World Cities and Places - A Microservices Application

Abstract goes here

Donald Knuth

Dennis Ritchie

B.Sc.(Hons) of Science in Computing in Software Development

May 7, 2020

Final Year Project

Advised by: Dr Alan Turing
Department of Computer Science and Applied Physics
Galway-Mayo Institute of Technology (GMIT)

About this project

Abstract A brief description of what the project is, in about two-hundred and fifty words.

Authors Explain here who the authors are.

Contents

List of Figures

Chapter 1

Introduction

Jose I. Retamal

This project has been developed as part of the BSC(Honours) in Science in Computing in Software development course for the module minor project and dissertation on the last year of a fourth-year course. The module corresponds to 15 credits out of 60 in the year.

Two students have developed the project, which has been divided into two parts: front-end and back-end, Elena was is developed the front-end and Jose the back-end.

We have developed a microservice application using go as a principal back end programming language, with a react native client and a REST-API(Figure 1.1).

1.1 Objectives

We expect to learn and prepare to afront the professional software environment after doing this project. The main objective of this project is to learn as much as we can to be ready to start our professional path in the software industry. Some of the objectives that we expect to achieve by doing this project are:

- Research and learn new technologies that are utilized in the software industry.
- Apply agile techniques to develop an application based on initial requirements.
- Develop a scalable microservices distributed system to manage a dynamically growing database.

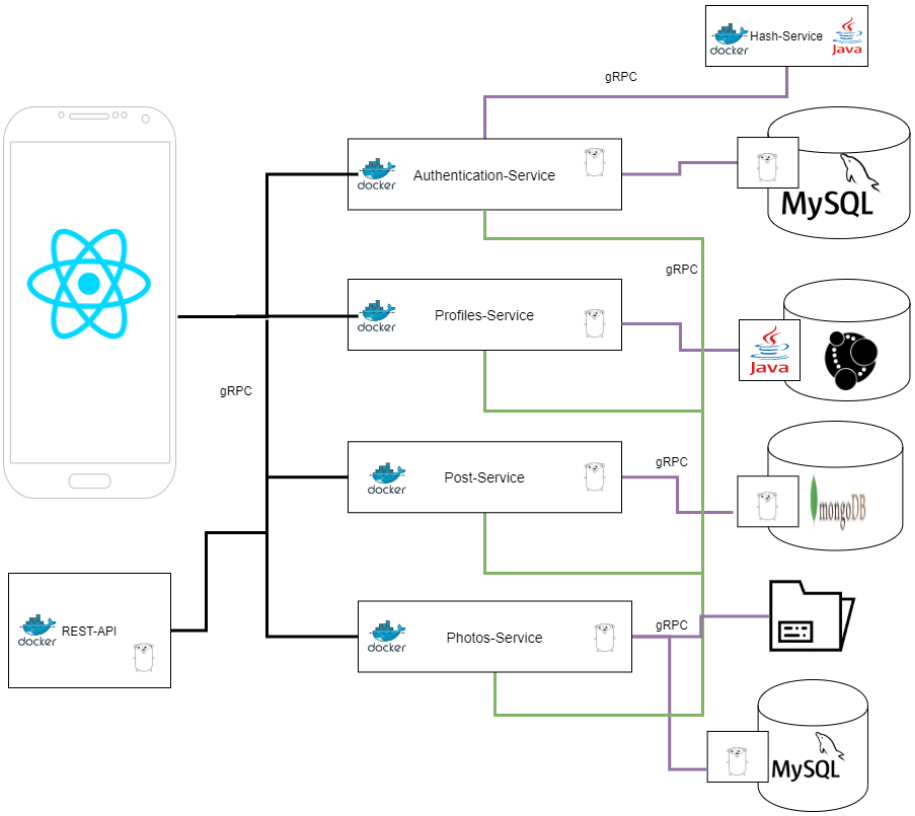

Figure 1.1: Application - UML.

- Implement the system on the internet using the most popular cloud providers. Develop an application that efficiently works with images.
- Implement a reliable authentication system that securely stores passwords using hash and salt.
- Develop a high-performance mobile application that integrates the microservices.
- Develop a REST API that integrates the microservices.
- Works in a team utilizing flourishing software development tools.

1.2 Design overview

In this project, we utilize cutting-edge technologies to develop a microservice application. The application is developed using relational, document base, and graph databases. We look at the benefits of each database, and we use them where we can maximize performance base on how the data is store and access on each type of database.

The advantages that we want from developing the application as a distributed system are [1]:

- Avoid expensive queries for fast performance. We use multiples databases to store the data in a simple way that makes access very fast. We avoid joins and any expensive query.
- Scalability, each component performs a specific task; more instances of each service can be added.
- Flexibility, each service is independent and composed of independents components as well; more components to perform a different task can be added on each service can be added, and new services can be created for a completely new task.
- Maintainability, services are easy to maintain because they are encapsulated and perform simple tasks.

1.2.1 The Solution

We have designed a tourism application where users create the data. It stores data about three main types: users, cities, and places; all three have a public profile(public to register users of the application). Users create the cities and the places, and they can be created only once. Cities and places have a profile where users can post. Users can mark Cities and places as visited, so they will show in their profiles and have actualization about recent posts.

The principal user interfaces in the client are:

- User profile, which shows visited places and visited cities.
- Cities profile, which includes the places in the city and posts about the city.
- Places profile, which includes posts about the place.

1.2.2 The Microservices

The application is divided into four main microservices:

- The authentication service provides secure authentication storing passwords in binary format using hash and salt.
- The profiles-service store primary data of all profiles.
- The post-service, store, and manage posts.
- The photo-service, manage, and stores photo for profiles and posts.

1.2.3 Databases Design

Each service has its database. We have research best database that fits each of the services:

- The authentication service uses a relational database. When a user login a token is generated, that token can be used to authenticate each request that the user performs. A leader/slave replication has been set up to improve performance. The read from the database is distributed on multiple databases to perform the most used operation that is to authenticate requests.
- The profiles services use a graph database. We used the graph database to create relations that avoid complicated queries.
- The post-service use a document-based database. Posts are indexed and stored in scalable documents. We can use inexpensive queries to perform CRUD operations in the posts of a city or place.
- The photo-service store images in the file system and use a relational database to store URLs. Binary images have public access from the file system, so the image is loaded directly from the client without the need to send the image through the service, the service store URL which is sent to the user.

1.2.4 Technologies introduction

After some research, we decide to use Goland as the main back end programing language; some parts of the system were also writing using Java.

We have used gRPC as the communication interface for the microservices because it provides a transparent client/ server relation. We created a go module with all gRPC interfaces that can be imported in all the components of the application.

All the services are run using docker containers, and the images are published in Docker Hub and then pulled from virtual machines to run the service.

We have work with the 3 most popular internet services providers(Azure, google cloud, and AWS), all of them offer free student credit, and we use that. In general, the 3 services have excellent performance and outstanding support, which was used several times to learn and fix errors.

We find that one of the best for us was google cloud because they live chat and convenient SSH on the browser. Also, the container optimized OS is outstanding to work with docker images.

1.2.5 Methodology

The project has been developed in an iterative approach based on agile principles, based on the original principles we have created an adaptation to feet the demands of our project.

We define stages that are reviewed during all the project development, measure progress based on the working code, continuously meet with the team, and be ready to adapt to any change in circumstances.

The solution was created component by component, integrating them after several independent testing, we keep a working solution all the time to which we integrate more components as they are developed.

1.2.6 Context

In Chapter 2, we explain the methodology used to develop the project. Chapter 3 is a technology review, where we research and decide the best technologies to use in each component for then have an in-depth look at them. In Chapter 4, we explain the full system design and the design of each element. Chapter 5 is the evaluation of the system; we show how it works and test it. And finally, in chapter 6, we provide a conclusion where we look at the achieved goals and possible future work.

