized"})
        return
    }

    user, err := mr.userUseCase.GetByID(ctx, tokenID)

    if err != nil {
        ctx.AbortWithStatusJSON(http.StatusUnauthorized, gin.H{"error": "Unauthorized"})
        return
    }

    if user.Type != entity.ADMIN {
        ctx.AbortWithStatusJSON(http.StatusUnauthorized, gin.H{"error": "Unauthorized"})
    }

    ctx.Next()
}
```

An example **middleware** that check if the user that issued the request is an Admin

- **usecase package**: contains functions used by the controllers to query the **repo** functions and do some data processing if needed.

```go
func (uc *BarberShopUseCase) GetOwnedShops(ctx context.Context, user *entity.User) ([]*entity.BarberShop, error) {
    return uc.shopRepo.GetOwnedShops(ctx, user)
}

func (uc *BarberShopUseCase) Store(ctx context.Context, shop *entity.BarberShop) error {
    return uc.shopRepo.Store(ctx, shop)
}
```

A couple of example **usecases** that don't do anything apart from using the **repos**

```go
func (uc *BarberShopUseCase) GetByID(ctx context.Context, viewerID string, ID string) (*entity.BarberShop, error) {

    // return the shop
    shop, err := uc.shopRepo.GetByID(ctx, ID)

    if err != nil {
        return nil, err
    }

    // save the view
    err = uc.viewRepo.Store(ctx, &entity.ShopView{
        UserID:       viewerID,
        BarbershopID: ID,
    })

    if err != nil {
        return nil, err
    }

    return shop, nil

}
```

A more complex **usecase** that does some extra processing

- **repo package**: contains the functions that query the databases.

```go
func (r *BarberShopRepo) GetOwnedShops(ctx context.Context, user *entity.User) ([]*entity.BarberShop, error) {

    if user.Type != entity.BARBER {
        return nil, fmt.Errorf("user is not a Barber")
    }

    filter := bson.M{"_id": bson.M{"$in": user.OwnedShops}}

    cur, err := r.DB.Collection("barbershops").Find(ctx, filter)
    if err != nil {
        return nil, err
    }

    defer cur.Close(ctx)

    shops := []*entity.BarberShop{}

    for cur.Next(ctx) {
        shop := entity.BarberShop{}
        if err := cur.Decode(&shop); err != nil {
            return nil, err
        }
        shops = append(shops, &shop)
    }
    return shops, nil
}
```

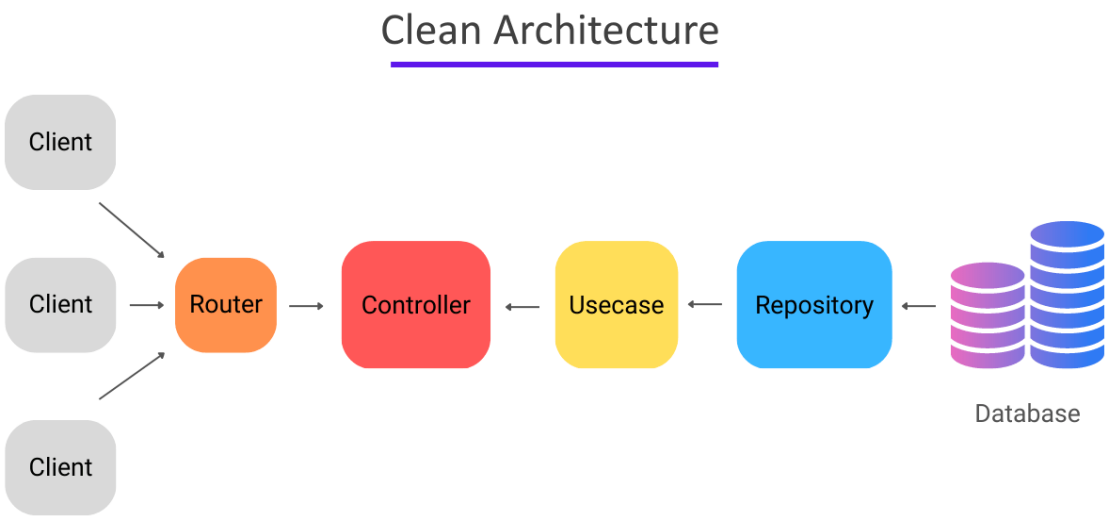
An example repo function that queries MongoDB

- **entity package**: contains the **entities** used throughout the **Backend**. An entity is a data structure representing a document or element.

```
type BarberShop struct {
    ID          string `bson:"_id"`
    Name        string
    Rating      float64
    Location *Location `json:"location" bson:"location"`

    Address     string
    Description string
    ImageLink   string
    Phone       string

    Employees int
}
```

The **entity** representing a Shop's data

The **Backend** was built with the **Clean Architecture** principle. This principle offers a clean **distinction of code** based on its functionality. The decoupling of code into smaller areas allows a programmer to change how some parts behave, without altering other areas' functionality.
The **controllers** handle data input/output only, and are composed of **Gin routes**.
The **usecases** handle data processing, formatting, and collection. They prepare data for the **controllers**, and may call one or more **repo** functions.
The **repos** interface directly with the chosen **databases** and handle all the usual CRUD/Aggregation needs.
The **entities** are the various types of data that flows through all these layers.



A diagram that summarizes the Clean Architecture control flow

4.2.3. External Packages/Libraries

The **Backend** employs a limited number of external packages/libraries. Here is a list of them:

- **GinSwagger**: package used to generate the **REST Api documentation** as stated above. The endpoint documentation is available at http://172.16.5.42/api/swagger/index.html
- **mongo-driver**: the Golang package that offers the **driver** used to interface with **MongoDB**.
- **go-redis**: the Golang package that offers the **driver** used to interface with **Redis**.
- **GeoApify**: a REST based service that offers **reverse Geolocation** functionality. The **Backend** uses this service to extract coordinates, needed to execute a **Georadius** search of the Shops, from a human-readable address or location.

4.2.4. Unit Tests

Unit Tests are used extensively throughout the **Backend** in order to ensure correct functionality, catch bugs early, and allow features to be developed “in a vacuum”, separated from the rest of the code.

Each **controller** and **repo** function has its own Unit Test that uses the powerful Go Testing Tools, which comes with the language and IDE.

```
run test | debug test
func (s *RepoSuite) TestGetAppointmentCancellationUserRanking() {

    s.SetupAnalyticsTestSuite()

    analyticsRepo := repo.NewAdminAnalyticsRepo(s.db)

    analytics, err := analyticsRepo.GetAppointmentCancellationUserRanking(context.Background())
    s.Require().NoError(err)
    s.Require().Equal(analytics[0]["username"], fixture[USER1_USERNAME])
    s.Require().Equal(analytics[0]["cancelCount"], 1)
}
```

An example **repo test**. Notice how the **IDE integration** offers a button at the top of the function to run individual unit tests.

Moreover, the project’s **Github repo** is configured to trigger a re-run of all unit tests before a **PR** can be merged. This ensures that the merged code is sane and doesn’t create issues.

4.3. Scraper and Importer

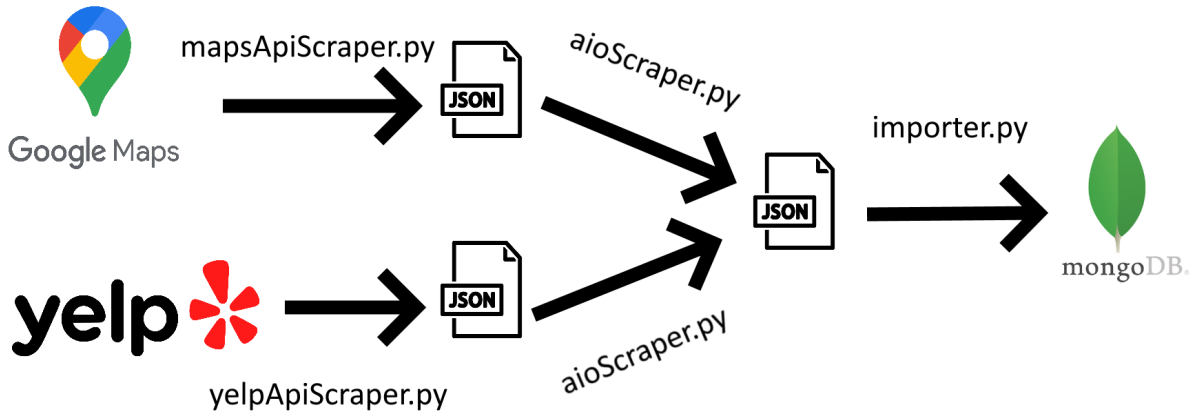
The data used in this project was obtained from 2 sources:

- **Google Maps/Places**
- **Yelp**

These sources are queried through their **Developer API**s by 2 individual **Python** scripts, and the resulting data is saved as **JSON**. Then, a 3rd **Python** script collects all the resulting **JSON** data, **merges** Shop and Review data whenever possible, and **homogenizes** the scraped results. This step is necessary due to the fact that both **Maps** and **Yelp** produce data in **JSON** format, albeit with different field names and structure. Also, a shop may be present in only one source or both.

Lastly, the formatted data is run through a Python **importer** script. This script handles **making adjustments** wherever necessary, such as converting coordinates to the format used by MongoDB's Georadius, **faking** the data that cannot be scraped, via the **Faker** library, preparing **indexes**, and **importing** the results into their appropriate **MongoDB Collection**.

The **faked data** is either **data that cannot be known**/scraped, such as a **user login details**, or data that is too **application specific**, such as **ShopViews** and **Appointments**.



A graph that summarizes the data collection and importing process

4.4. Docker Compose

The **Frontend**, **Backend** and **databases** are run in **containers**, powered by the **Docker** technology. A **container** acts as a self-contained system in which applications run in a sandboxed environment. The advantage of containers is the ability to **quickly spin them up on demand**, on any platform, without having to install anything apart from the Docker engine. It is also easy to manage them remotely through the use of **Docker Contexts**, which issue commands through the SSH protocol.

A **Docker container** is composed of an "**image**", which defines the programs available/preinstalled in the container. A high number of images are already pre-built, such as the ones that were used for mongo and redis, while the remaining ones can be manually built and customized. The process of declaring how images have to be built and used is done by a "**Docker Compose**" file, used by the **docker compose** utility. This file is written in **yaml** language.

