INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Development of a Front-End solution for the Gig Economy in the beauty & body-art sectors

PEDRO MIGUEL RODRIGUES GOMES
Outubro de 2020

POLITÉCNICO
DO PORTO

Development of a Front-End solution for the Gig Economy in the beauty & body-art sectors.

Pedro Gomes — 11030383@isep.ipp.pt
Advisor — Filipe de Faria Pacheco — FFP@isep.ipp.pt

2019-2020

A Project submitted for the purpose of completing the Master's degree in Graphical Systems from Instituto Superior de Engenharia do Porto.

Dedicatory

A special thanks to my family for being an exceptional role model and always capable of providing guidance through every chapter of my life; to professor Filipe de Faria Pacheco for his dedication and availability and to my friends and colleagues.

Summary

Opportunity

The Gig Economy is revolutionizing the way people work and **on-demand** platform-work is becoming more common, growing its economy and expanding into more and new industries. Online platforms promoting **on-demand** either remotely or **on-location** work as a form of employment are increasing by the day as workers don't want to be tied to a lengthy career, job titles or typical 9 to 5 work schedules. While platforms such as Uber and Airbnb dominate the market on transportation and home-sharing respectively, the on-demand, on-location market for short-lived tasks has untapped potential.

Social media business profiles are emerging at an accelerated pace, from fully developed businesses boosting their presence in the current consumer market, to self-employed people promoting themselves online to attract more clients and make extra income. Many recognize this generation of buyers as one deeply reliant on social-media to decide what to buy next, let it be services or products. Social-media profiles for cosmetic jobs ranging from Barber, Hairdressing, Manicure, Pedicure, Make-up to more artistic such Tattoo Artist and Body Piercing are increasing; however, social-media platforms, while providing the best mean for content and media delivery, they lack functionalities to allow users to properly arrange meetings and handle appointments.

Solution

GigWho is a mobile platform that connects independent platform workers in the beauty and body art sector with clients in a swift and consistence experience. It's an all-in-one mobile solution for convenient services such as: Barber, Hairdresser, Pedicure, Manicure, Make-Up, Tattoos and Body Piercing. The clients are able to request a service whenever they want, according to workers' availability, and workers can add a new source of revenue with full flexibility. **GigWho** is aimed for self-employed workers granting them the ability to choose when and how they want to work, with specific Platform-Work features (schedule appointments, client management)

as well as typical social media features (media content display, likes and shares, push-notifications). As for the clients, **GigWho** provides them convenience and flexibly to choose their favourite worker and handle everything through the platform.

Keywords: Gig Economy, platform-work, on-demand, on-location, social-media, beauty and body-art.

Sumário (PT)

A forma como a civilização desempenha funções está a mudar, tal como a forma de como as pessoas esperam que o mercado responda aos seus pedidos. Nos dias que correm, ter um trabalho não implica um ter horário fixo, um título, ou até um contrato. Numa sociedade que raramente para, um horário flexível não implica a perda de oportunidades; e, independentemente das consequências que resultam da carência de um contrato, existe quem beneficie das condições que dessa situação advêm.

As empresas tecnologicas começaram a tirar proveito da tecnologia e da incansável necessidade da sociedade, para providenciar vários serviços "on-demand", fisicamente ou online, e a acordar com trabalhadores independentes para dispor das suas capacidades através da plataforma. Plataformas como a Uber, Lyft ou Upwork juntam o produto e a procura de uma forma direta e transversal aos modelos anteriores.

Nos dias de correm, a presença online das empresas é cada vez maior, mesmo quando fora da sector do 'e-commerce'. Para além de websites, são criados perfis nas várias redes sociais (Facebook, Twitter, Instagram) que as pessoas, e os seus clientes, usam como principal instrumento social e comunicação. As redes sociais também atuam como meio de promoção para estas empresas e para trabalhadores independentes individuais. Estas permitem a partilha de conteúdo de uma forma clara e direta, que antes era mais exigente e dispendiosa.