Chapter 2

Methodology

2.1 Back End

Jose I. Retamal

Most successful software projects are developed using some agile methodology(http://agilemanifesto.org/). There is a reason for this and is because the software is an intellectual product that needs to adapt to change on requirements, software dependencies, and hardware. This constant change required an adaptative and iterative way to develop.

We adapted some of the agile principles for our project:(http://agilemanifesto.org/principles.html)

- Continuous delivery. The application is tested at all stages, and we keep a working application all the time.
- Adapt to changes at any stage. Even if we started with a design and chose the technologies to be use, they can change at any stage.
- Work together with all participants. We continuously meet with all people involved in the project, check the stage of the project, and check if there is anything that needs to change.
- Self-motivation. There is a personal interest in the project to all participants.
- Working software is the measure of the stage of the project. What is done and working in the software is the main reference point on at what stage we are on the project. Simplicity. We develop a working application most simply, but without leaving the performance aside.
- Self-organizing. Everyone in the project organizes himself in the way they think it would be the best way to have a maximum amount of productivity.

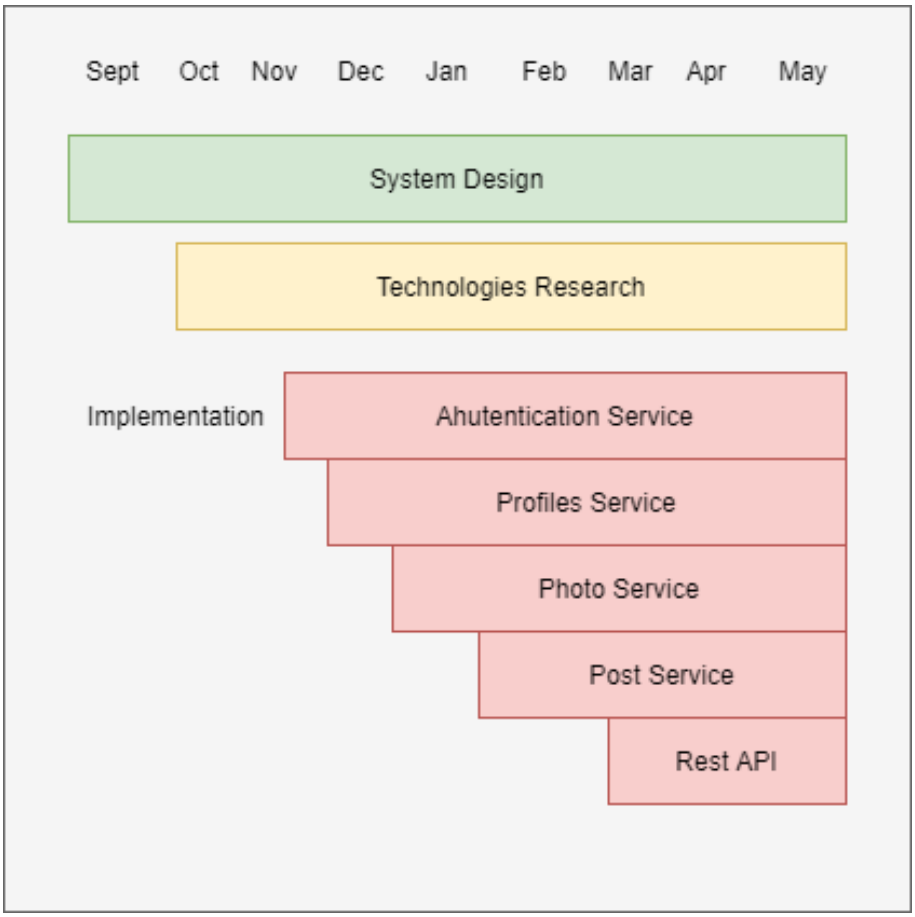

Figure 2.1: Methodology- Project Gueneral Gant Chart.

Stages of the Project

There where three stages in the development of the application. They are not a hard stage, and they overlap each other(Figure 2.1). These stages are a reference to know what to do, and they are regularly reviewed.

- System Design.
- Technology Research.
- Implementation.

System Design We create a simple design of the system at the start. This defines, in a general way, the main components of the system. This was a quick process, and the design was always subject to updates. To

design the system, first, the requirements where set. Then the design was done following them.

- Design the full system in a general way, understand more or less the number of services required.
- Define the function of each component.

Technology Research After we have an idea of the system, some time was spent on looking for the best technologies that will suit the application, some prototypes where design to check if they work.

- Decide the main technologies to be used.
- Test compatibility.

Implementation The implementation has been done using an iterative process. Starting from the authentication service, the system was build block by block. After one component was developed, it was tested to check that the full system works.

On each component, we start by designing the database, the DBA, and then the main service with the endpoint to the client.

When starting to develop a new component, the full system design has to be check.

2.1.1 System Development life cycle

Each component is developed iteratively after some functionality is implemented the component is first tested locally, them docker images is created ann updated to docker hub, them the image is downloaded in the server, and the whole system is tested. The life cycle steps are below(Figure 2.2):

- Build and test the component locally.
- Create/update a docker image and update it to the docker hub.
- Pull from docker hub, run the component in the network, and test.
- Test the full system.

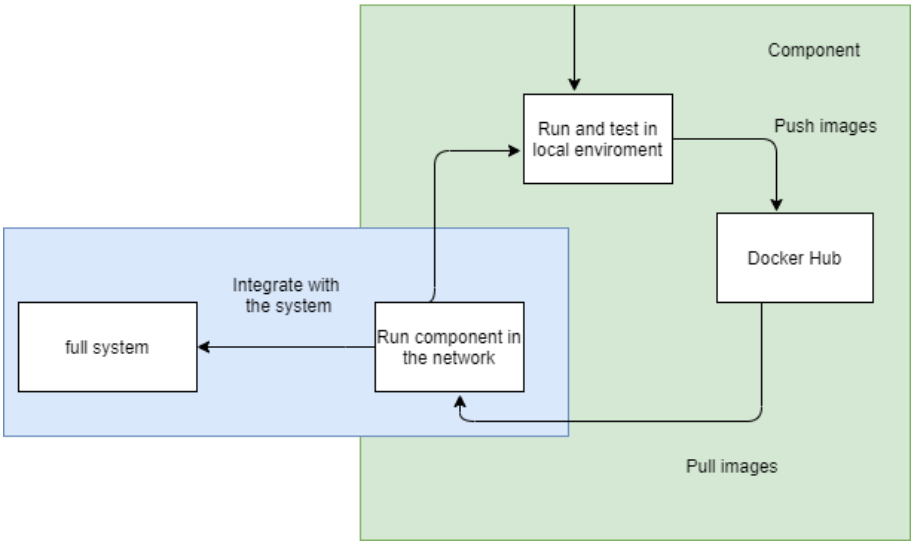

Figure 2.2: Methodology- Component Iteration.

2.1.2 Testing

Because of the time frame, we do not develop the application using automated testing. We know there are many advantages to it, but we need to trade off this feature. Some aspects that we considered to decide that automated testing is not for the project: Most of the benefits of automated testing are long time benefits. Automatested testing is practical another application that needs to be maintained.

Instead, based on the principle of self-motivated development, we develop the project using a test culture that relies on developers: Each functionality needs to be tested by the developer at developing time. Bugs found needs to be personally reported to the developer. When integrating new components, the full system needs to be checked.

Chapter 3

Technology Review

3.1 Technologie Research and Decision

Jose I. Retamal

We start with some research about the best technologies that can be used to develop the system. The technologies that we need are:

- Backend programming language.
- Communication interface.
- Databases for each service.
- How to store images
- Client technologies.
- How and where run the services.

The factors we take into consideration to choose the technologies are:

- Accessibility. Technologies must be accessible without a cost; they can be free and open-source or offer student free usage.
- New to us. One of the ideas of the project is to learn how to work with new technologies and learn. So we look for technologies that are not familiar to us.
- Popular and used in Professionally used. We want to learn how to use technologies that are popular and used in professional software development.