```yaml
cache:
  restart: always
  container_name: cache
  image : redis
  ports:
    - 0.0.0.0:6379:6379

mongodb1:
  container_name: mongodb1
  image: mongo
  ports:
    - 0.0.0.0:27017:27017
  restart: always
  command: mongod --replSet barberReplSet
  healthcheck:
    test: echo 'db.runCommand("ping").ok' | mongosh mongodb1:27017/test --quiet
    interval: 5s
    timeout: 15s
    retries: 3
    start_period: 5s
```

An extract from the docker-compose.yml file located in /docker/deploy/

4.5. Database CRUD Operations

As any Database, **CRUD Operations** (Create Read Update Delete) must be performed in order to handle the data.

Below is a list of some of the most relevant Operations on the Collections found in MongoDB.

4.5.1. Users

Create

```go
func (r *UserRepo) Store(ctx context.Context, user *entity.User) error {

    if err := r.DB.Collection("users").FindOne(ctx, bson.M{"email": user.Email}).Err(); err == nil {
        return fmt.Errorf("user already exists")
    }

    user.ID = uuid.NewString()
    if user.SignupDate.IsZero() {
        user.SignupDate = time.Now()
    }

    _, err := r.DB.Collection("users").InsertOne(ctx, user)

    if err != nil {
        return fmt.Errorf("error inserting the user")
    }

    return nil
}
```

Create a User

Read

Due to the various needs of user data throughout the application, User information can be retrieved in several ways:

```go
func (r *UserRepo) GetByID(ctx context.Context, ID string) (*entity.User, error) {
    user := &entity.User{}

    err := r.DB.Collection("users").FindOne(ctx, bson.M{"_id": ID}).Decode(&user)

    if err != nil {
        if err == mongo.ErrNoDocuments {
            return nil, fmt.Errorf("user not found")
        }
        return nil, err
    }
    return user, nil
}
```

Get a User by its ID

```go
func (r *UserRepo) List(ctx context.Context, email string) ([]*entity.User, error) {
    filter := bson.M{}
    if email != "" {
        filter["email"] = primitive.Regex{Pattern: email, Options: "i"}
    }

    cur, err := r.DB.Collection("users").Find(ctx, filter)
    if err != nil {
        return nil, err
    }

    defer cur.Close(ctx)

    users := []*entity.User{}

    for cur.Next(ctx) {
        user := entity.User{}
        if err := cur.Decode(&user); err != nil {
            return nil, err
        }
        users = append(users, &user)
    }
    return users, nil
}
```

Get a list of Users, and optionally filter by their registered email address

```go
func (r *UserRepo) GetByEmail(ctx context.Context, email string) (*entity.User, error) {
    user := &entity.User{}
    err := r.DB.Collection("users").FindOne(ctx, bson.M{"email": email}).Decode(&user)
    if err != nil {
        if err == mongo.ErrNoDocuments {
            return nil, fmt.Errorf("user not found")
        }
        return nil, err
    }
    return user, nil
}
```

Get a User by their registered email address

Update

```go
func (r *UserRepo) ModifyByID(ctx context.Context, ID string, user *entity.User) error {

    update := bson.M{}

    if user != nil {
        if user.Email != "" {
            update["email"] = user.Email
        }

        if user.Password != "" {
            update["password"] = user.Password
        }

        if user.Type != "" {
            update["type"] = user.Type
        }
    }

    res, err := r.DB.Collection("users").UpdateOne(ctx, bson.M{"_id": ID}, bson.M{"$set": update})

    if res.MatchedCount == 0 {
        return errors.New("barberShop not found")
    }
    return err
}
```

Update any field of the User, based on need

Delete

```go
func (r *UserRepo) DeleteByID(ctx context.Context, ID string) error {

    res, err := r.DB.Collection("users").DeleteOne(ctx, bson.M{"_id": ID})
    if err != nil {
        return err
    }
    if res.DeletedCount == 0 {
        return fmt.Errorf("user not found")
    }
    return nil
}
```

Delete a User account

4.5.2. Shops

Create

```go
func (r *BarberShopRepo) Store(ctx context.Context, shop *entity.BarberShop) error {

    // cannot add a barber without a location if there is an index on it
    if shop.Location == nil {
        shop.Location = entity.FAKE_LOCATION
    }

    if err := r.DB.Collection("barbershops").FindOne(ctx, bson.M{"name": shop.Name}).Err(); err == nil {
        return fmt.Errorf("barber shop already exists")
    }

    shop.ID = uuid.NewString()
    _, err := r.DB.Collection("barbershops").InsertOne(ctx, shop)
    if err != nil {
        shop.ID = ""
        return fmt.Errorf("error inserting the barber shop: %s", err.Error())
    }
    return nil
}
```

Create a new Shop

Read

```go
func (r *BarberShopRepo) GetByID(ctx context.Context, ID string) (*entity.BarberShop, error) {

    barber := &entity.BarberShop{}

    err := r.DB.Collection("barbershops").FindOne(ctx, bson.M{"_id": ID}).Decode(&barber)

    if err != nil {
        if err == mongo.ErrNoDocuments {
            return nil, fmt.Errorf("barbershop not found")
        }
        return nil, err
    }
    return barber, nil
}
```

Get a Shop by its ID

```go
func (r *BarberShopRepo) GetOwnedShops(ctx context.Context, user *entity.User) ([]*entity.BarberShop, error) {

    if user.Type != entity.BARBER {
        return nil, fmt.Errorf("user is not a Barber")
    }

    filter := bson.M{"_id": bson.M{"$in": user.OwnedShops}}

    cur, err := r.DB.Collection("barbershops").Find(ctx, filter)
    if err != nil {
        return nil, err
    }

    defer cur.Close(ctx)

    shops := []*entity.BarberShop{}

    for cur.Next(ctx) {
        shop := entity.BarberShop{}
        if err := cur.Decode(&shop); err != nil {
            return nil, err
        }
        shops = append(shops, &shop)
    }
    return shops, nil
}
```

Get a list of the Shops owned by a given Barber User

```go
func (r *BarberShopRepo) Find(ctx context.Context, lat float64, lon float64, name string, radius float64) ([]*entity.BarberShop, error) {

    filter := bson.D{}

    if radius != 0 {
        filter = append(
            filter,
            bson.E{
                Key: "location",
                Value: bson.D{
                    {Key: "$near", Value: bson.D{
                        {Key: "$geometry", Value: entity.NewLocation(lon, lat)},
                        {Key: "$maxDistance", Value: radius},
                    }},
                },
            },
        )
    }

    if name != "" {
        filter = append(
            filter,
            bson.E{Key: "name", Value: primitive.Regex{Pattern: name, Options: "i"}},
        )
    }

    cur, err := r.DB.Collection("barbershops").Find(ctx, filter)
    if err != nil {
        return nil, err
    }

    defer cur.Close(ctx)

    shops := []*entity.BarberShop{}

    for cur.Next(ctx) {
        var shop entity.BarberShop

        if err := cur.Decode(&shop); err != nil {
            return nil, err
        }
        shops = append(shops, &shop)
    }
    return shops, nil
}
```

.Use a Georadius search to get all the Shops near a given location. Optionally filter by name.

Update

```go
func (r *BarberShopRepo) ModifyByID(ctx context.Context, ID string, shop *entity.BarberShop) error {

    update := bson.M{}

    if shop != nil {
        if shop.Location != nil {
            update["location"] = shop.Location
        }
        if shop.Name != "" {
            update["name"] = shop.Name
        }
        if shop.Description != "" {
            update["description"] = shop.Description
        }
        if shop.Employees != -1 {
            update["employees"] = shop.Employees
        }
    }

    res, err := r.DB.Collection("barbershops").UpdateOne(ctx, bson.M{"_id": ID}, bson.M{"$set": update})
    if res.MatchedCount == 0 {
        return errors.New("shop not found")
    }
    return err
}
```

Update any field of the Shop, based on need

Delete

```go
func (r *BarberShopRepo) DeleteByID(ctx context.Context, ID string) error {

    res, err := r.DB.Collection("barbershops").DeleteOne(ctx, bson.M{"_id": ID})
    if err != nil {
        return err
    }
    if res.DeletedCount == 0 {
        return fmt.Errorf("user not found")
    }
    return nil
}
```

Delete a Shop given its ID

4.5.3. Appointments

Create

At the time of an Appointments' creation, the corresponding User is also updated. This is a form of replication used to quickly display the last booked appointment in the User profile without having to query the Appointments collection.

```go
func (r *AppointmentRepo) Book(ctx context.Context, appointment *entity.Appointment) error {

    var barbershop entity.BarberShop
    err := r.DB.Collection("barbershops").FindOne(ctx, bson.M{"_id": appointment.BarbershopID}).Decode(&barbershop)

    if err != nil {
        return errors.New("the specified shop does not exists")
    }

    appointment.ID = uuid.NewString()
    appointment.BarbershopName = barbershop.Name
    if appointment.CreatedAt.IsZero() {
        appointment.CreatedAt = time.Now()
    }
    if appointment.Status == "" {
        appointment.Status = "pending"
    }

    // Store appointment fields before removing unused fields
    userID := appointment.UserID

    // Add the Appointment its collection
    _, err = r.DB.Collection("appointments").InsertOne(ctx, appointment)
    if err != nil {
        appointment.ID = ""
        return fmt.Errorf("error inserting the appointment: %s", err.Error())
    }

    // Remove unused field in the User's Appointment
    appointment.UserID = ""
    appointment.Username = ""

    // Add the appointment to the User
    userFilter := bson.M{"_id": userID}
    userUpdate := bson.M{"$set": bson.M{"currentAppointment": appointment}}
    _, err = r.DB.Collection("users").UpdateOne(ctx, userFilter, userUpdate)
    if err != nil {
        return err
    }

    // Add back the UserID for debugging purposes
    appointment.UserID = userID

    return err
}
```

Creation of a new Appointment and updating of the User

Read

```go
func (r *AppointmentRepo) GetByID(ctx context.Context, ID string) (*entity.Appointment, error) {

    appointment := &entity.Appointment{}

    err := r.DB.Collection("appointments").FindOne(ctx, bson.M{"_id": ID}).Decode(&appointment)

    if err != nil {
        if err == mongo.ErrNoDocuments {
            return nil, fmt.Errorf("appointment not found")
        }
        return nil, err
    }
    return appointment, nil
}
```

Get an Appointment by knowing its ID

Update/Delete

Instead of removing an **Appointment** from the database if it gets canceled, the "**status**" field is updated to "canceled". This is useful in order to make interesting **Aggregations** on the amount of **Appointments** canceled per **Shop** or **User**.

```go
func (r *AppointmentRepo) SetStatusFromUser(ctx context.Context, userID string, appointment *entity.Appointment) error {
    if appointment == nil {
        return errors.New("you must provide an appointment")
    }

    // Remove the appointment from the user
    userFilter := bson.M{"_id": userID}
    userUpdate := bson.M{"$unset": bson.M{"currentAppointment": ""}}
    _, err := r.DB.Collection("users").UpdateOne(ctx, userFilter, userUpdate)

    if err != nil {
        return err
    }

    filter := bson.M{"_id": appointment.ID}
    update := bson.M{"$set": bson.M{"status": appointment.Status}}

    _, err = r.DB.Collection("appointments").UpdateOne(ctx, filter, update)

    return err
}
```

Update the status of an Appointment by knowing the User it belong to as their current Appointment

4.5.4. ShopViews

Due to the fact that ShopViews are only used as an engagement metric by the Aggregations, they are create-only.

Create

```go
func (r *ShopViewRepo) Store(ctx context.Context, view *entity.ShopView) error {

    view.CreatedAt = time.Now()
    view.ID = uuid.NewString()
    _, err := r.DB.Collection("shopviews").InsertOne(ctx, view)
    if err != nil {
        view.ID = ""
        return fmt.Errorf("error inserting the shopview: %s", err.Error())
    }

    return err
}
```

Create a View on a Shop whenever a User browses a Shop's profile.

4.5.5. Reviews

Create

```go
func (r *ReviewRepo) Store(ctx context.Context, review *entity.Review, shopID string) error {

    shop := &entity.BarberShop{}
    err := r.DB.Collection("barbershops").FindOne(ctx, bson.M{"_id": shopID}).Decode(&shop)

    if err != nil {
        if err == mongo.ErrNoDocuments {
            return fmt.Errorf("the specified barber shop does not exist")
        }
        return err
    }

    user := &entity.User{}
    err = r.DB.Collection("users").FindOne(ctx, bson.M{"_id": review.UserID}).Decode(&user)

    if err != nil {
        if err == mongo.ErrNoDocuments {
            return fmt.Errorf("the specified user does not exist")
        }
        return err
    }

    review.ID = uuid.NewString()
    review.ShopID = shopID
    review.ShopName = shop.Name
    review.Username = user.Username
    review.Reported = false
    review.UpVotes = []string{}
    review.DownVotes = []string{}
    if review.CreatedAt.IsZero() {
        review.CreatedAt = time.Now()
    }

    _, err = r.DB.Collection("reviews").InsertOne(ctx, review)
    if err != nil {
        review.ID = ""
        return fmt.Errorf("error inserting the review: %s", err.Error())
    }

    return err
}
```

Create a new Review given a User and a Shop

Read

```go
func (r *ReviewRepo) GetByBarberShopID(ctx context.Context, shopID string) ([]*entity.Review, error) {

    err := r.DB.Collection("barbershops").FindOne(ctx, bson.M{"_id": shopID}).Err()

    if err != nil {
        if err == mongo.ErrNoDocuments {
            return nil, fmt.Errorf("the specified barber shop does not exist")
        }
        return nil, err
    }

    cur, err := r.DB.Collection("reviews").Find(ctx, bson.M{"shopId": shopID})
    if err != nil {
        return nil, err
    }

    defer cur.Close(ctx)

    reviews := []*entity.Review{}

    for cur.Next(ctx) {
        var review entity.Review

        if err := cur.Decode(&review); err != nil {
            return nil, err
        }
        reviews = append(reviews, &review)
    }
    return reviews, nil
}
```

Get a Review given its ID

Update

Reviews are updated whenever a new Up/DownVote gets inserted.