A **GigWho** vem explorar este novo estilo de autopromoção online, através das redes sociais, juntamente com a nova tendência de trabalho independente. É uma plataforma que junta os trabalhadores aos clientes em áreas relacionadas com a estética e arte corporal, tais como: Barbeiro, Cabeleireiro, Pedicure, Manicure, Maquilhagem, Tatuador e Body Piecer.

Contents

List of Figures

List of Tables

Listings

1 Introduction

1.1 Context

The term "Gig Economy" refers to a general workforce environment in which short-term engagements, temporary contracts, and independent contracting is commonplace. It's also referred to as the "freelancer economy" "agile workforce", "sharing economy", or "independent workforce".

...

The freelancer economy (or freelance economy) differs from traditional employment in that jobs are not permanent, but more specifically, the term relates to many one-off tasks or individual shift assignments. However, the term may also be used to reference longer-term freelance arrangements and independent contracting assignments. — [Stringfellow, 2019].

Platform-work is a new work methodology outside the 9 to 5 work standards and what is propelling the Gig Economy [Intuit, 2017]. It is a new form of employment that uses online platforms to connect supply with demand, through paid services. These platforms are typically online matchmakers and/or tech frameworks that do not own the means for production, but instead create the means for connection [Moazed and L. Johnson, 2016, p. 30].

Platform workers usually have a formal agreement with the platform, such as Uber and Airbnb, to provide services to their clients. More often, companies tend to hire independent contractors and freelancers instead of full-time employees. For the workers it might imply increased freedom as they can shift careers more often and work on multiple projects for as many clients in or out of their area of expertise. They do not have to be bound to a single company for long periods of time and can have a flexible work schedule on temporary positions. All these factors can lead to a superior service provision and more competitive markets. However, these perks usually contrast with the lack of common employee benefits, such as health insurance, paid leave, paid holidays and compensations. This is because platform workers are seen as freelancers and not actual employees for the

company they work for. Platforms usually acquire their workforce online and by fulfilling the requirements and following their standards, one can start working immediately.

Many platforms exist to fill a specific market segment, for transportation based services there is Uber and Lyft. Both provide work for those willing to and both fill a high demand for quality ride-hailing services at a lower price. One of the main reasons Uber got to be so successful was the cost per travel vs its competitors (with Taxis being the main, if not the only one) [Johnson, 2020]. They explored a not well regulated market with the use of technology, and pushed the Gig Economy far beyond where it would be today. There are other platforms that took advantage of the state of technology and existing gaps other services couldn't either identify nor fulfil. Upwork is a great example of a work platform that adopted the flexible work schedule, focused on remote services, and heavily capitalized on peoples knowledge, allowing them to perform on demand services for wherever part in the world. Usually, these platforms make great use of mobile devices to execute their model. Some, such as Uber and Lyft, completely rely on the portability given through our small devices while others use them to complement their web platform, e.g. UpWork.

Platform-work is also impacting the beauty industry to a great extent. Providing beauty services anywhere, on-demand, has opened new opportunities for the industry and has an increasing market presence ([Sankhla, 2018]).

Social-media businesses profiles are a common trend as business are not only becoming more susceptible to innovation but also have further interest in promoting themselves online, where their clients spend much of their time and depend on social media to decide what to buy or to do [Kats, 2019]. On an individual level, social media allows workers to showcase their work as portfolio, bring in more clients and earn more.

1.1.1 The Origin of GigWho

GigWho started with a group of three friends with an ambition to make the world more available and responsive through an application for the Gig

Economy. There are already many applications that act as a mediator for the acquisition and provision of services. These applications create immediate, continuous and economical supply chains straight to the final consumer, disrupting previous chains and methodologies. The Internet, a utility already profoundly ingrained in our society, and the capable hardware interfacing with it were major factors that allowed new chains of service to exist.

In 2015, Tom Goodwin wrote:

Uber, the world's largest taxi company, owns no vehicles. Facebook, the world's most popular media owner, creates no content. Alibaba, the most valuable retailer, has no inventory. And Airbnb, the world's largest accommodation provider, owns no real estate. Something interesting is happening.

...