- Mature and recognized. We prefer technologies with background and maintained/developed by a recognized organization or a big open source community. Even if there are always new technologies coming out, we prefer those that are already established.
- It adapts well to the purpose need in the system. Simple and with resources to learn. We need to learn the technologies, so we instead chose those that are fast to learn and with many resources(tutorials and documentation) to access.

3.1.1 Backend Programing Language

Jose I. Retamal

We want a simple, with excellent performance, fast and straightforward to learn programing language. Go is a modern programming language that was specially designed by Google to develop scalable software systems. After reading some forums where developers gave opinions about how it is to work with go, we have extracted the advantages and disadvantages of it.

https://builtin.com/software-engineering-perspectives/golang-advantages
https://www.pluralsight.com/blog/software-development/golang-get-started
advantages

- Go was specially designed to make software designed easier.
- General-purpose back end language.
- Simple to understand can be pick up quickly by new developers.
- Go is designed to be simple, avoiding having many features and flexibility to be simple and very practical.
- Enforce a coding style.
- Simple concurrency primitives.
- Very fast compiling time and small usage of memory.
- Very mature libraries for gRPC.
- Static typing that speed up developing time.
- SDK for all popular cloud providers.
- Built-in testing and benchmarking.
- Fast publish of libraries through Github.
- Scalability, go was designed by Google with scalability in mind.

- Go was designed by experienced developers having in mind the industry requirements and taking into consideration the developer's experience.

Disadvantages:

- Relative young programing language.
- Code with no-frills means that you write more code than other modern programming languages like Ruby.
- Go has no classes, so there is a need to think about the design differently. Go is not an object-oriented programing language.

Most of the disadvantages have a positive aspect, as well. Writing more code sometimes makes it more transparent and easier to debug. Also, having to design the program in a not object-oriented way make the design more simple and straight forward. Goland fits all the requirements we are looking for the backend programing language, and we think it will be excellent to learn and based on other experiences that seem like it is excellent for backend, and we found that many big companies used it as primary backend programing language.

3.1.2 Communication Protocol

Jose I. Retamal

We need a protocol for the microservices to communicate. We have considering using REST or gRPC.

We need many services to communicate in a fast and reliable way, and they would all share the same network that is the internet.

Some advantages of gRPC over REST are:We have compared gRPC with rest and found advantages of gRPC over REST for our system:

- CLear defined Interfaces.
- Bidirectional streaming.
- Much faster and compact.
- East to understand and implement.
- Lossy coupled between client/server.

The interface is defined using Protobuffers, which is programming language independent. The same code is compiled for the different programming languages, it has libraries in most programming languages, and it does have with Go and Java that we will use in the application.

So gRPC is a modern implementation of RPC that has been developed by Google to meets requirements of high traffic distributed systems. It uses HTTP and is fast to learn and easy to implement.

3.1.3 Ahutentication Service Database

Jose I. Retamal

We need a database on where the data can be accessed quickly and reliably by index. We are also looking for a database that can be replicated. A relational SQL database fits the requirements. From MariaDB, MySQL, PostgreSQL, and oracle. MySQL and PostgresSQL have been selected for further analysis.

MySql

MySQL 5.6 [2]

Pros

- Oracle has brought more investment into MySQL, meaning there is a future with it.
- It is a solid product. MySQL 5.6 is a reliable product with all the features fully tested.
- There are many teams from Oracle working in MySQL.

Cons

- Not so mature as other relations databases like PostgreSQL. This means that it has fewer features than the other more mature database systems.
- Not fully open source anymore, in theory, is open source, but in practice, Oracle has taken over.
- Many have replaced MySQL for MariaDB. Since oracle takes over MySQL, many big names like RedHat Enterprise Linux, Fedora, have moved to MariaDB.

PostgreSQL [3]

Pros

- Reach libraries for transactions. Fully documented with comments made it easy to know what part of the code does what and how it is done.
- Many adjustable parameters make the system easier to personalize.
- Easy to extend, if extra features are needed is possible to add the feature. Secure and reliable, also security can be personalized and extendible.

Cons

- Open source, not owned for any organization that takes care of it.
- Slow performance, there have been reported issues with performance and backup recovery.

Conclusion There is not a real need for a feature-rich database; the database that we need to implement is simple, and the main aspect to consider is the replication—also, some of the cons of MySQL that be considered as advantages. Because MySQL is robust and has all the functionality needed, it has been chosen as the best option.

3.1.4 Profiles Service Database

Jose I. Retamal

We need a database that store names and description of the users, cities, and places. Also, we need to access places that are in a city, and users need to be able to mark any city or place as visited for, then get info and make comments about the places that they have been.

It makes sense to have some relationship between the users and the places/cities to connect them. Also, a relationship between the place and city can be used.

We Considered three types of databases: relational, document, and graph database. If we chose the relational database, we would need to have a table where each user has entries for the cities and places they visit. To get the data, we will need to perform expensive joins. If in the future we want to add more relations like friendships, for example, it will get more complicated, and queries will be more expensive.

If we chose a document database, we could have like a list of places and a list of cities for each user, this would be easy, and queries would be not so expensive but there would it would use much more storage.

For a graph database relationship are natural, it has flexibility and would be much easier to add extra relationships and also more fields on each node if necessary. The performance would be good, and the relations will not produce the use of extra storage as with a document-based database [4].

Considering this, we have chosen a graph database to store the profiles, the key advantages this type of database will give are:

- Performance, relationships are natural for a graph database; queries would be no expensive.
- Flexibility, more fields, or relationships can be added in a relatively easy way.

Considerations

Some accessible graph databases are TigerGraph, Neo4j, and DataStax. They all have great performance and offer more or less the same functionality [5]. We have chosen Neo4j because it has a more significant community and is more popular, meaning that there is more online documentation and resources.

Database access driver

After we have decided what database would be used, we need to choose the programing language and the driver to access it. Given the main programing language of the system is going, we have considered this programming language. We also research java for the driver.

Drivers considered:

- Go -https://github.com/johnnadratowski/golang-neo4j-bolt-driver
- Java- https://github.com/neo4j/neo4j-java-driver

Decision

Neo4jdatabase with the official java driver.

3.1.5 Store Images

Jose I. Retamal

We have considered a few ways to store the images.

- Store image in MongoDB using Gridsfs https://docs.mongodb.com/manual/core/gridfs/
- Store images in MySQL database using binary blobs.
- Store in the file system and store URL in a database.

The best way to store the images is to use the file system and store the URL in another database, because:

- Fewer data need to be transferred. If we store the image in a database, it would need to be retrieved from the database and then sent to the user. So the service will need to deal with a huge amount of data. If we just store the URL and allow the user to access the file system, the service will need to deal with a much lower amount of data. Resulting in better performance.
- Much easier to escalate. More buckets can be easily added, and the amount of data store in the database is meager and can be stored in a very simple way, just as single tables without the need to join data.
- Internet services providers(Google Cloud, Azure, and AWS) supply ready to use file systems that can have public access: buckets. They are cheap, have high performance, and are scalable.

To store the URL, we have decided to use a relational database because we need to access the data using keys.

3.1.6 Post Database

Jose I. Retamal

For store post, we need a database that:

- Can store a huge amount of data, because posts are the things that occur more in the application because cities and places can have "any" amount of post each we expect that they would more posts that anything in the application.
- The post would need to be stored by a city or place, and they would be for one particular city or place. Some sort of indexing would help performance.

In resume, we need a highly scalable database with little structure, on this description fits perfect a document-based database.

We have considered three different document base databases: InterSystems cache, MongoDB, and Apache CouchDB(). We have chosen MongoDB because:

- Simple to use.
- It easy to install in the Linux environment.
- Native replication is included.

- Even if it has fewer features than the others, it is good because we don't really need the extra features.
- It has a go driver to access the database.

About seven to ten pages.