```go
func (v *VoteRepo) UpVoteByID(ctx context.Context, userID, shopID, reviewID string) error {

    err := v.DB.Collection("barbershops").FindOne(ctx, bson.M{"_id": shopID}).Err()

    if err != nil {
        if err == mongo.ErrNoDocuments {
            return fmt.Errorf("the specified barber shop does not exist")
        }
        return err
    }

    reviewFilter := bson.M{"_id": reviewID}
    err = v.DB.Collection("reviews").FindOne(ctx, reviewFilter).Err()

    if err != nil {
        if err == mongo.ErrNoDocuments {
            return fmt.Errorf("the specified review does not exist")
        }
        return err
    }

    // Check that Upvote wasn't already present
    upvoteFilter := bson.M{"_id": reviewID, "upvotes": bson.M{"$in": bson.A{userID}}}
    err = v.DB.Collection("reviews").FindOne(ctx, upvoteFilter).Err()
    if err != mongo.ErrNoDocuments {
        return fmt.Errorf("user already upvoted")
    }

    // Check if user has already downvoted the review, and remove if it has
    downvoteUpdate := bson.M{"$pull": bson.M{"downvotes": userID}}
    _, err = v.DB.Collection("reviews").UpdateOne(ctx, reviewFilter, downvoteUpdate)
    if err != nil {
        return err
    }

    upvoteUpdate := bson.M{"$push": bson.M{"upvotes": userID}}
    _, err = v.DB.Collection("reviews").UpdateOne(ctx, reviewFilter, upvoteUpdate)

    return err

}
```

Add the specified UserID to the list of Users who have Upvoted. Check if that User previously Downvoted and remove that Downvote if present.

```go
func (v *VoteRepo) DownVoteByID(ctx context.Context, userID, shopID, reviewID string) error {

    err := v.DB.Collection("barbershops").FindOne(ctx, bson.M{"_id": shopID}).Err()

    if err != nil {
        if err == mongo.ErrNoDocuments {
            return fmt.Errorf("the specified barber shop does not exist")
        }
        return err
    }

    reviewFilter := bson.M{"_id": reviewID}
    err = v.DB.Collection("reviews").FindOne(ctx, reviewFilter).Err()

    if err != nil {
        if err == mongo.ErrNoDocuments {
            return fmt.Errorf("the specified review does not exist")
        }
        return err
    }

    // Check that the Downvote wasn't already present
    downvoteFilter := bson.M{"_id": reviewID, "downvotes": bson.M{"$in": bson.A{userID}}}
    err = v.DB.Collection("reviews").FindOne(ctx, downvoteFilter).Err()
    if err != mongo.ErrNoDocuments {
        return fmt.Errorf("user already downvoted")
    }

    // Check if user has already upvoted the review, and remove if it has
    upvoteUpdate := bson.M{"$pull": bson.M{"upvotes": userID}}
    _, err = v.DB.Collection("reviews").UpdateOne(ctx, reviewFilter, upvoteUpdate)
    if err != nil {
        return err
    }

    downvoteUpdate := bson.M{"$push": bson.M{"downvotes": userID}}
    _, err = v.DB.Collection("reviews").UpdateOne(ctx, reviewFilter, downvoteUpdate)

    return err
}
```

Add the specified UserID to the list of Users who have Downvoted. Check if that User previously Upvoted and remove that Upvote if present.

```go
func (v *VoteRepo) RemoveVoteByID(ctx context.Context, userID, shopID, reviewID string) error {

    reviewFilter := bson.M{"_id": reviewID}
    err := v.DB.Collection("reviews").FindOne(ctx, reviewFilter).Err()

    if err != nil {
        if err == mongo.ErrNoDocuments {
            return fmt.Errorf("the specified review does not exist")
        }
        return err
    }

    upvoteUpdate := bson.M{"$pull": bson.M{"upvotes": userID}}
    _, err = v.DB.Collection("reviews").UpdateOne(ctx, reviewFilter, upvoteUpdate)
    if err != nil {
        return err
    }

    downvoteUpdate := bson.M{"$pull": bson.M{"downvotes": userID}}
    _, err = v.DB.Collection("reviews").UpdateOne(ctx, reviewFilter, downvoteUpdate)
    return err

}
```

Remove the specified UserID from the lists of users who have Upvoted or Downvoted the Review.

Delete

```go
func (r *ReviewRepo) DeleteByID(ctx context.Context, reviewID string) error {

    res, err := r.DB.Collection("reviews").DeleteOne(ctx, bson.M{"_id": reviewID})
    if err != nil {
        return err
    }
    if res.DeletedCount == 0 {
        return fmt.Errorf("review not found")
    }
    return nil
}
```

Delete a Review given its ID.

4.6. Key-Value Database Operations

As previously mentioned, **Redis** is the chosen **Key-Value Database** for this application. It is used as a **cache** for the **Appointments** of a **Shop**. Each **Shop** has a **Calendar** with **Slots** where **Users** can book an **Appointment**. Each **Shop** can have a variable number of **Employees**, thus a **Slot** becomes unavailable when the number of **Booked Appointments** is equal to the number of **Employees**.
Rebuilding the **Calendar** each time a **Shop** profile gets accessed would be incredibly inefficient. Thus, the summarized information on each **Slot** is stored into **Redis**, and read each time a **Shop** profile gets viewed by a **User**. A **Slot** gets created/updated each time a **User** books an **Appointment**, or in the case a **Shop** varies its number of **Employees**.

```go
func (r *SlotRepo) get(barberShopID string, date time.Time) (*entity.Slot, error) {

    key := key(barberShopID, date)
    data, err := r.Client.Get(key).Result()

    if err != nil {
        return nil, err
    }

    slot := &entity.Slot{}
    if err := json.Unmarshal([]byte(data), slot); err != nil {
        return nil, err
    }

    return slot, nil
}

func (r *SlotRepo) set(barberShopID string, date time.Time, slot *entity.Slot) error {

    key := key(barberShopID, date)

    if slot.Start.Before(time.Now()) {
        return errors.New("cannot create a slot prior to now")
    }

    expTime := time.Until(slot.Start.Add(time.Hour * 24))
    content, err := json.Marshal(slot)

    if err != nil {
        return err
    }

    err = r.Client.Set(key, content, expTime).Err()

    return err
}
```

The basic Get/Set functions used throughout the Slot repo

```go
func (r *SlotRepo) Book(ctx context.Context, appointment *entity.Appointment, slot *entity.Slot) error {

    if appointment.BarbershopID == "" {
        return errors.New("barberShopID not specified")
    }

    if slot.BookedAppointments >= slot.Employees {
        return errors.New("cannot book because this slot is full")
    }

    slot.BookedAppointments += 1

    return r.set(appointment.BarbershopID, appointment.StartDate, slot)
}
```

Create a new Slot if it didn't exist, or update it

```go
func (r *SlotRepo) GetByBarberShopID(ctx context.Context, ID string) ([]string, []*entity.Slot, error) {

    key := fmt.Sprintf("barbershop:%s:slots:*", ID)

    keys, err := r.Client.Keys(key).Result()

    sort.Slice(keys, func(i, j int) bool {
        return keys[i] < keys[j]
    })

    if err != nil {
        return nil, nil, err
    }

    slots := make([]*entity.Slot, 0, len(keys))

    for _, key := range keys {

        data, err := r.Client.Get(key).Result()
        if err != nil {
            return nil, nil, err
        }

        var slot entity.Slot
        if err := json.Unmarshal([]byte(data), &slot); err != nil {
            return nil, nil, err
        }

        slots = append(slots, &slot)
    }

    return keys, slots, nil
}
```

Get all the Slots associated with a given ShopID. USed to display the Calendar.

```go
func (r *SlotRepo) Cancel(ctx context.Context, appointment *entity.Appointment) error {

    slot, err := r.get(appointment.BarbershopID, appointment.StartDate)

    if err != nil {
        return errors.New("the slot does not exists")
    }

    if slot.BookedAppointments > 0 {
        slot.BookedAppointments -= 1
    }

    return r.set(appointment.BarbershopID, appointment.StartDate, slot)
}

func (r *SlotRepo) SetEmployees(ctx context.Context, shopID string, availableEmployees int) error {

    keys, slots, err := r.GetByBarberShopID(ctx, shopID)

    if len(keys) != len(slots) {
        return errors.New("key and slot length don't match")
    }

    for i := 0; i < len(keys); i++ {
        slots[i].Employees = availableEmployees
        expTime := time.Until(slots[i].Start.Add(time.Hour * 24))
        content, err := json.Marshal(slots[i])

        if err != nil {
            return err
        }

        err = r.Client.Set(keys[i], content, expTime).Err()

        if err != nil {
            return err
        }
    }

    return err
}
```

Update a Slot if the Appointment gets canceled or the number of Employees gets varied.

4.7. DocumentDB Aggregations

Aggregations are used in order to produce **Analytics** that can obtain useful data on the behavior of a **Shop** or the **platform** as a whole. **BarberShop** offers two types of **Aggregations**: **Shop Analytics** and **Admin Analytics**.
Shop Analytics, as the name entails, are computed per **Shop**, and they can be accessed by the **Barber** that owns that **Shop** through their profile.
Admin Analytics are instead only accessible by the **Admin** through their profile, and report statistics on the general health of the platform, or the status of the various **Shops**.

The available **Shop Analytics** are:

GetAppointmentCountByShop

A simple Aggregation that counts the number of **Appointments**, grouped by month

```go
func (r *BarberAnalyticsRepo) GetAppointmentCountByShop(ctx context.Context, shopID string) (map[string]int, error) {

    matchStage := bson.D{{"$match", bson.D{{"shopId", shopID}}}}

    groupStage := bson.D{{
        "$group",
        bson.D{
            {"_id", bson.D{
                {"$dateToString", bson.D{
                    {"date", "$startDate"},
                    {"format", "%Y-%m"},
                }},
            }},
            {"count", bson.D{
                {"$sum", 1},
            }},
        },
    }}

    cur, err := r.DB.Collection("appointments").Aggregate(ctx, mongo.Pipeline{matchStage, groupStage})
    if err != nil {
        return nil, err
    }
```

```
db.appointments.aggregate([
        { "$match": {"shopId":shopId}},
        {"$group": {
                "_id":{
                        "$dateToString":{"date":"$startDate","format":"%Y-%m"}
                },
                "count":{"$sum":1}
        }}
        ])
```

GetViewCountByShop

A simple Aggregation that counts the number of **ShopViews**, grouped by month.