This sentence from Tom Goodwin ignited our motivation to build something different, to join all short-lived services/tasks in one single application. Services that ranged from Barber, Baby Sitting, Chef, Catering, Personal Trainer to Plumber, Electrician and much more, all to be performed **on-location**. There was never a clear view of the services to be supported, but we always had one major priority: The process should be simple enough, like nothing we've ever seen before. The user chooses which of the service he requires and, in a few minutes, a worker would be knocking at his door.

We first started the development of a mock with cryptocurrency support where the user would pay in cryptocurrency. Soon, the idea was dropped as the use of cryptocurrency is awfully complicated for the average user to understand, especially when compared to the current payment methods available. The process of implementing it in the current model was also rather difficult, and so we started to consider more traditional payments mechanisms. From that, the team debated if supporting any number of services would be feasible since there would be no control of what is being done through the platform. We decided to only support a specific set of more easily managed categories. However, because **GigWho** was purely focused on the **on-demand**, **on-location** supply, we realized it would require a

massive infrastructure as well thought logistics to move forward with the concept, and decided to reach out to a handful of VC entities for support and financial investment. Unfortunately, all answers turned out negative.

Right now, **GigWho** solely exists within the context of this thesis and has completely changed from its initial concept. The author, Pedro Gomes, is the only contributor.

1.2 Problem

There isn't a viable solution that joins Platform-Work specific features with the ones provided by social media platforms. Even though social media is currently acting as promotion channel, they do not provide the right tools to handle customers, appointments or payments. The current use case resolves around using direct messages in order to do all the above. An example would be: A client using Instagram searches for haircuts or nails and is immediately presented with page suggestions from pedicure/manicure and hairdresser/barber workers. The user, if pleased, will direct message the worker asking about his availability and service costs. The communication might not be short, between aligning a date, asking for the price and location, and not answering immediately, it might take more time than it should. Not just that but there's also no useful way to store all the exchanged information afterwards. Both worker and client might have to write it down somewhere, memorize it or open the conversation multiple times to revise the details.

1.3 Goals

The proposed solution joins characteristics from Platform-work platforms such as:

1. In-app payment.
2. On-demand.
3. On-location (per worker availability).
4. GPS features to aid both clients and workers.
5. Clients Appointment Management w/ Status Push Notifications.

With characteristics typical for a social media platform:

1. Push-Notifications
2. Content media display.
3. Follow worker profile pages.
4. Like content.
5. Suggestive content.
6. The ability to share content to outside the platform.

GigWho is a beauty and body-art platform-work platform, mainly focused on self-employed workers who might already use social media as promoting channel, wish to boost their productivity and add new sources of income while working when, where and how they want. Workers are free to sign in and start building their profile. Clients are equally free to sign-in and start navigating the platform, finding pages based on their location and the type of services they want to. They can also schedule appointments, interact with the content and follow their favourite workers. The app also contemplates push notifications for certain status updates about the appointment and store events.

The reason behind categories on beauty and body-art

The reason why **GigWho** considers these categories (Barber, Hairdresser, Manicure, Pedicure, Make-Up, Tattoo Artists and Body Piercing) is due to its compatible nature with platform work and social media platforms. These tasks are usually short-lived, done in one session (might be slightly more based on the dimension of the project), momentary (e.g.: the client decision process when deciding if he needs a haircut is usually quick), paid in the act, and do not require a professional relation nor contract between the client and the worker. These jobs are perfect to be provided through an online platform, where the client can easily access a wide range of workers. Furthermore, they are also perfectly suitable for a social-media platform, where sharing content with friends, family and strangers is a central goal. A picture is worth a thousand words and workers are able to build their portfolio online using

nothing but pictures of work they have done. In conclusion, these jobs are a perfect match between work and social media.

1.4 Approach

A succinct approach to this project's problem can be decomposed into the following phases:

1. Search for information regarding Platform-work. Evaluate if it is a viable subject to be explored. Identify current gaps on existing workflows that require a solution and validate the viability of the GigWho platform as a solution capable of mitigating those gaps.
2. Elaborate a Value Analysis of the proposed solution. How can it bring value to the current work model, answer if it can facilitate the process and if it allows all participants to have more control over their workflow. The Value Analysis also contemplates a discussion about the most adequate framework adequate to build the platform.
3. Consider which features would make sense for the solution at hand and from those apply technical processes such as the Quality Evaluation Framework (QEF). As a quality assessment tool this framework allows to monitor all deliverables and assert the overall quality of the application.
4. Analyse, design and implement the features from the previous phase.
5. Evaluate and test.