- Describe each of the technologies you used at a conceptual level. Standards, Database Model (e.g. MongoDB, CouchDB), XMl, WSDL, JSON, JAXP.
- Use references (IEEE format, e.g. [1]), Books, Papers, URLs (timestamp) – sources should be authoritative.

3.2 XML

Here's some nicely formatted XML:

```
<this>
  <looks lookswhat="good">
    Good
  </looks>
</this>
```

Chapter 4

System Design

4.1 Ahuthentication Service

Jose I. Retamal

This service provides user authentication. It is composed of three components: the hash service, the database, and the main service (Figure 4.1).

The password is store securely, is hashed and salted using the hash service and then the hash and salt is stored in the database.

The database is replicated using the master follower topology. Create, update, and delete operations are always performed in the master, read operations are performed in followers.

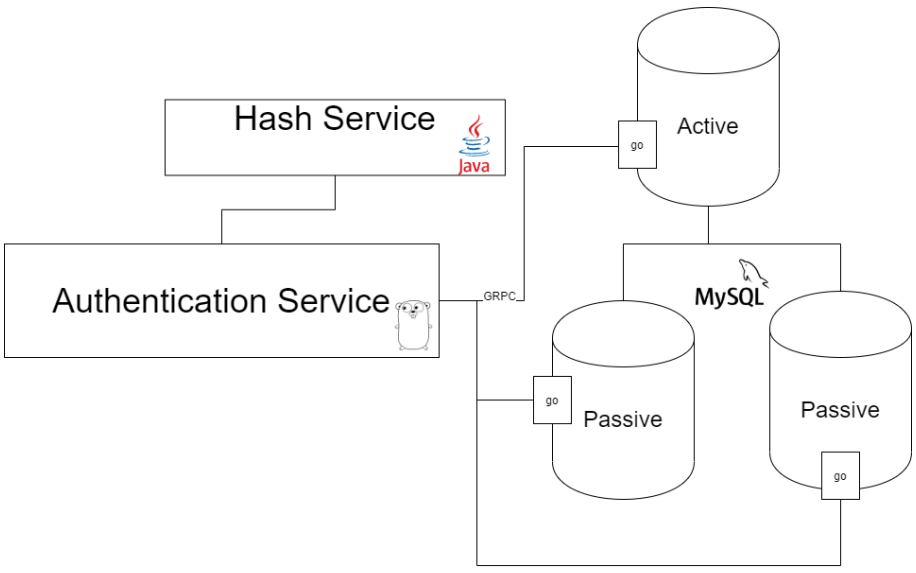

Figure 4.1: Authentication Service- UML.

4.1.1 Database Replication

Replication it has been set up using MySQL 5.7 (See Appendix B.1.4), we have set up a master and a replicate running in different Virtual Machine using Azure services. Using the instruction in Appendix B.1.4 is possible to add more followers, for do tables in master need to be locked for a few minutes.

The load balancing is done using Round Robin, and standards go grpc librarie. helps in performance because the most used operation is to get user data or token to check authentication.

Is implemented as client-side replication, the client chooses a server one at the time to make the calls, the client in this situation is the authentication service, and the server is the authentication dba. The load balancing is implemented in the authentication service(Figure 4.3).

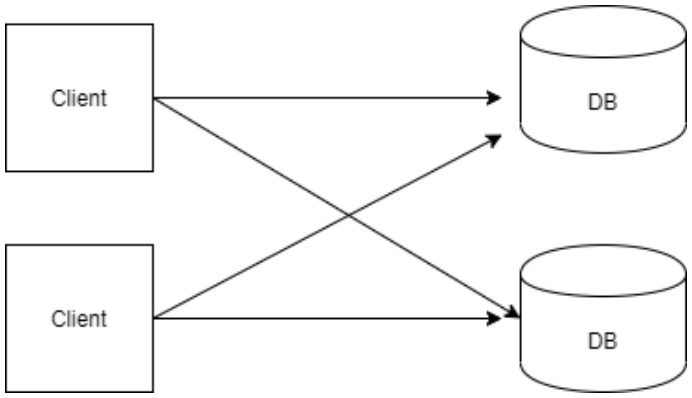

Figure 4.2: Authentication Service- Create User Sequence Diagram.

4.1.2 Endpoints

Create User

Users create an account, and when this happens, a new entry is created in the authentication database. Also, a new entry is created in the profiles database (Figure 4.3). To create a user, this service needs to communicate with the profiles service.

Login user

To login a user, we perform the followings steps (Figure 4.4):

- Get user data from the authentication database.
- Check the password using the hash service.

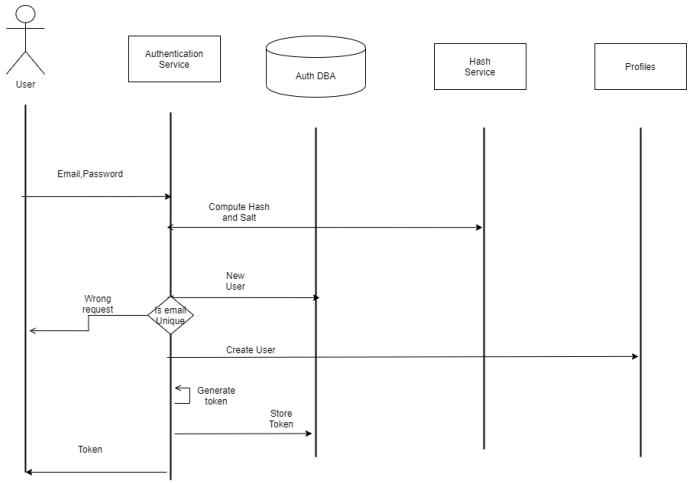

Figure 4.3: Authentication Service- Create User Sequence Diagram.

- If the password is valid will generate a unique token, it stores the token in the database and sends the response to the client.

Check Token

The most used endpoint, here is where replication plays a role for fast checking tokens. This is used in the most requests to all services to ensure security across the application (Figure 4.5).

Log out

The request includes the token, and that token is removed from the sessions table in the authentication database.

4.1.3 Authentication DBA

This program provides access to the authentications database. Is written in go and runs in a docker container, it connects to a MySQL database running in localhost. The application communicates with the main authentication service through a grpc interface.

Database

The authentication database store the necessary user data for authentication and login.

It is composed of two tables: the authentication table and the sessions table. The authentication table contains the user name, the password hash, and the password salt(Figure 4.6).

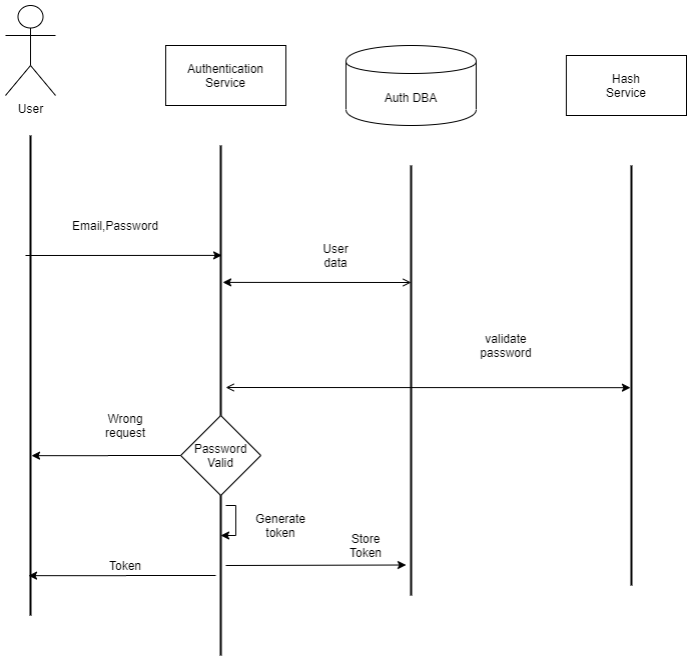

Figure 4.4: Authentication Service- Login Sequence Diagram.