```go
func (r *BarberAnalyticsRepo) GetViewCountByShop(ctx context.Context, shopID string) (map[string]int, error) {

    matchStage := bson.D{{"$match", bson.D{{"shopId", shopID}}}}

    groupStage := bson.D{{
        "$group",
        bson.D{
            {"_id", bson.D{
                {"$dateToString", bson.D{
                    {"date", "$createdAt"},
                    {"format", "%Y-%m"},
                }},
            }},
            {"count", bson.D{
                {"$sum", 1},
            }},
        },
    }}

    cur, err := r.DB.Collection("shopviews").Aggregate(ctx, mongo.Pipeline{matchStage, groupStage})
    if err != nil {
        return nil, err
    }
```

db.shopviews.aggregate([

{ “$match”: {“shopId”:shopId}},

{“$group”: {

“_id”:{

“$dateToString”:{“date”:”$startDate”,”format”:”%Y-%m”}

},

“count”:{“$sum”:1}

}}

])

GetReviewCountByShop

A simple Aggregation that counts the number of **Reviews**, grouped by month.

```go
func (r *BarberAnalyticsRepo) GetReviewCountByShop(ctx context.Context, shopID string) (map[string]int, error) {

    matchStage := bson.D{{"$match", bson.D{{"shopId", shopID}}}}

    groupStage := bson.D{{
        "$group",
        bson.D{
            {"_id", bson.D{
                {"$dateToString", bson.D{
                    {"date", "$createdAt"},
                    {"format", "%Y-%m"},
                }},
            }},
            {"count", bson.D{
                {"$sum", 1},
            }},
        },
    }}

    cur, err := r.DB.Collection("reviews").Aggregate(ctx, mongo.Pipeline{matchStage, groupStage})
    if err != nil {
        return nil, err
    }
```

db.reviews.aggregate([

{ “$match”: {“shopId”:shopId}},

{“$group”: {

“_id”:{

“$dateToString”:{“date”:”$startDate”,”format”:”%Y-%m”}

},

“count”:{“$sum”:1}

}}

])

GetAppointmentCancellationRatioByShop

An Aggregation that gets the **ratio of canceled Appointments**. Each **Appointment** has a field that represents its status.

```
func (r *BarberAnalyticsRepo) GetAppointmentCancellationRatioByShop(ctx context.Context, shopID string) (map[string]float64, error) {

    matchStage := bson.D{{"$match", bson.D{{"shopId", shopID}}}}

    setStage := bson.D{{
        "$set",
        bson.D{
            {"isCanceled", bson.D{
                {"$cond", bson.A{
                    bson.D{{"$eq", bson.A{"$status", "canceled"}}},
                    1,
                    0,
                }},
            }},
        },
    }}

    groupStage := bson.D{{
        "$group",
        bson.D{
            {"_id", bson.D{
                {"$dateToString", bson.D{
                    {"date", "$startDate"},
                    {"format", "%Y-%m"},
                }},
            }},
            {"cancelCount", bson.D{
                {"$sum", "$isCanceled"},
            }},
            {"appCount", bson.D{
                {"$sum", 1},
            }},
        },
    }}

    projectStage := bson.D{{
        "$project",
        bson.D{
            {"cancellationRatio", bson.D{
                {"$trunc", bson.A{
                    bson.D{{"$divide", bson.A{"$cancelCount", "$appCount"}}},
                    2,
                }},
            }},
        },
    }}
```

```
cur, err := r.DB.Collection("appointments").Aggregate(ctx, mongo.Pipeline{matchStage, setStage, groupStage, projectStage})
if err != nil {
    return nil, err
}
```

```
db.appointments.aggregate([
        { “$match”: {“shopId”:shopId}},
        {“$set”: {
                “isCanceled”:{
                        {“$cond”: [{
                                “$eq”:[“$status”,”canceled”],
                                1,
                                0
                        }]}
                }
        }},
        {“$group”: {
```

```
            "_id":{
                "$dateToString":{"date":"$startDate","format":"%Y-%m"}
            },
            "cancelCount":{"$sum":"$isCanceled},
            "appCount":{"$sum":1}
        }},
        {"$project":{
            "cancellationRatio":{
                "$trunc":{"$divide":["$cancelCount","$appCount"]}
            }
        }}
        ])
```

GetAppointmentViewRatioByShop

An Aggregation that gets the **ratio of booked Appointments** over the **ShopViews** made by the platform's users. This is a useful metric as it denotes how much a **Shop** is successful at attracting potential customers once they see the **Shop**'s profile.

```go
func (r *BarberAnalyticsRepo) GetAppointmentViewRatioByShop(ctx context.Context, shopID string) (map[string]float64, error) {

    viewCount, err := r.GetViewCountByShop(ctx, shopID)
    if err != nil {
        return nil, err
    }

    appointmentCount, err := r.GetAppointmentCountByShop(ctx, shopID)
    if err != nil {
        return nil, err
    }

    results := make(map[string]float64)
    for month, vCount := range viewCount {
        aCount, ok := appointmentCount[month]
        if ok {
            results[month] = float64(aCount) / float64(vCount)
        } else {
            results[month] = 0.0
        }
    }

    return results, err

}
```

This Aggregation reuses the AppointmentCount and ViewCount Aggregations.

GetUpDownVoteCountByShop

A simple Aggregation that counts the number of **Upvotes** and **Downvotes** on all the **Reviews** of a **Shop**, grouped by month.

```go
func (r *BarberAnalyticsRepo) GetUpDownVoteCountByShop(ctx context.Context, shopID string) (map[string]map[string]int, error) {

    matchStage := bson.D{{"$match", bson.D{{"shopId", shopID}}}}

    projectStage := bson.D{{
        "$project",
        bson.D{
            {"createdAt", 1},
            {"upCount", bson.D{
                {"$size", "$upvotes"},
            }},
            {"downCount", bson.D{
                {"$size", "$downvotes"},
            }},
        },
    }}

    groupStage := bson.D{{
        "$group",
        bson.D{
            {"_id", bson.D{
                {"$dateToString", bson.D{
                    {"date", "$createdAt"},
                    {"format", "%Y-%m"},
                }},
            }},
            {"upCount", bson.D{
                {"$sum", "$upCount"},
            }},
            {"downCount", bson.D{
                {"$sum", "$downCount"},
            }},
        },
    }}

    cur, err := r.DB.Collection("reviews").Aggregate(ctx, mongo.Pipeline{matchStage, projectStage, groupStage})
    if err != nil {
        return nil, err
    }
```

db.reviews.aggregate([

{ “$match”: {“shopId”:shopId}},

{“$project”: {

“createdAt”: 1,

“upCount”:{“$size”:”$upvotes”},

“downCount”:{“$size”:”$downvotes”}

}}

{“$group”: {

“_id”:{

“$dateToString”:{“date”:”$startDate”,”format”:”%Y-%m”}

},

“upCount”:{“$sum”:”$upCount”},

“downCount”:{“$sum”:”$downCount”}

}}

])

GetReviewWeightedRatingByShop

An Aggregation that computes the **Weighted Rating** of a **Shop**. The **WeightedRating** is computed from the **rating** of its **Reviews**, weighted by a **WeightedScore**. The **WeightedScore** is calculated by multiplying together a **FreshnessScore** and a **VoteScore**. The **FreshnessScore** is assigned based on the “freshness” of a **Review**, making the the most recent **Reviews** the most impactful. The **FreshnessScore** uses the following formula:

created < 30 days -> 5 points
30 days <= created < 365 days -> 2 points
created > 365 days -> 1 point

The **VoteScore** is instead calculated by summing the amount of **Upvotes**, minus the amount of **Downvotes** a **Review** has.
The final **WeightedRating** is calculated by a standard weighted average, with the formula: *sum(weightedScore * rating) / sum(weightedScore)*

Note: Due to the sheer volume of this and the following Aggregation, their stages will be written “piece-by-piece” as they appear in the code. The nested pipeline of the “$lookup” operators would otherwise render the code illegible.

```go
// This aggregation produces a weighted rating of a Shop based on its Reviews.
// Reviews are weighted depending on Freshness and VoteScore
// Freshness:
//
//      created < 30 days -> 5 points
//   30 days <= created < 365 days -> 2 points
//   created > 365 days -> 1 point
//
// VoteScore: #upvotes - #downvotes
// WeightedScore: freshness * voteScore
// WeightedRating: (weightedScore * rating) / sum(weightedScore)
func (r *BarberAnalyticsRepo) GetReviewWeightedRatingByShop(ctx context.Context, shopID string) (float64, error) {

    matchStage := bson.D{{"$match", bson.D{{"shopId", shopID}}}}

    setStage1 := bson.D{{
        "$set",
        bson.D{
            {"upCount", bson.D{
                {"$size", "$upvotes"},
            }},
            {"downCount", bson.D{
                {"$size", "$downvotes"},
            }},
            {"daysElapsed", bson.D{
                {"$dateDiff", bson.D{
                    {"startDate", "$createdAt"},
                    {"endDate", "$$NOW"},
                    {"unit", "day"},
                }},
            }},
        },
    }}
```

{ “$match”: {“shopId”:shopId}},
{“$set”: {
“upCount”:{“$size”:”$upvotes”},
“downCount”:{“$size”:”$downvotes”},
“daysElapsed”:{
“$dateDiff”:{
“startDate”: “$createdAt”,
“endDate”: “$$NOW”,
“unit”, “day”
}
}
}},

```
setStage2 := bson.D{{
    "$set",
    bson.D{
        {"freshnessScore", bson.D{
            {"$switch", bson.D{
                {"branches", bson.A{
                    bson.D{
                        {"case", bson.D{
                            {"$and", bson.A{
                                bson.D{{"$gte", bson.A{"$daysElapsed", 0}}},
                                bson.D{{"$lt", bson.A{"$daysElapsed", 30}}},
                            }},
                        }},
                        {"then", 5},
                    },
                    bson.D{
                        {"case", bson.D{
                            {"$and", bson.A{
                                bson.D{{"$gte", bson.A{"$daysElapsed", 30}}},
                                bson.D{{"$lt", bson.A{"$daysElapsed", 365}}},
                            }},
                        }},
                        {"then", 2},
                    },
                }},
                {"default", 1},
            }},
        }},
        {"voteScore", bson.D{
            {"$cond", bson.A{
                bson.D{{"$eq", bson.A{bson.D{{"$subtract", bson.A{"$upCount", "$downCount"}}}, 0}}},
                1,
                bson.D{{"$subtract", bson.A{"$upCount", "$downCount"}}},
            }},
        }},
    },
}}
```

{“$set”: {
“freshnessScore”:{
“$switch”:{
“branches”:[
{“case”:{
“$and”:[
{“$gte”:[“$daysElapsed”,0]},
{“$lt”:[“$daysElapsed”,30]}
]
},
“then”:5},
{“case”:{
“$and”:[
{“$gte”:[“$daysElapsed”,30]},
{“$lt”:[“$daysElapsed”,365]}
]
},
“then”:2},

```
                    ],
                    "default": 1
                }
            },
            "voteScore":{
                "$cond":[
                    {"$eq":[{"$subtract":["$upCount","$downCount"]},0]},
                    1,
                    "$subtract":["$upCount","$downCount"]
                ]
            }
        }},
```

```go
setStage3 := bson.D{{
    "$set",
    bson.D{
        {"weightedScore", bson.D{
            {"$multiply", bson.A{"$freshnessScore", "$voteScore"}},
        }},
    },
}}

groupStage := bson.D{{
    "$group",
    bson.D{
        {"_id", "$shopId"},
        {"numerator", bson.D{
            {"$sum", bson.D{
                {"$multiply", bson.A{"$weightedScore", "$rating"}},
            }},
        }},
        {"denominator", bson.D{
            {"$sum", "$weightedScore"},
        }},
    },
}}

projectStage := bson.D{{
    "$project",
    bson.D{
        {"_id", 0},
        {"weightedRating", bson.D{
            {"$trunc", bson.A{
                bson.D{{"$divide", bson.A{"$numerator", "$denominator"}}},
                2,
            }},
        }},
    },
}}

cur, err := r.DB.Collection("reviews").Aggregate(ctx, mongo.Pipeline{matchStage, setStage1, setStage2, setStage3, groupStage, projectStage})
if err != nil {
    return 0, err
}
```