1.5 Document Structure

The structure is organized as the following:

- **Introduction**: Provides a brief context about the main subject of this thesis as well as the relevant problem. It also displays a list of goals proposed to be achieved alongside the approach taken to accomplish them. The approach refers to a list of stages sorted by the order in which they must occur.

- **State of the Art**: Adds current and previous knowledge of platforms related with Platform-Work to establish them as business and as products. Explores the elements that contributed to the evolution of working online using these. It also reveals some of the most used cross-platform frameworks for mobile development and addresses their position in the current development ecosystem.
- **Value Analysis**: This section explains how GigWho can generate value for all parties involved (workers and clients) and how that value is perceived. It also applies a comparison model to proper evaluate all the considered frameworks to develop GigWho.
- **Evaluation Planning**: This section refers to the evaluation model used to assess the development of GigWho.
- **Design**: Explains the process of designing the GigWho platform and all its integrated external components.
- **Implementation**: Section with actual implementation snippets and comprehensive explanations about how GigWho works.
- **Evaluation**: This section is dedicated to the verification of the developed features as well as outright evaluation encompassing developer observations.
- **Conclusion**: Final thoughts about what was done with a sum up of the primary goal and personal comments.

2 State of the Art

2.1 Solutions for Platform-work

Nowadays Platform-work as a form on employment is a trending subject. There are many platforms available that aid and support this work paradigm and are able to generate high gross volume while employing a considerable amount of people. According to a study done by MasterCard — The Global Gig Economy: Capitalizing on a $500B Opportunity [Matercard and Associates, 2019] — the Gig economy generated $204B in Gross Volume in 2018, with Transportation-Based services comprising 58% of it. They also expect a growth of more than double, to $455B, by the year of 2023. Platforms that adopt Platform-work as their main business model are becoming more common. Uber, TaskRabbit, Airbnb, Handy, Upwork, Fiver are huge successful companies that rely on external work force and provide them appropriate software infrastructure to operate. These platforms are pushing new work standards without enforcing convoluted contracts and delivering flexibility with the possibility of higher income. On the other hand, they lack security, growth opportunity and work regulations.

Technology is the key factor that allowed the growth of such platforms as they totally rely on internet access and, some of them, on smartphones to properly operate. It is not possible to conceive the idea of platform-work models without internet access. They require high accessibility and some (such as Uber) couldn't even operate without smartphones constantly connected.

The number of smartphones and worldwide internet users is rising ([Clement, 2019], [Milijic, 2019]) and directly impacting the number of eligible self-employers. Most of these platforms first launched their services on regions (North-America and EU) where the internet access and smartphone users were the highest when compared to the rest of the world ([Clement, 2019]). Both elements are disrupting the way people connect, making it instant, and platforms are smartly taking advantage of it. Cultural change is also a factor playing a major role here. People became used to fulfil their needs on demand and workers are prone to prioritize a flexible work schedule

instead of the typical 9 to 5. Finally, the lack of quality alternatives prior to the surge of these platforms also impacted their impact of the markets. Uber quality service with better cars, more pleasant drivers, price of the trip disclosed beforehand, at a lower price instantly surpassed the one provided by local taxis. Today, in the 70 countries Uber claims it operates, their price is lower than local the alternative on 66 of them([Johnson, 2020]).

Nowadays platform work has a wide variance and target completely different markets. Uber for transportation, Airbnb for house rental and accommodations, Upwork for more technical demanding jobs (even qualified) from software developer, admin support, data science to engineering and architecture. It's the ideal platform for someone who has the required skill set and wants to freelance remotely. The worker sets a price for their services and is hired based on their skills, or applies to open jobs from clients on the platform. Upwork worker profile also acts as portfolio, to showcase his experience and clients can directly contact them based on what they think the best match is. The communication can be initiated by either one.