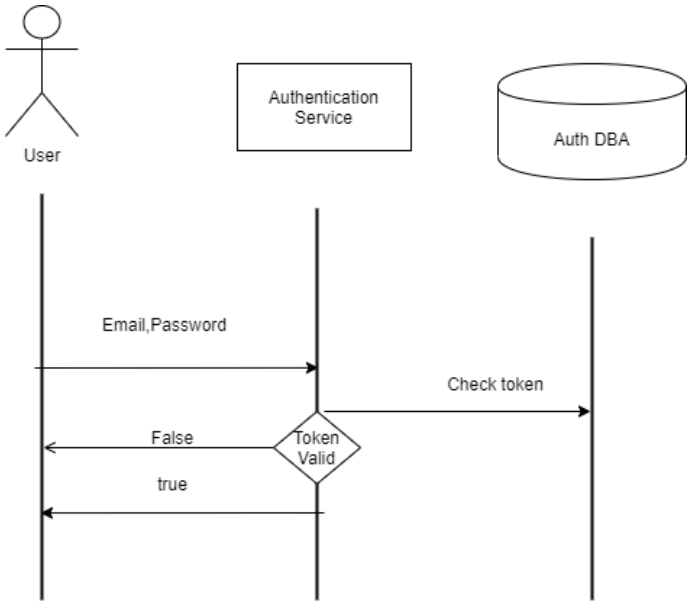

Figure 4.5: Authentication Service- Check Token Sequence Diagram.

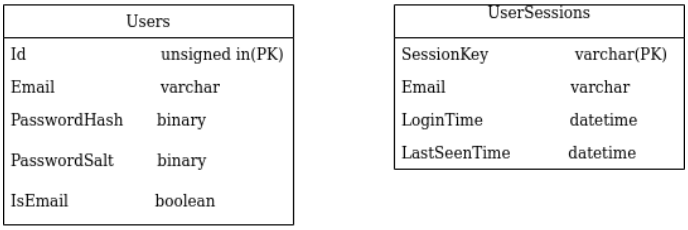

Figure 4.6: Authentication DBA- Authentication Database.

The user name is the email and is the unique identifier for all the systems, is not the primary key of the database, but is indexed for quick access. For this database, an extra integer field is added as the primary key. The password hash and salt are 32 bytes binaries strings. The sessions table uses a unique session key as a primary key for a quick check if a session exists.

When a user login a session is created and stored in this table. The user then can log in to the application using that session. When the user logs out, the session is deleted from the database.

Endpoints

- AddUser() : Create a new user in the database. When the user creates the account, it will create the profile automatically in the profiles database.
- GetUser(): Returns the user data used to authenticate the user.
- UpdateUser(): Update the user data, is used for changing the password.
- CreateSeassion(): Create a new session in the database. Used when the user login using the password.
- GetSeassion() : Return a user session if exist. Used to check if the session exists so the user can log in without the password.
- DeleteSession : Delete user session if exits. Used when the user logs out from the device.

4.1.4 Hash Service

This service creates a password salted password hash using a randomly generated salt. It has been adapted from a Distributed Systems project from semester 7. It checks the password in fix amount of time for security reasons. Attackers can guess passwords guess by comparing the time it takes to validate a password.

4.2 Profiles Service

Jose I. Retamal Jose I. Retamal

The profile service manages information about users, cities, and places. The data contained is editable by the users and provide information about them.

The service is composed of the main service that connects to a Neo4j database using a Java DBA(Figure 4.7). The main service is design to be stateless; therefore, many instances of the service can be created for load balancing and scaling the system.

There are relations between the different types of profiles that allow us to get data quickly and avoid the use of complicated queries.

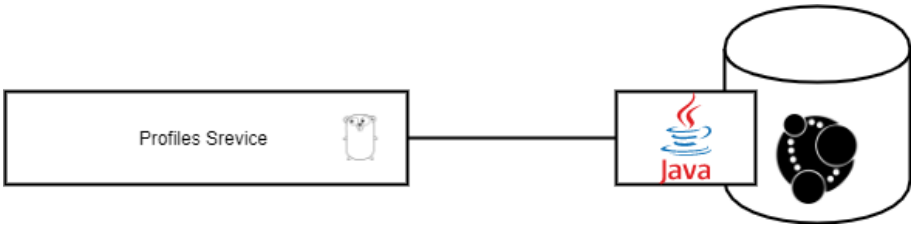

Figure 4.7: Profiles Service - Main UML.

4.2.1 Request Sequence

Each request contains a token that is checked with the authentication service if the token is valid, then the service access the database to serve the request(Figure 4.8).

4.2.2 Endpoints

Users

- Create User. This method is only called by the authentication service because users create an account using that service. The authentication service then sends the user data to the profiles service.
- Get User. Get all data about the user.
- Update a User. Updated the user.

City's Users create the cities; the user who creates a city can then update the data. A city can only be created once.

- Create Cty. Create the city and a relation between the user who creates the city and the city

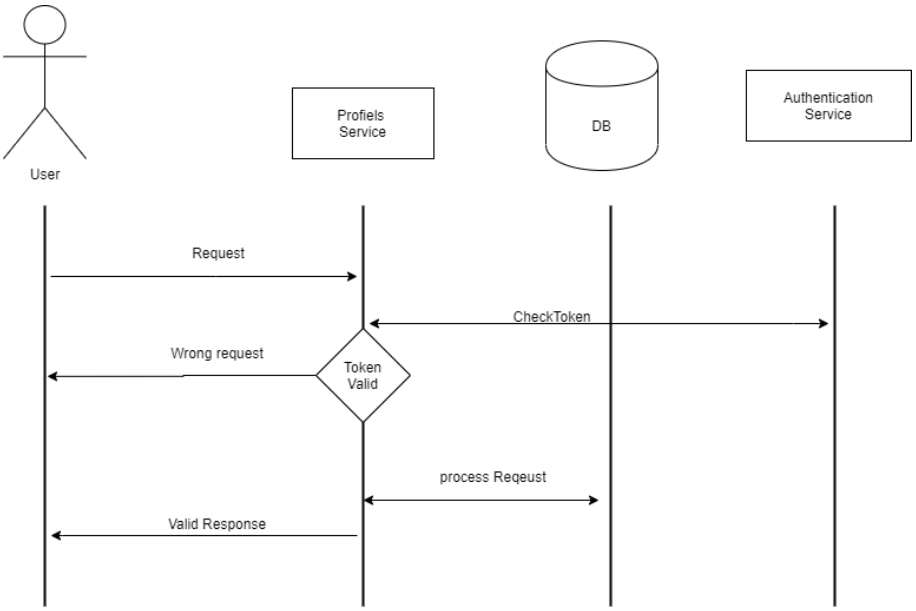

Figure 4.8: Profiles Service- Request Sequence.

- Visit City. Create a relation between the city and the user Update City.
- Get City
- Get all Cities

Places Users create places; places belong to a city.

- Create Place. Create the place and a relation between the user who creates the place and the place. Also, create a relation between the place and the city that belong.
- Visit Place. Create a relation between the place and the user/ Update Place.
- Get Place.
- Get All Place

4.2.3 The Database

The database is composed of three types of nodes: User, City, and Place(Figure 4.9). Each node has a unique integer id. Users are also identified by the email, which is the unique id al over the system. A city can

User	
Id	int (PK)
Email	string
Name	string
Description	string

Place	
Id	int (PK)
Name	string
City	String
Country	string
Lat	float
Lon	float
Description	string

City	
Id	int (PK)
Name	string
Country	string
Lat	float
Lon	float
Description	string

Figure 4.9: Profiles Service- Neo4j DB Classes.