```
        {"$set": {
            "weightedScore":{
                "$multiply":["$freshnessScore","$voteScore"]
            }
        }},
        {"$group":{
            "_id": "$shopId",
            "numerator": {
```

```
                "$sum": {"$multiply":["$weightedScore","$rating"]}
            },
            "denominator": {
                "$sum":"$weightedScore"
            }
    }},
    {"$project":{
            "_id":0,
            "weightedRating":{
                "$trunc":{"$divide":["$numerator","$denominator"]}
            }
    }}
```

GetInactiveUsersByShop

An Aggregation born to target **customer retention issues**. It gets the usernames of **Users** who previously booked an **Appointment** in the **Shop**, continue using the platform, but no longer book **Appointments** in that particular **Shop**. This data might be used to send users some **targeted fidelity bonuses**.

This Aggregation is quite complex and can be basically broken into 4 steps:

- Use a **Replace root** to get just a single doc with the matching shopId
- Find all the **Users** that made an **Appointment** in the **Shop** in the **last 90 days**
- Find all the **Users** that made an **Appointment** in the **last 90 days in another Shop** and **weren't in the new users** (**active users in other Shops**)
- Find all the **Users** that made an **Appointment** in the past and are **active in other Shops**
- Tl;Dr: **OlderClientsShop** in (**NewerClients** not in **NewerClientsShop**)

```go
// This aggregation is quite complex and can be basically broken into 4 steps:
// - Use a Replace root to get just a single doc with the shopId
// - Find all the users that made an appointment in the Shop in the last 90 days
// - Find all the users that made an appointment in the last 90 days in another Shop and weren't in the new users (active users in other Shops)
// - Find all the users that made an appointment in the past and are active in other Shops
// Tl;Dr: OlderClients in (NewerClients not in NewerClientsShop)
func (r *BarberAnalyticsRepo) GetInactiveUsersByShop(ctx context.Context, shopID string) ([]string, error) {

    matchStage1 := bson.D{{
        "$match", bson.D{
            {"shopId", shopID},
        },
    }}

    groupStage1 := bson.D{{
        "$group",
        bson.D{
            {"_id", "$shopId"},
            {"doc", bson.D{
                {"$first", "$$ROOT"},
            }},
        },
    }}

    replaceRootStage1 := bson.D{{
        "$replaceRoot",
        bson.D{
            {"newRoot", "$doc"},
        },
    }}
```

{ “$match”: {“shopId”:shopId}},
{“$group”: {
“_id”:”$shopId”,
“doc”:{“$first”:”$$ROOT”}
}},
{“$replaceRoot”:{“newRoot”:”$doc”}},

```
lookupMatchShopClientsStage := bson.D{{
    "$match", bson.D{
        {"shopId", shopID},
        {"status", bson.D{
            {"$ne", "canceled"},
        }},
    },
}}

lookupSetElapsedDaysStage := bson.D{{
    "$set",
    bson.D{
        {"daysElapsed", bson.D{
            {"$dateDiff", bson.D{
                {"startDate", "$startDate"},
                {"endDate", "$$NOW"},
                {"unit", "day"},
            }},
        }},
    },
}}

lookupMatchNewerAppointmentsStage := bson.D{{
    "$match", bson.D{
        {"$expr", bson.D{
            {"$lt", bson.A{"$daysElapsed", 90}},
        }},
    },
}}

lookupProjectUserIdStage := bson.D{{
    "$project", bson.D{
        {"_id", 0},
        {"userId", 1},
    },
}}

lookupNewerClientsShopPipeline := bson.A{lookupMatchShopClientsStage, lookupSetElapsedDaysStage, lookupMatchNewerAppointmentsStage, lookupProjectUserIdStage}

lookupNewerClientsShopStage := bson.D{{
    "$lookup", bson.D{
        {"from", "appointments"},
        {"pipeline", lookupNewerClientsShopPipeline},
        {"as", "newClientsShop"},
    },
}}
```

lookupNewerClientsShopPipeline: *[*

{ “$match”: {

“shopId”:shopId,

“status”:{“$ne”:”canceled”}

}},

{“$set”: {

“daysElapsed”:{

“$dateDiff”:{

“startDate”: “$createdAt”,

“endDate”: “$$NOW”,

“unit”, “day”

}

}

}},

{ “$match”: {

“$expr”:{“$lt”:[“$daysElapsed”,90]}

}},

{“$project”:{

“_id”:0,

“userId”:1

}}

]

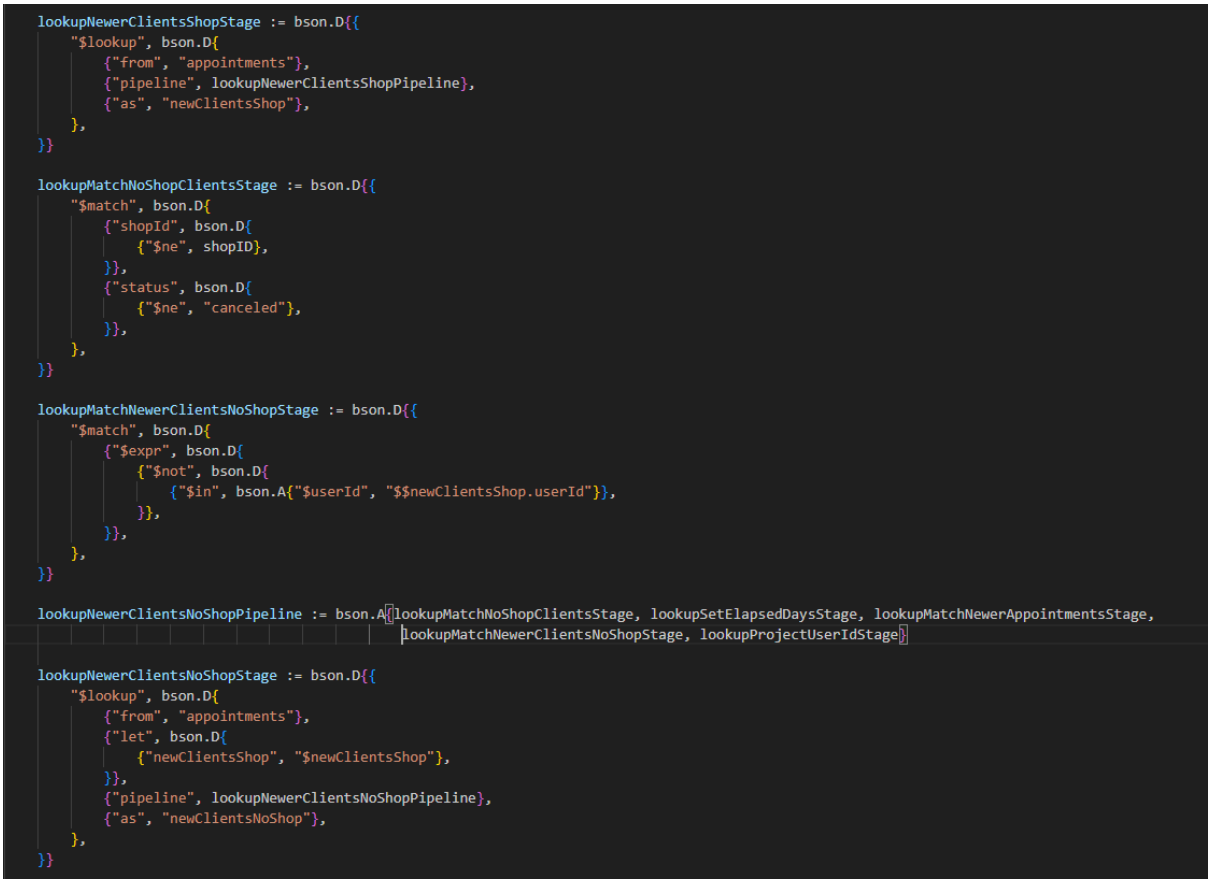
{“$lookup”:{

“from”:”appointments”,

*“pipeline”:*lookupNewerClientsShopPipeline,

“as”:”newerClientsShop”

}},



```
lookupNewerClientsShopStage := bson.D{{
    "$lookup", bson.D{
        {"from", "appointments"},
        {"pipeline", lookupNewerClientsShopPipeline},
        {"as", "newClientsShop"},
    },
}}

lookupMatchNoShopClientsStage := bson.D{{
    "$match", bson.D{
        {"shopId", bson.D{
            {"$ne", shopID},
        }},
        {"status", bson.D{
            {"$ne", "canceled"},
        }},
    },
}}

lookupMatchNewerClientsNoShopStage := bson.D{{
    "$match", bson.D{
        {"$expr", bson.D{
            {"$not", bson.D{
                {"$in", bson.A{"$userId", "$$newClientsShop.userId"}},
            }},
        }},
    },
}}

lookupNewerClientsNoShopPipeline := bson.A{lookupMatchNoShopClientsStage, lookupSetElapsedDaysStage, lookupMatchNewerAppointmentsStage,
                                           lookupMatchNewerClientsNoShopStage, lookupProjectUserIdStage}

lookupNewerClientsNoShopStage := bson.D{{
    "$lookup", bson.D{
        {"from", "appointments"},
        {"let", bson.D{
            {"newClientsShop", "$newClientsShop"},
        }},
        {"pipeline", lookupNewerClientsNoShopPipeline},
        {"as", "newClientsNoShop"},
    },
}}
```

lookupNewerClientsNoShopPipeline: *[*

{ “$match”: {

“shopId”:{“$ne”:shopId},

“status”:{“$ne”:”canceled”}

}},

{“$set”: {

“daysElapsed”:{

“$dateDiff”:{

“startDate”: “$createdAt”,

“endDate”: “$$NOW”,

```
                    "unit", "day"
                }
            }
        }},
        { "$match": {
            "$expr":{"$lt":["$daysElapsed",90]}
        }},
        {"$match":{
            "$expr":{
                "$not":{"$in":["$userId","$$newClientsShop.userId"]}
            }
        }},
        {"$project":{
            "_id":0,
            "userId":1
        }}
]

        {"$lookup":{
            "from":"appointments",
            "let":{"newClientsShop","$newClientsShop"},
            "pipeline":lookupNewerClientsNoShopPipeline,
            "as":"newerClientsNoShop"
        }},
```

```
lookupMatchOlderAppointmentsStage := bson.D{{
    "$match", bson.D{
        {"$expr", bson.D{
            {"$gte", bson.A{"$daysElapsed", 90}},
        }},
    },
}}

lookupMatchOlderClientsNotReturningStage := bson.D{{
    "$match", bson.D{
        {"$expr", bson.D{
            {"$in", bson.A{"$userId", "$$newClientsNoShop.userId"}},
        }},
    },
}}

lookupGroupByUsernameStage := bson.D{{
    "$group",
    bson.D{
        {"_id", "$username"},
    },
}}

lookupProjectUsernameStage := bson.D{{
    "$project", bson.D{
        {"_id", 0},
        {"username", "$_id"},
    },
}}

lookupOlderClientsNotReturningPipeline := bson.A{lookupMatchShopClientsStage, lookupSetElapsedDaysStage, lookupMatchOlderAppointmentsStage,
                                                 lookupMatchOlderClientsNotReturningStage, lookupGroupByUsernameStage, lookupProjectUsernameStage}

lookupOlderClientsNotReturningStage := bson.D{{
    "$lookup", bson.D{
        {"from", "appointments"},
        {"let", bson.D{
            {"newClientsNoShop", "$newClientsNoShop"},
        }},
        {"pipeline", lookupOlderClientsNotReturningPipeline},
        {"as", "oldClientsShopUsername"},
    },
}}
```

lookupOlderClientsNotReturningPipeline:*[*

{ “$match”: {

“shopId”:shopId,

“status”:{“$ne”:”canceled”}

}},

{“$set”: {

“daysElapsed”:{

“$dateDiff”:{

“startDate”: “$createdAt”,

“endDate”: “$$NOW”,

“unit”, “day”

}

}

}},

{ “$match”: {

“$expr”:{“$gte”:[“$daysElapsed”,90]}

}},

{“$match”:{

“$expr”:{

“$in”:[“$userId”,”$$newClientsNoShop.userId”]

}