AirTasker is a platform dedicated to household tasks only available in Australia. Home cleaning, furniture assembly, handyman and gardening are a few services supported by the platform. It follows an auction based system to connect their users. Clients post a detailed task they want completed with a price they think is fair and workers bid on it with the price they expect from it. The client assigns the worker and both communicate to decide how the job is going to proceed.

TaskRabbit, operating in US, is another platform targeting the same market but without the bidding model. They have a suggested price for the task at hand and match workers based on the client requisites (when, where, how long and details).

ListMinut is also for the same market but extends its services to baby-sitting, pet-sitting, package transportation and much more. They operate in France and Netherlands.

One platform operating locally, in Portugal, is Zaask, who also targets multiple services from the following sectors: Home, Health and Well-being,

Events, Tutoring, Weddings and Business (from business consultants, to social media marketing managers). They categorize themselves as the Platform for qualified gigs. They accept work requests from their clients and dispatch them to their workers. After analysing, the worker either accepts, cancels or counter-propose. Launched in 2012, they are now partnering with IKEA for their *Furniture Mounting* and *Kitchen Installing* services.

As for beauty there are: The Glam App in US and London, Soothe in the US, Canada, Australia and UK and the Beauty Now, operating in Portugal. The Beauty Now is a platform-work for beauty services performed on-location only. These services range from: Pedicure, Manicure, Make-up, Hairdresser, Massages... It allows clients to call for services on-demand or schedule for whenever they want to. The worker is the one meeting the client at their desired location. The major difference is that GigWho is more worker-oriented, letting the worker choose where he wants to work, either from his place or on-location, meeting the client. GigWho does not match-make workers and clients on-demand but instead provides a way for the client to find the most adequate worker and schedule an appointment (for now or later).

A social-media business software tool worth mentioning is Facebook for Business. A platform from Facebook to establish business within their network. It can enhance businesses by providing a vast set of tools to manage advertisement, handle employees, products, metrics, payments, appointments and client engagement. A complete solution for small to large scale businesses within the Facebook ecosystem.

2.2 Cross-Platform Mobile Frameworks

The **GigWho** application is a mobile app that targets two of the most used operative systems: iOS and Android ([mob]). As such, information was gathered about state-of-the-art cross-platform frameworks, for mobile development, that are available, and which one is preferred for the considered use case.

Let's delve into the selected four and explore some of its strengths and weaknesses.

2.2.1 Ionic

An open-source UI toolkit for building performant, high quality mobile and desktop applications using web technologies (HTML, CSS and JavaScript). Ionic (and Capacitor) are maintained by the Ionic team and, together, have about 45k stars on GitHub. It can either be used standalone, with JavaScript/TypeScript and their stylish UI component library, or with front-end framework giants such Angular, React and Vue, providing their respectively powerful front-end capabilities. It's a framework-less component library where developers can choose whatever flavour available they want.

The native API to features such as GPS, File System, Camera and much more, is granted using Cordova (formerly known as PhoneGap) or using its new application run-time Capacitor. Both act as bridge to access native device functionalities.

With Ionic, you build what is called hybrid applications, or Progressive Web Apps, since they never run natively on your device, but in a browser, while being packaged as native applications for distribution in the apps stores (App Store for iOS, Play Store for Android). Because Ionic does not build true native applications, performance will not always be granted. The app runs inside a native API called WebView (for Android) or UIWebView (for iOS) and their job is to wrap progressive web apps on mobile devices. Serving mobile applications like this introduces additional overhead that could drastically impact the app performance. Not only that but both Cordova and Capacitor, the bridges that grant native access, might not cover

100% of it, leading to some limitations in the future. They do allow, however, to create and inject custom plugins.

The unfortunate thing about Ionic also plays in its favour, because they are progressive web apps, most, if not all source code works on iOS, Android and on the Web without having to tweak much between and both Capacitor and Cordova provide API fall-back based on the platform they're running on. With Ionic the code is totally reusable between platform, you write once and run everywhere. Overall, it provides clean UI components, uses modern web technologies, produces hybrid apps that can run on any device and can be integrated with front-end frameworks with considerable community and documentation. Where Ionic lacks is performance.