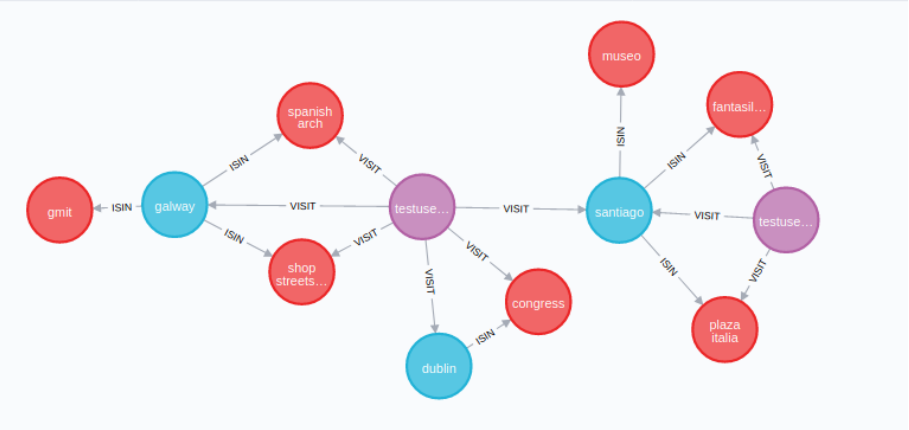

Figure 4.10: Profiles Service- Neo4j DB Dodes.

also be identified by the name and the country, place by name, the city that belongs, and the country(Figure 4.10).

Relations

- Places are in a City. A relation between the city and the place, this relations allows us to retrieve quickly the places that belong to a city.
- Users visit cities and places. Create a relation between the user and the city/place. As users can mark several cities/places as visited, having this as a relation simplified the queries and give quick access to all cities/places that the user has visited.

4.3 Photo service

Jose I. Retamal Jose I. Retamal

The photo-service provides image storage and access to them. It is composed of the main service, the database, and storage buckets. The binary image is store as a public object in the bucket, and the URL to the image is stored in a MySQL database. Images can be accessed from the client using the public URL(Figure 4.11).

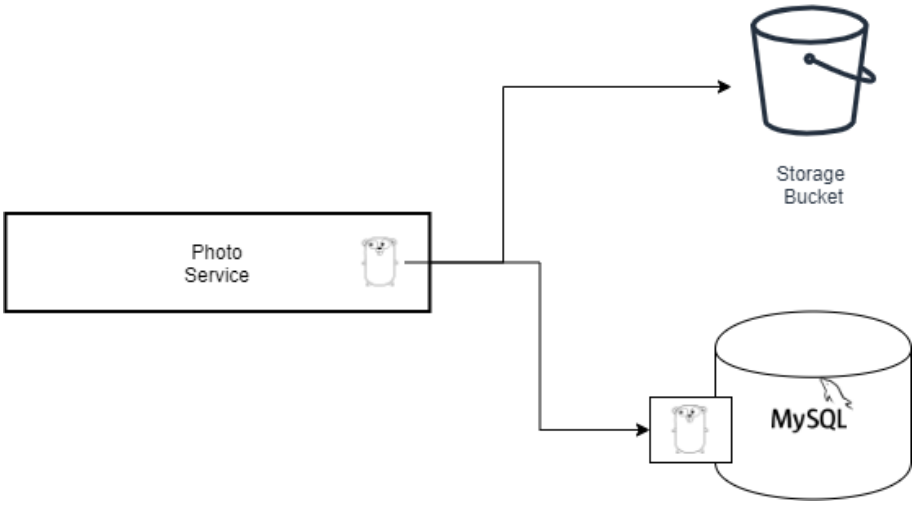

Figure 4.11: Authentication Service- UML.

4.3.1 Request Sequence

Upload Image

When the main service receives a request to upload an image, it generates a random URL, the image is upload to the bucket using that URL, and the URL is stored in the database. Then the URL is sent to the client for access to the image(figure 4.12).

Get Image

When the client requests an image, the client is first authenticated then the image URL is retrieved from the database. The URL is sent to the user who accesses the image directly from the bucket 4.13).

4.3.2 The Database

The database store the URL to the image in the bucket. Images for City, Place, Posts, and User Profile are stored, there is a table for each of them(Figure 4.14).

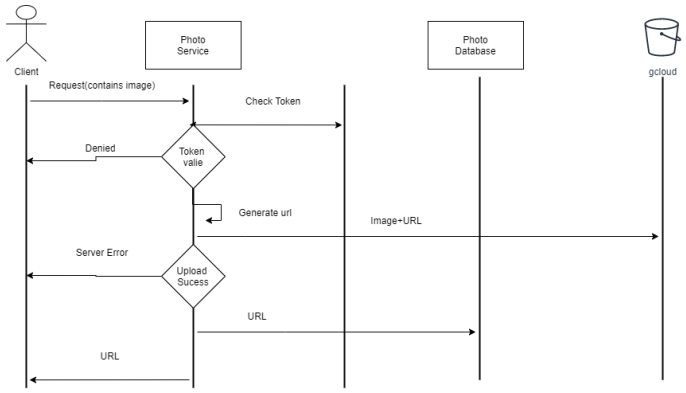

Figure 4.12: Photo Service- upload Image Sequence Diagram.

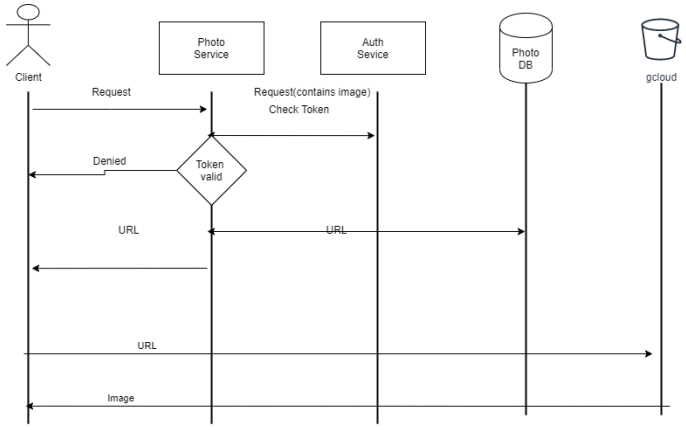

Figure 4.13: Photo Service- Get Image Request Sequence Diagram.

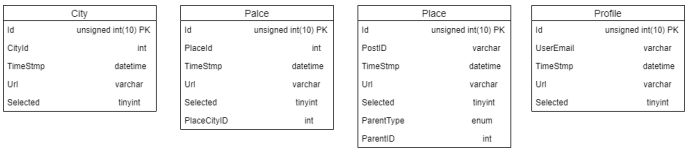

Figure 4.14: Photo Service-URLS Database Entity Diagram.

All tables have an autogenerated integer primary key and each, and all images are also identified depending on the type. Bellow, we explain how they are linked using id from different databases.

- City Images

  The unique id is the CityId, which is the Neo4j unique id.

- Place Images

  The unique id is the PlaceId, which is the Neo4j unique id.

  The PlaceCityId is the id of the city of which the place belongs. It is used for getting all the images of the places in a city.

- PostImages

  The postId is the id of the post in MongoDB.

  There are two types of Posts, city and place post. They are stored in the same table, and an enum is used to distinguish.

- Profile images

  The UserEmail is the unique id, which is the PK in the auth database.

4.3.3 The bucket

Images are store in jpg format with a medium compression for maximum storage capacity without losing notable quality(Figure 4.15). Inside the bucket, images are public objects, and they can be accessed for anyone who has the URL. Folders organize objects (Figure 4.16), they are numbered, and more folders and buckets can be added. The URL is generated randomly by the main service, which is composed of two integers for a fast generation of them.