```
        }},
        {"$group":{
                "_id":"$username"
        }},
        {"$project":{
                "_id":0,
                "username":"$_id"
        }}
]

        {"$lookup":{
                "from":"appointments",
                "let":{"newClientsNoShop","$newClientsNoShop"},
                "pipeline":lookupOlderClientsNotReturningPipeline,
                "as":"oldClientsShopUsername"
        }},
```

```go
projectStage1 := bson.D{{
    "$project",
    bson.D{
        {"_id", 0},
        {"oldClientsShopUsername", 1},
    },
}}

cur, err := r.DB.Collection("appointments").Aggregate(ctx, mongo.Pipeline{matchStage1, groupStage1, replaceRootStage1,
    lookupNewerClientsShopStage, lookupNewerClientsNoShopStage, lookupOlderClientsNotReturningStage, projectStage1})
if err != nil {
    return nil, err
}
```

```
        {"$project":{
                "_id":0,
                "oldClientsShopUsername":1
        }}
```

The available **Admin Analytics** are:

GetAppointmentCount

A simple Aggregation that counts the number of **Appointments**, grouped by month

```
func (r *AdminAnalyticsRepo) GetAppointmentCount(ctx context.Context) (map[string]int, error) {

    groupStage := bson.D{{
        "$group",
        bson.D{
            {"_id", bson.D{
                {"$dateToString", bson.D{
                    {"date", "$startDate"},
                    {"format", "%Y-%m"},
                }},
            }},
            {"count", bson.D{
                {"$sum", 1},
            }},
        },
    }}

    cur, err := r.DB.Collection("appointments").Aggregate(ctx, mongo.Pipeline{groupStage})
    if err != nil {
        return nil, err
    }
```

db.appointments.aggregate([
{“$group”: {
“_id”:{
“$dateToString”:{“date”:”$startDate”,”format”:”%Y-%m”}
},
“count”:{“$sum”:1}
}}
])

GetViewCount

A simple Aggregation that counts the number of **ShopViews**, grouped by month

```go
func (r *AdminAnalyticsRepo) GetViewCount(ctx context.Context) (map[string]int, error) {

    groupStage := bson.D{{
        "$group",
        bson.D{
            {"_id", bson.D{
                {"$dateToString", bson.D{
                    {"date", "$createdAt"},
                    {"format", "%Y-%m"},
                }},
            }},
            {"count", bson.D{
                {"$sum", 1},
            }},
        },
    }}

    cur, err := r.DB.Collection("shopviews").Aggregate(ctx, mongo.Pipeline{groupStage})
    if err != nil {
        return nil, err
    }
```

db.shopviews.aggregate([

{“$group”: {

“_id”:{

“$dateToString”:{“date”:”$startDate”,”format”:”%Y-%m”}

},

“count”:{“$sum”:1}

}}

])

GetReviewCount

A simple Aggregation that counts the number of **Reviews**, grouped by month

```go
func (r *AdminAnalyticsRepo) GetReviewCount(ctx context.Context) (map[string]int, error) {

    groupStage := bson.D{{
        "$group",
        bson.D{
            {"_id", bson.D{
                {"$dateToString", bson.D{
                    {"date", "$createdAt"},
                    {"format", "%Y-%m"},
                }},
            }},
            {"count", bson.D{
                {"$sum", 1},
            }},
        },
    }}

    cur, err := r.DB.Collection("reviews").Aggregate(ctx, mongo.Pipeline{groupStage})
    if err != nil {
        return nil, err
    }
```

db.reviews.aggregate([

{“$group”: {

“_id”:{

“$dateToString”:{“date”:”$startDate”,”format”:”%Y-%m”}

},

“count”:{“$sum”:1}

}}

])

GetNewUsersCount

A simple Aggregation that counts the number of **User registrations**, grouped by month

```
func (r *AdminAnalyticsRepo) GetNewUsersCount(ctx context.Context) (map[string]int, error) {

    groupStage := bson.D{{
        "$group",
        bson.D{
            {"_id", bson.D{
                {"$dateToString", bson.D{
                    {"date", "$signupDate"},
                    {"format", "%Y-%m"},
                }},
            }},
            {"count", bson.D{
                {"$sum", 1},
            }},
        },
    }}

    cur, err := r.DB.Collection("users").Aggregate(ctx, mongo.Pipeline{groupStage})
    if err != nil {
        return nil, err
    }
```

db.users.aggregate([

{“$group”: {

“_id”:{

“$dateToString”:{“date”:”$signupDate”,”format”:”%Y-%m”}

},

“count”:{“$sum”:1}

}}

])

GetAppointmentCancellationUserRanking

Get the amount of **canceled Appointments** each **User** has, then sort them descendent. This might help identify problematic/troll **Users**.

```go
// Appointments are grouped by username. Each Appointment is counted if it has the field status == "canceled".
// The result is then sorted in descending order.
func (r *AdminAnalyticsRepo) GetAppointmentCancellationUserRanking(ctx context.Context) ([]bson.M, error) {

    setStage := bson.D{{
        "$set",
        bson.D{
            {"isCanceled", bson.D{
                {"$cond", bson.A{
                    bson.D{{"$eq", bson.A{"$status", "canceled"}}},
                    1,
                    0,
                }},
            }},
        },
    }}

    groupStage := bson.D{{
        "$group",
        bson.D{
            {"_id", "$username"},
            {"cancelCount", bson.D{
                {"$sum", "$isCanceled"},
            }},
        },
    }}

    projectStage := bson.D{{
        "$project",
        bson.D{
            {"_id", 0},
            {"username", "$_id"},
            {"cancelCount", 1},
        },
    }}

    sortStage := bson.D{{
        "$sort",
        bson.D{
            {"cancelCount", -1},
        },
    }}

    cur, err := r.DB.Collection("appointments").Aggregate(ctx, mongo.Pipeline{setStage, groupStage, projectStage, sortStage})
    if err != nil {
        return nil, err
    }
```

```
db.appointments.aggregate([
        {“$set”: {
                “isCanceled”:{
                        {“$cond”: [{
                                “$eq”:[“$status”,”canceled”],
                                1,
                                0
                        }]}
                }
        }},
        {“$group”: {
                “_id”:”$username”,
```

```
        “cancelCount”:{“$sum”:”isCanceled”}
}},
{“$project”:{
        “_id”:0,
        “username”:”$_id”,
        “cancelCount”:1
}},
{“$sort”:{
        “cancelCount”:-1
}}
])
```

GetAppointmentCancellationShopRanking

Get the amount of **canceled Appointments** each **Shop** has, then sort them descendent. This might help identify problematic **Shops**.

```go
// Appointments are grouped by shop name. Each Appointment is counted if it has the field status == "canceled".
// The result is then sorted in descending order.
func (r *AdminAnalyticsRepo) GetAppointmentCancellationShopRanking(ctx context.Context) ([]bson.M, error) {

    setStage := bson.D{{
        "$set",
        bson.D{
            {"isCanceled", bson.D{
                {"$cond", bson.A{
                    bson.D{{"$eq", bson.A{"$status", "canceled"}}},
                    1,
                    0,
                }},
            }},
        },
    }}

    groupStage := bson.D{{
        "$group",
        bson.D{
            {"_id", "$shopId"},
            {"shopName", bson.D{
                {"$first", "$shopName"},
            }},
            {"cancelCount", bson.D{
                {"$sum", "$isCanceled"},
            }},
        },
    }}

    projectStage := bson.D{{
        "$project",
        bson.D{
            {"_id", 0},
            {"shopName", "$shopName"},
            {"cancelCount", 1},
        },
    }}

    sortStage := bson.D{{
        "$sort",
        bson.D{
            {"cancelCount", -1},
        },
    }}

    cur, err := r.DB.Collection("appointments").Aggregate(ctx, mongo.Pipeline{setStage, groupStage, projectStage, sortStage})
    if err != nil {
        return nil, err
```

db.appointments.aggregate([

{“$set”: {

“isCanceled”:{

{“$cond”: [{

“$eq”:[“$status”,”canceled”],

1,

0

}]}

}

}},

{“$group”: {

“_id”:”$shopId”,

```
        “shopName”:{“$first”,”$shopName”},
        “cancelCount”:{“$sum”:”$isCanceled”}
}},
{“$project”:{
        “_id”:0,
        “shopName”:”$shopName”,
        “cancelCount”:1
}},
{“$sort”:{
        “cancelCount”:-1
}}
])
```

GetEngagementShopRanking

Rank each **Shop** and sort them descendent based on the **Engagement** they get on the platform. The **EngagementScore** is computed as: 5*(#appointments) + (#upVotes + #downVotes)

```go
// The Engagement of a shop is computed as follows: 5*(#appointments) + (#upVotes + #downVotes)
// The ranking is achieved by:
// 1) Computing the size of upVotes and downVotes of each Review
// 2) Summing those results together on each Review
// 3) Grouping together the Reviews by shopId and summing all the results
// 4) For each shop, looking up its number of Appointments, multiplied by 5
// 5) Summing together the results of steps 4 and 5 for each shop
// 6) Sorting the shops descendently by the Engagement Rating of step 5
func (r *AdminAnalyticsRepo) GetEngagementShopRanking(ctx context.Context) ([]bson.M, error) {

    setStage := bson.D{{
        "$set",
        bson.D{
            {"upCount", bson.D{
                {"$size", "$upvotes"},
            }},
            {"downCount", bson.D{
                {"$size", "$downvotes"},
            }},
        },
    }}

    setStage2 := bson.D{{
        "$set",
        bson.D{
            {"voteEngagement", bson.D{
                {"$sum", bson.A{"$upCount", "$downCount"}},
            }},
        },
    }}

    groupStage := bson.D{{
        "$group",
        bson.D{
            {"_id", "$shopId"},
            {"shopName", bson.D{
                {"$first", "$shopName"},
            }},
            {"voteEngagement", bson.D{
                {"$sum", "$voteEngagement"},
            }},
        },
    }}
```

{“$set”: {
“upCount”:{“$size”:”$upvotes”},
“downCount”:{“$size”:”$downvotes”}
}},
{“$set”: {
“voteEngagement”:{“$sum”:[“$upCount”,”$downCount”]}
}},

{“$group”: {
“_id”:”$shopId”,
“shopName”:{“$first”,”$shopName”},
“voteEngagement”:{“$sum”:”$voteEnagement”}
}},