4.4 Post Service

Jose I. Retamal Jose I. Retamal

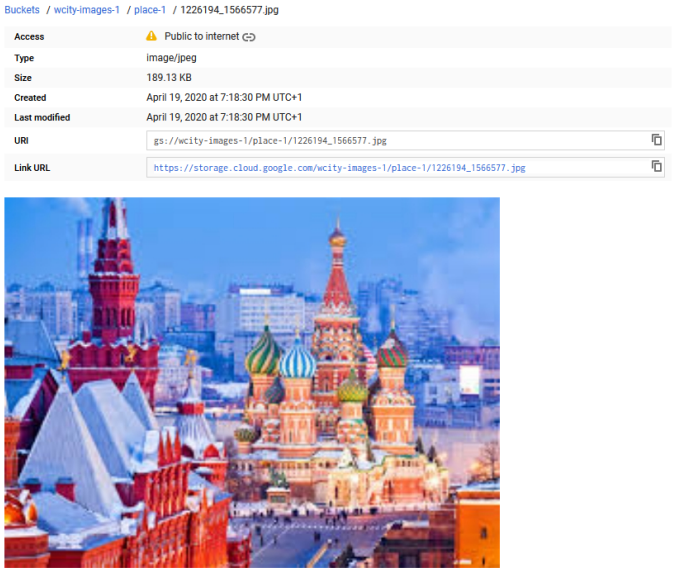

Figure 4.15: Photo Service- Image in Bucker.

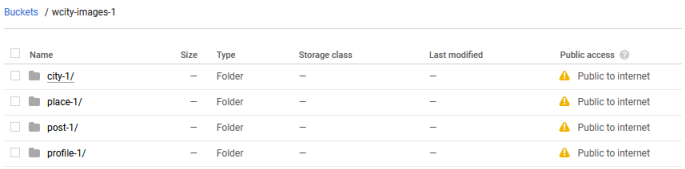

Figure 4.16: Photo Service-Folders Structur in a Bucket.

Post Service manage posts, there are posts for cities and places. The service is composed of two paint parts: The main service and the database(Fugure 4.17).

The main service connects to the client and provides the main endpoints for creating view and update posts. It connects to the Mongo database and checks requests on the authentication service.

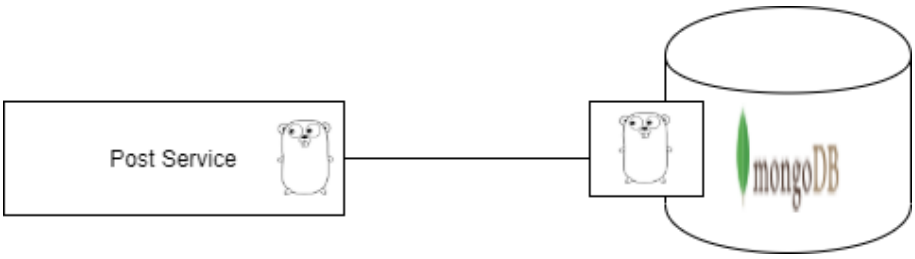

Figure 4.17: Post Service- UML.

4.4.1 Request Sequence

- Create Post

  When a user creates a post, the request contains the post data and the authentication token. The token is check in the authentication service, and then the post is stored in the database.

4.4.2 The database

Post are grouped by place and cities using the Neo4j unique id for each city and place. The database is composed of two tables, one for cities and another for places(Figure 4.18). All data is stored in one collection. The data is indexed by the unique id (index id) therefore, the performance is not affected by the amount of data stored.

As many pages as needed.

- Architecture, UML etc. An overview of the different components of the system. Diagrams etc... Screen shots etc.

Column 1	Column 2
Rows 2.1	Row 2.2

Table 4.1: A table.

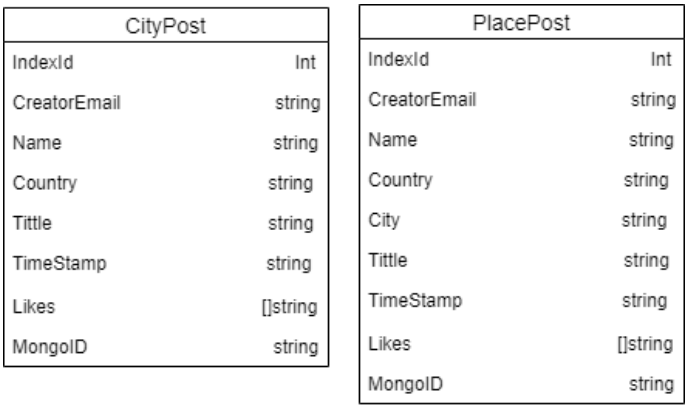

Figure 4.18: Post Service- Database Entity Diagram.

Chapter 5

System Evaluation

As many pages as needed.

- Prove that your software is robust. How? Testing etc.
- Use performance benchmarks (space and time) if algorithmic.
- Measure the outcomes / outputs of your system / software against the objectives from the Introduction.
- Highlight any limitations or opportuni-ties in your approach or technologies used.

Chapter 6

Conclusion

About three pages.

- Briefly summarise your context and ob-jectives (a few lines).
- Highlight your findings from the evalua-tion section / chapter and any opportuni-ties identified.

Bibliography

[1] A. S. Tanenbaum, *Distributed Systems*. 2017.

[2] L. Beatty, "The pros and cons of mysql." https://www.smartfile.com/blog/the-pros-and-cons-of-mysql/. [Online; accessed 12-Novemer-2019].

[3] S. Dhruv, "Pros and cons of using postgresql for application development." https://www.aalpha.net/blog/pros-and-cons-of-using-postgresql-for-application-development/. [Online; accessed 12-Novemer-2019].

[4] M. Wu, "What are the major advantages of using a graph database?." https://dzone.com/articles/what-are-the-pros-and-cons-of-using-a-graph-databa. [Online; accessed 29-Novemer-2019].

[5] solid IT, "System properties comparison datastax enterprise vs. neo4j vs. tigergraph." https://db-engines.com/en/system/Datastax+Enterprise%3BNeo4j%3BTigerGraph. [Online; accessed 29-Novemer-2019].

Appendices

Appendix A

Docker

Jose I. Retamal

A.1 Install Docker in Ubuntu Using Command Line

A.1.1 Setup repository

- Update packages

```
$ sudo apt-get update
```

- Install packages to allow apt to use a repository over HTTPS:

```
$ sudo apt-get install \
  apt-transport-https \
  ca-certificates \
  curl \
  gnupg-agent \
  software-properties-common
```

- Add Docker's official GPG key :

```
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg
↪  | sudo apt-key add -software-properties-common
```

- Verify that you now have the key with the fingerprint 9DC8 5822 9FC7 DD38 854A E2D8 8D81 803C 0EBF CD88, by searching for the last 8 characters of the fingerprint.

```
$ sudo apt-key fingerprint 0EBFCD88
```

- set up the stable repository:

```
$ sudo add-apt-repository \
  "deb [arch=amd64]
  ↪ https://download.docker.com/linux/ubuntu \
  $(lsb_release -cs) \
  stable"
```

A.1.2 Install Docker Community

- Install the latest version of Docker Engine - Community and container, or go to the next step to install a specific version:

```
  $ sudo apt-get install docker-ce docker-ce-cli
  ↪ containerd.io
```

- Verify that Docker Engine - Community is installed correctly by running the hello-world image.

```
  $ sudo docker run hello-world
```

A.2 Run Image Using Docker Hub

- Create repository in Docker Hub.

  https://docs.docker.com/docker-hub/repos/

- Build Image (Local machine):

```
$ sudo docker image build -t
↪ docker-hub-user-name/image-name:version-tag .
```

  Example:

```
$ sudo docker image build -t joseretamal/hash-service:1.0
↪ .
```

- Push Image (Local machine):

```
$ sudo docker push
↪ docker-hub-user-name/image-name:version-tag
```

  Example:

```
$ sudo docker push  joseretamal/hash-service:1.0
```

- Pull Image (Remote machine):

```
$ sudo docker pull
↪ docker-hub-user-name/image-name:version-tag
```

Example:

```
$ sudo docker pull joseretamal/hash-service:1.0
```

- Run image (Remote machine):