```
lookupGroupAndScoreAppointmentsStage := bson.D{{
    "$group",
    bson.D{
        {"_id", "$shopId"},
        {"appointmentEngagement", bson.D{
            {"$sum", 5},
        }},
    },
}}

lookupScoreAppointmentsPipeline := bson.A{lookupGroupAndScoreAppointmentsStage}

lookupScoreAppointmentsStage := bson.D{{
    "$lookup", bson.D{
        {"from", "appointments"},
        {"localField", "_id"},
        {"foreignField", "shopId"},
        {"pipeline", lookupScoreAppointmentsPipeline},
        {"as", "appointmentEngagementList"},
    },
}}
```

lookupScoreAppointmentsPipeline:[
{“$group”: {
“_id”:”$shopId”,
“appointmentEngagement”:{“$sum”:5}
}},
]

{“$lookup”:{
“from”:”appointments”,
“localField”:”_id”,
“foreignField”:”shopId”,
“pipeline”:lookupScoreAppointmentsPipeline,
“as”:”appointmentEngagementList”
}},

```
setStage3 := bson.D{{
    "$set",
    bson.D{
        {"engagementScoreElem", bson.D{
            {"$arrayElemAt", bson.A{"$appointmentEngagementList", 0}},
        }},
    },
}}

setStage4 := bson.D{{
    "$set",
    bson.D{
        {"engagementScore", bson.D{
            {"$add", bson.A{"$voteEngagement", "$engagementScoreElem.appointmentEngagement"}},
        }},
    },
}}

projectStage := bson.D{{
    "$project",
    bson.D{
        {"_id", 0},
        {"shopName", "$shopName"},
        {"engagementScore", 1},
    },
}}

sortStage := bson.D{{
    "$sort",
    bson.D{
        {"engagementScore", -1},
    },
}}

cur, err := r.DB.Collection("reviews").Aggregate(ctx, mongo.Pipeline{setStage, setStage2, groupStage, lookup
if err != nil {
    return nil, err
}
```

{“$set”: {
“engagementScoreElem”:{
“$arrayElementAt”:[“$appointmentEngagementList”,0]
}
}},
{“$set”: {
“engagementScore”:{“$add”:[
“$voteEngagement”,
”$engagementScoreElem.appointmentEngagement”
]}
}},
{“$project”:{
“_id”:0,
“shopName”:”$shopName”,
“engagementScore”:1
}},

```
{“$sort”:{
        “engagementScore”:-1
}}
```

4.8. MongoDB Indexes and Performance Analysis

While using a database, **Indexes** are needed in order to speed-up complex operations and/or ensure uniqueness of values. BarberShop's **Mongo** server has several of these **Indexes**, which greatly help in different situations. To demonstrate the effectiveness of the chosen **Indexes**, a **Performance Analysis** was conducted. An **Index** on the **Shop's Location** field is mandatory in order to use MongoDB's `**$near**` operator, thus it will be excluded from the performance tests.

The performance analysis has been performed using the **Golang test framework**, which provides a benchmark environment to test functions runtimes.

```
run benchmark | debug benchmark
func BenchmarkIndexes(b *testing.B) {

    ctx, cancel := context.WithTimeout(context.Background(), time.Minute*2)
    defer cancel()

    cfg, err := config.NewConfig()
    if err != nil {
        fmt.Println("cannot find config")
        return
    }
    mongo, err := mongo.New(cfg.Mongo.Host, cfg.Mongo.Port, "barbershop")
    if err != nil {
        fmt.Printf("mongo-error: %s", err.Error())
        return
    }
    userRepo := repo.NewUserRepo(mongo)
    barberShopRepo := repo.NewBarberShopRepo(mongo)
    adminAnalyticsRepo := repo.NewAdminAnalyticsRepo(mongo)

    err = repo.CreateTestIndexes(mongo, ctx)
    if err != nil {
        b.Fail()
    }
    users, err := userRepo.List(ctx, "")
    if err != nil {
        b.Fail()
    }
    barbershops, err := barberShopRepo.Find(ctx, -1, -1, "", 0)
    if err != nil {
        b.Fail()
    }

    fmt.Println("Number of users: ", len(users))
    fmt.Println("Number of barbershops: ", len(barbershops))
```

Setup the benchmark environment

```go
b.Run("user-GetByEmail-with-index", func(b *testing.B) {
    for i := 0; i < b.N; i++ {
        _, err := userRepo.GetByEmail(ctx, users[len(users)-1].Email)
        if err != nil {
            fmt.Println(err)
            b.Fail()
        }
    }
})

b.Run("barbershop-Find-with-index", func(b *testing.B) {
    for i := 0; i < b.N; i++ {
        _, err := barberShopRepo.Find(ctx, 41.9027835, 12.4963655, "", 10000)
        if err != nil {
            fmt.Println(err)
            b.Fail()
        }
    }
})

b.Run("shopEgagementRanking-index", func(b *testing.B) {
    for i := 0; i < b.N; i++ {
        _, err := adminAnalyticsRepo.GetEngagementShopRanking(ctx)
        if err != nil {
            b.Fail()
        }
    }
})

err = repo.DropTestIndexes(mongo, ctx)
if err != nil {
    b.Fail()
}
```

Benchmarks with indexes

```go
b.Run("user-GetByEmail", func(b *testing.B) {
    for i := 0; i < b.N; i++ {
        _, err := userRepo.GetByEmail(ctx, users[len(users)-1].Email)
        if err != nil {
            b.Fail()
        }
    }
})

b.Run("shopEgagementRanking", func(b *testing.B) {
    for i := 0; i < b.N; i++ {
        _, err := adminAnalyticsRepo.GetEngagementShopRanking(ctx)
        if err != nil {
            b.Fail()
        }
    }
})
}
```

Benchmarks without indexes

The benchmarked function is executed multiple times by the testing framework, and the execution time is measured.
The variable b.N contains the number of iterations that should be performed for the benchmark. By default, Go automatically determines a suitable value for b.N based on the benchmark's execution time.

The benchmarked **Indexes** were:

- “**email**” on **Users**
- “**location**” on **Shops**
- “**shopId**” on **Appointments**

Results:

```
Number of users:  6048
Number of barbershops:  2451
goos: linux
goarch: amd64
pkg: github.com/just-hms/large-scale-multistructure-db/be/internal/usecase/repo
cpu: AMD Ryzen 7 6800HS Creator Edition
BenchmarkIndexes/user-GetByEmail-with-index-16              8047          149448 ns/op
BenchmarkIndexes/barbershop-Find-with-index-16               789         1631020 ns/op
BenchmarkIndexes/shopEgagementRanking-index-16               8       129056744 ns/op
BenchmarkIndexes/user-GetByEmail-16                        586         2130349 ns/op
BenchmarkIndexes/shopEgagementRanking-16                     1  78497640308 ns/op
PASS
ok      github.com/just-hms/large-scale-multistructure-db/be/internal/usecase/repo      84.603s
```

In conclusion:

- “**email**” on **Users**: **14.25x** speedup
- “**location**” on **Shops**: **necessary**. Allows very fast queries even at a 100km radius
- “**shopId**” on **Appointments**: **608.24x** speedup in the most demanding Aggregation.

5. Conclusion

5.1 Closing Words

Through the use of the most recent technologies, **BarberShop** shaped up as a production-ready application that is up to standard with real world usage. The power of its Databases allowed it to have fast, reliable, and easy access to the high volumes of data that a platform of this type needs. As a whole, the team is very proud of what has been achieved, and it has been a learning experience that taught everyone valuable “real-world” skills.

5.2 Expandability

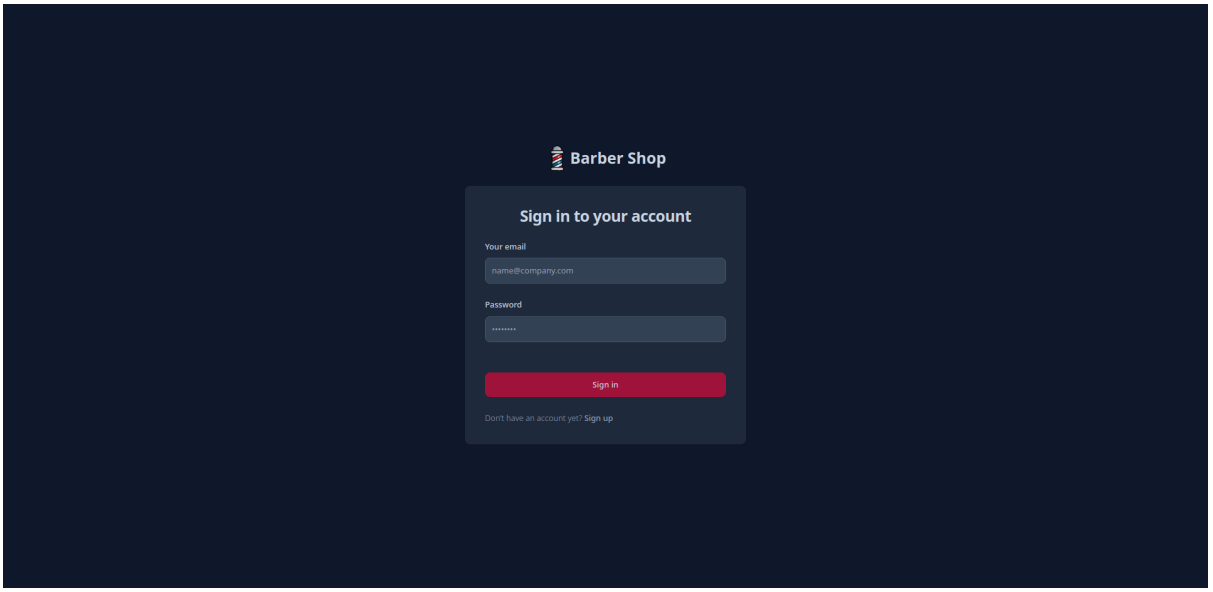
As it was previously mentioned, **BarberShop** would be a prime target for **Geographical Sharding** of its data, due to the nature of the application. New servers, depending on load, could be easily spun up due to **Docker** and its **Contexts**, due to the fact that they don’t require the remote targets to have any code on them at all, and come with every necessary program pre-installed in them.

Thanks to BarberShop’s extensive capabilities of **Data Analytics**, a proposed useful feature would be automatic **reward programs** for loyal customers, and new **advertisement** instruments for **Barbers**.

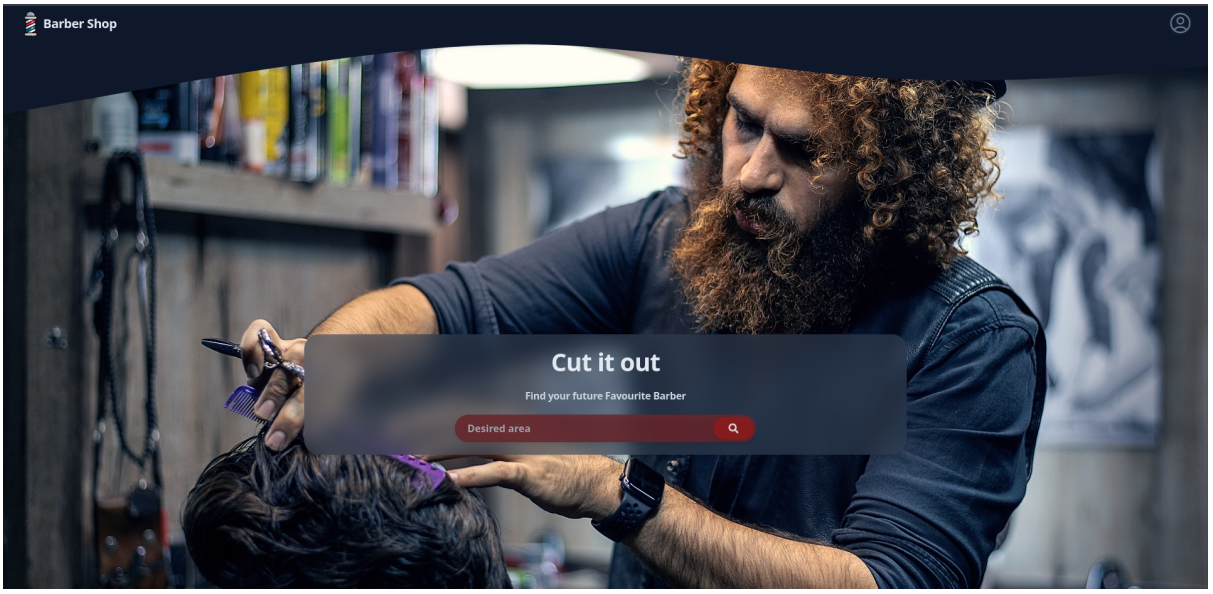
6. User Manual

Home page

The landing page for the application is a simple login form, which is the same for all users. Users can sign in on this page or register on the sign-up page linked in the form. The code for login and registration can be found in *fe/login_form.tsx* and *fe/signup_form.tsx*.



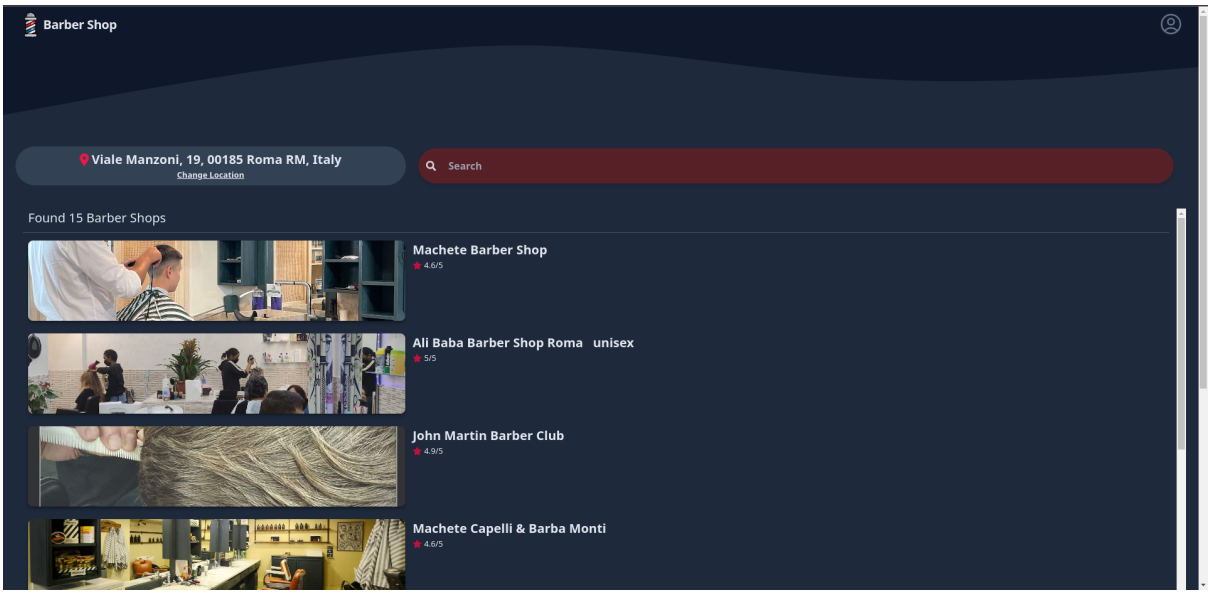
After signing in or registering, all users land on the homepage, which includes a simple search bar and the profile icon on the top right. Clicking the profile icon leads to the personal space.



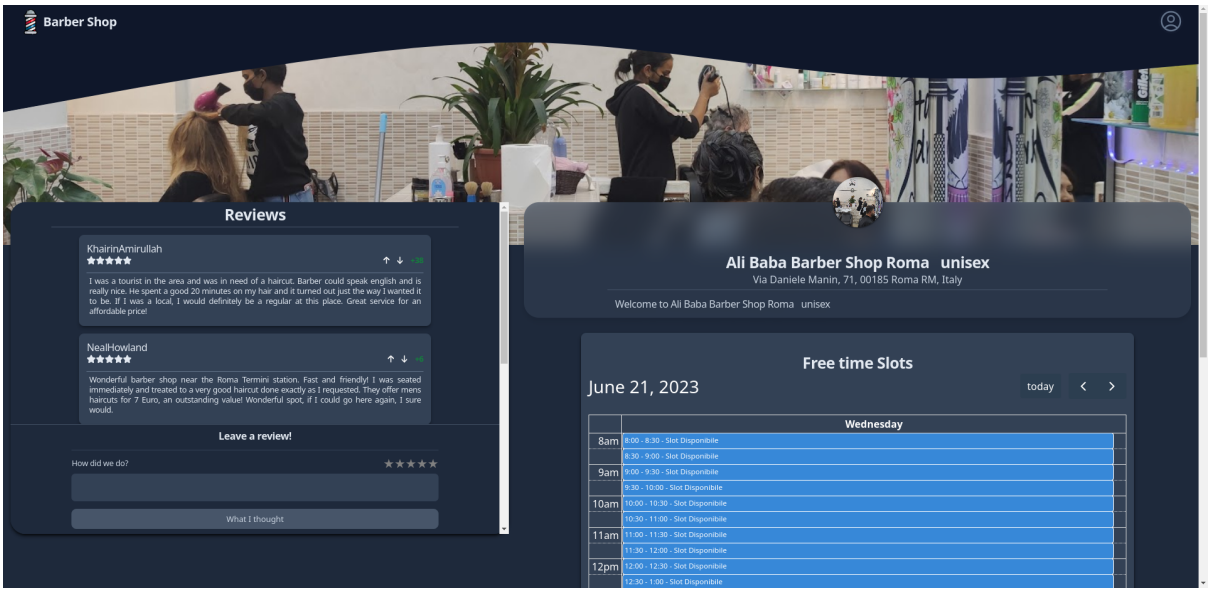
By searching for a location, the user is redirected to the search endpoint, implemented in *fe/pages/search.tsx*. This endpoint performs all the necessary fetch operations to geocode the user-provided location and search for nearby shops in the database.

Search

Once on the search page, the app displays all the shops found near the provided location, along with brief information about each shop.



By clicking on a shop, the user is redirected to the shop endpoint, which shows detailed information about the shop. On this page, users can leave a review for the shop, book an appointment, and have a global view of the shop itself.

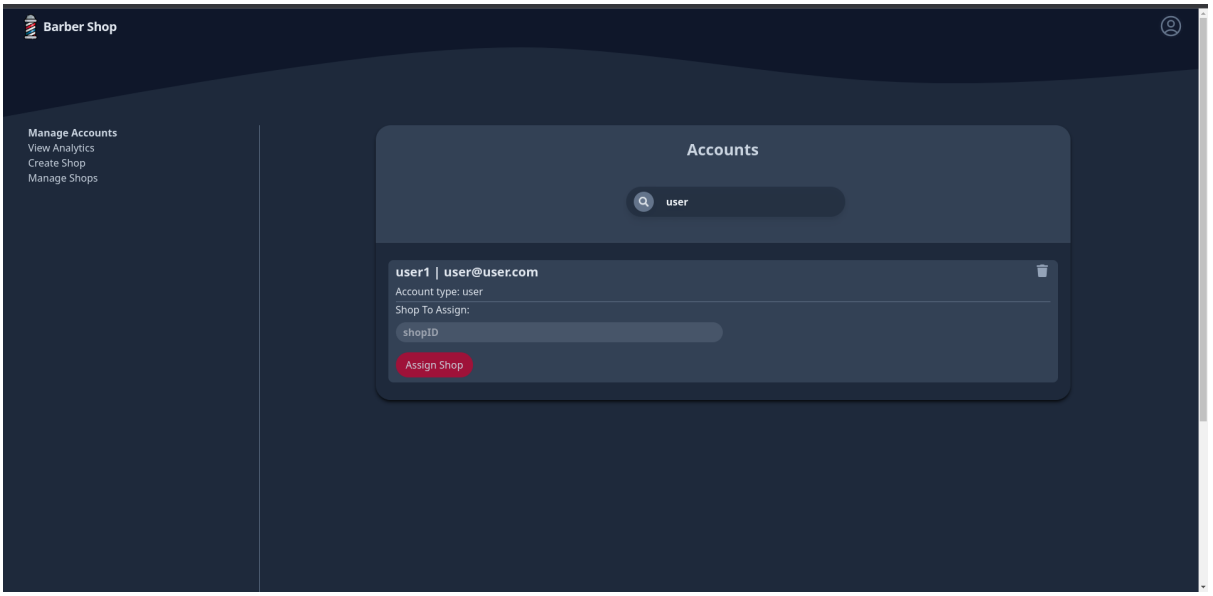


Account management

By clicking on the top right user icon, each user can access his/her personal profile page. Each type of user has a different landing page.

Admin

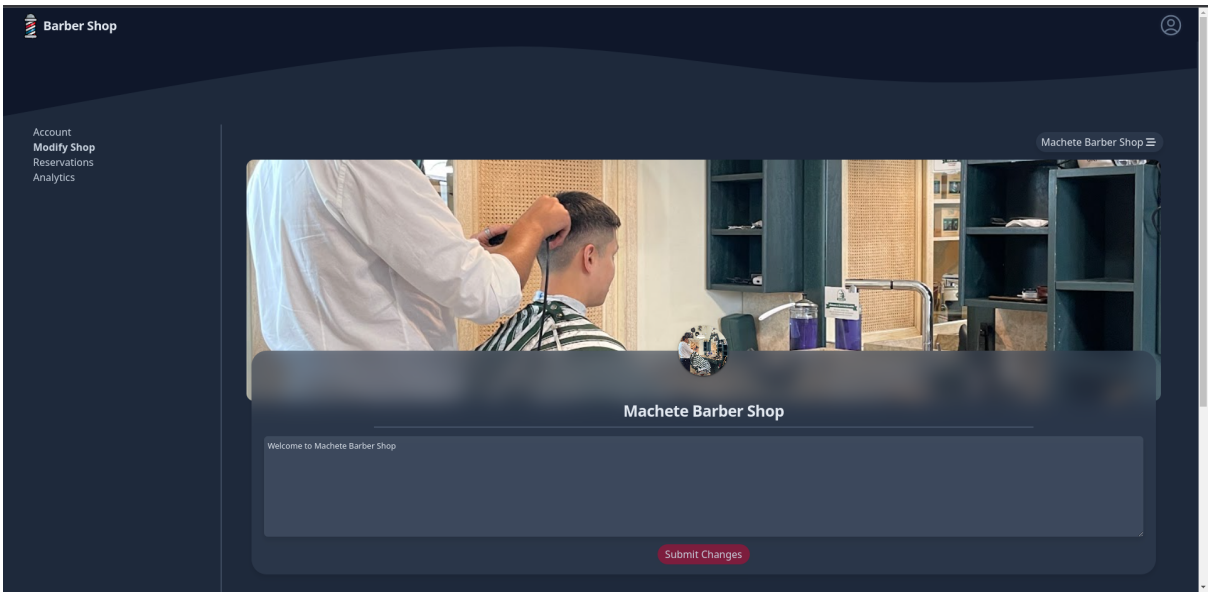
The admin panel allows the admin to manage shops and users, including deleting them and assigning shops to their own barbers. The most relevant section of this panel is the analytics, which provides the admin with all the analytics regarding the website.



The most remarkable implementation here is the analyst data visualization:
Implemented in */fe/components/chart_components/*,by the mean of *chart.tsx* and *list_chart.tsx*.
This code plots the values of the required analytics paginated in pages composed of 30 elements each, for readability purposes.

Barber

Similar to the admin, the barber has their own landing page, which displays analytics regarding the shops they own. They can also change the description and the number of employees for each owned shop.



User

The user is provided with basic information regarding his profile, as every other type of user is, and with the current appointment booked, if present.