  Opening a port and restart on crash or reboot:

```
$ sudo docker run -d -p internal-port:open-port --restart
↪ always --name instance-name
↪ user-name/image-name:version-tag
```

Example:

```
$ sudo docker run -d -p 5151:5151 --restart always --name
↪ hash-service joseretamal/hash-service:1.0
```

  Allowing instance to full network acess (allows acess to local host):

```
$ sudo docker run -d -p  --network="host" --restart
↪ always --name instance-name
↪ user-name/image-name:version-tag
```

Example:

```
$ sudo docker run -d -p  --network="host" --restart
↪ always --name hash-service
↪ joseretamal/hash-service:1.0
```

- Stop instance: (Remote machine):

```
$ sudo docker rm --force instance-name
```

Example:

```
$ sudo docker rm --force hash-service
```

- Check logs: (Remote machine):

```
$ sudo docker logs instance-name
```

Example:

```
$ sudo docker logs hash-service
```

- Bash into the container: (Remote machine):

```
$ sudo docker exec -it instance-name bash
```

Example:

```
$ sudo docker exec -it hash-service bash
```

Appendix B

MySql

Jose I. Retamal

B.1 Install Mysql in Linux Using Command Line

B.1.1 Install MySQL-shell

- Make sure you do not skip the step for updating package information for the MySQL APT repository:

```
$ sudo apt-get update
```

- Install MySQL Shell with this command:

```
$ sudo apt-get install mysql-shell
```

B.1.2 Install MySql server

```
$ sudo apt-get install mysql-server
```

B.1.3 Uninstall MySql server

```
$ sudo apt-get remove --purge mysql*
$ sudo apt-get purge mysql*
$ sudo apt-get autoremove
$ sudo apt-get autoclean
$  sudo apt-get remove dbconfig-mysql
$ sudo apt-get dist-upgrade
```

B.1.4 Setup Replication

https://www.digitalocean.com/community/tutorials/how-to-set-up-master-slave-replication-in-mysql

setup master

- Edit the mysql config file,for open the file using vi:

```
$sudo vi /etc/mysql/mysql.conf.d/mysqld.cnf
```

- make the followings changes to the the file, if the field are missing they must be added or if they are commented un commented:

```
server-id       = 1
log_bin                 = /var/log/mysql/mysql-bin.log
binlog_do_db  = replica1
sudo mysql_secure_installation
```

- Restart MySQL:

```
$ sudo service mysql restart
```

- Create user for replication and give permissions:

```
$ sudo mysql -u root
mysql>GRANT REPLICATION SLAVE ON *.* TO
↪  'slave_user'@'%' IDENTIFIED BY 'password';
FLUSH PRIVILEGES;
```

- Get master status, after select the database in one MySQL seasiion :

```
 mysql>FLUSH TABLES WITH READ LOCK;
```

- then open another MySQL seasion(keep the other open):
- Get master status, after select the database in one MySQL seasiion :

```
mysql>SHOW MASTER STATUS;
| File             | Position | Binlog_Do_DB |
+------------------+----------+--------------+--
| mysql-bin.0001580 |      154 | user_login   |
+------------------+----------+--------------+----
```

Note the file (mysql-bin-0001580) and the position.

- After take note of file name and position tables can be unlocked :

```
mysql>UNLOCK TABLES;
```

setup slave

- Edit slave config file :

```
$ sudo vi /etc/mysql/my.cnf
```

Make the following modifications:

```
server-id               = 2
relay-log               =
↪  /var/log/mysql/mysql-relay-bin.log
log_bin                 = /var/log/mysql/mysql-bin.log
binlog_do_db            = newdatabase
```

- Restart MySQL service :

```
$ sudo service mysql restart

```

- Config slave in mysql shell:

```
mysql> CHANGE MASTER TO
MASTER_HOST='104.40.206.141',
MASTER_USER='repl',
MASTER_PASSWORD='password',
MASTER_LOG_FILE='mysql-bin.000160',
MASTER_LOG_POS= 2439;
```

- Start slave

```
mysql> START SLAVE;
```

- Check status

```
mysql> SHOW SLAVE STATUS\G
```

Appendix C

Neo4J

Jose I. Retamal

C.1 Neo4j With Docker

C.1.1 Install

- Pull the latest image from docker hub(`https://hub.docker.com/_/neo4j/`):
- Run Neo4j:

```
$ sudo docker run \
--name neo4j \
-p7474:7474 -p7687:7687 \
-d \
-v $HOME/neo4j/data:/data \
-v $HOME/neo4j/logs:/logs \
-v $HOME/neo4j/import:/var/lib/neo4j/import \
-v $HOME/neo4j/plugins:/plugins \
--env NEO4J_AUTH=neo4j/test \
neo4j:latest
```

C.1.2 Access bash console:

- Access image bash:

```
$ sudo docker exec -it neo4j bash
```

- Access neo4j bash:

```
1 $ cypher-shell -u neo4j -p test
```

Appendix D

Google Cloud Storage

Jose I. Retamal

D.1 Upload Images To Bucket

https://cloud.google.com/storage/docs/reference/libraries#command-line

- Create the Service Account account, [NAME] is thenew name:

```
$ gcloud iam service-accounts create [NAME]
```

- Grant permissions:

```
$ gcloud projects add-iam-policy-binding [PROJECT_ID]
↪ --member
↪ "serviceAccount:[NAME]@[PROJECT_ID].iam.gserviceaccount.com"
↪ --role "roles/owner"
```

- Generate the key file, [FILENAME] is the name of the new file:

```
$ gcloud iam service-accounts keys create
↪ [FILE_NAME].json --iam-account
↪ [NAME]@[PROJECT_ID].iam.gserviceaccount.com
```

- Provides authentication to the application by setting credentias in the paht(Linux):

```
$ sudo export GOOGLE_APPLICATION_CREDENTIALS="[PATH]"
```

- Set path variablle in docker image file:

```
$ ENV GOOGLE_APPLICATION_CREDENTIALS="[PATH]"
```

- Go code to upload a file :

```
$ // Sample storage-quickstart creates a Google Cloud Storage bucket.
package main

import (
"context"
"fmt"
"log"
"time"

"cloud.google.com/go/storage"
)

func main() {
  ctx := context.Background()

  // Sets your Google Cloud Platform project ID.
  projectID := "YOUR_PROJECT_ID"

  // Creates a client.
  client, err := storage.NewClient(ctx)
  if err != nil {
    log.Fatalf("Failed to create client: %v", err)
  }

  // Sets the name for the new bucket.
  bucketName := "my-new-bucket"

  // Creates a Bucket instance.
  bucket := client.Bucket(bucketName)

  // Creates the new bucket.
  ctx, cancel := context.WithTimeout(ctx, time.Second*10)
  defer cancel()
  if err := bucket.Create(ctx, projectID, nil); err != nil {
    log.Fatalf("Failed to create bucket: %v", err)
  }

  fmt.Printf("Bucket %v created.\n", bucketName)
}
```

40

Appendix E

Goland

E.1 Create and Publish Modules

- Create module

```
$ go mod init <module-path>
```

- Install or update dependencies

```
$ go get
```

- Remove unused dependencies

```
$ go mod tidy
```

- Publish to GitHub

```
$ git add .
$ git commit -m "the commit"
$ git tag v1.0.0
$ git push origin v1.0.0
```

E.2 Install Go in Linux

```
$ sudo snap install go --classic
```

E.3 Go Docker Image

```
FROM golang:latest
COPY . /usr/src/myapp
WORKDIR /usr/src/myapp
```

```
4     EXPOSE <PORT-NUMBER>
5     ENTRYPOINT ["./<EXECUTABLE>", "<COMMAND-LINE-PARAMETERS>"]
```

E.4 Protocol Buffer

- Install

```
1     $ sudo snap install protobuf --clasic
```

- Compile using GRPC plugin

```
1     $ protoc --go_out=plugins=grpc:<OUT-FOLDER>
      ↪  <FILE-NAME(.proto)>
```