2.2.2 React Native

Is a cross-platform framework for mobile backed by Facebook, has around 85k stars and is the second most contributed open-source project on GitHub. Like Ionic, it uses current web technologies (HTML, CSS and JavaScript) making it easy for current web developers to get into in no time.

For React, documentation and community is not a problem and application's showcase is unmatched by any other framework with apps such as Facebook, Instagram (two of the most successful social media platforms), Discord, Pinterest, Uber (the most successful ride-sharing platform), Tesla and much more. The fact it is structuring many of the most used mobile applications nowadays proves that it is capable of fulfilling any use case (without considering 3D/Graphics rendering content, such as games).

On the performance side, React compiles its components into their native representatives and keeps a separate JavaScript thread responsible to handle business logic, dispatch commands to the native UI as well to the run-time bridge (React Native Bridge) to access device functionality. This architecture guarantees the UI thread is left untouched and is only responsible for rendering the UI. React also applies a similar concept to the *Virtual DOM* to dispatch UI changes in batches and only when actually required, to guarantee extreme responsiveness. Although fast, this system might not provide the best performance vs other frameworks that completely

compile to native (see Flutter — 2.2.4). However, such statements can't be made lightly as it depends on the device's hardware.

React Native has a substantial number of third-party modules already developed by the community to access native functionalities.

2.2.3 NativeScript

Cross-platform framework for Android and iOS devices. It's similar to React Native as it also only compiles its UI components into their native equivalents, it's similar to Ionic as it can be used with top-tier front-end frameworks (Angular, React, Vue and Svelte) and similar to both as it also uses current web technologies (HTML, CSS and JavaScript).

NativeScript injects all platform-specific APIs into its JavaScript run-time (V8 for Android and iOS[1]) running in the main UI thread, hence one can expect 100% access to the native APIs, out-of-the-box. The run-time act as a bridge for native functionality. However, its architecture heavily relies on a single main thread and performance might not be as good as React Native or Flutter.

NativeScript lacks in its application showcase and community size. It's not near as big as React Native nor growing as fast as Flutter but still considered a viable alternative, with 18k stars on GitHub and 160 contributors.

2.2.4 Flutter

Is the Google's UI toolkit for building beautiful, natively compiled applications for mobile, web and desktop from a single code base. Flutter immediately stands out from the rest of the Frameworks for two major factors, the first: code-base is compiled to native code in contrast to the other mentioned frameworks; and second: it is backed by non-other than Google. It does not use current web technologies but Dart as programming language. Dart is currently gaining some popularity because of Flutter himself, but until this point there were not strong reasons to consider using Dart for whatever

[1] iOS — Made possible due to the JIT-less V8 mode. JavaScript can be executed without allocating executable memory at run-time.

use case a developer might come across with. In fact, in 2018, Dart was considered to be the worst language to learn, by StackShare.

These frameworks are further discussed in the Value Analysis section where it is established a model to proper evaluate them against one another to decide which will be used to develop the **GigWho** app.

3 Value Analysis

It is important to understand the value **GigWho** and similar platforms bring to the current labour community, its status and to the overall market. How value is perceived as participants (either from the workers perspective or clients) versus the actual value attained is the subject of analysis in this section. This research helps understand the potential of implementing Platform-Work models and how they build a more responsive and interactive world. It is also important to perceive the value GigWho brings versus other platforms that provide similar services. Finally, it analyses the technology used to build GigWho, explaining the considered alternatives and the criteria that led to the final decision.

3.1 Value Analysis Objectives

Value Analysis is about contextualizing how a process is currently being done (how goals are achieved) and how the solution subject of analysis can improve or enhance a process pipeline. A value analysis allows to clearly demonstrate advantages a solution brings to the current model, as well as disadvantages, if any; It should answer if the positive factors outweigh the negative to conclude if the solution proposed is a valuable one.

The following is an analysis on how self-employed workers can benefit from using platforms such as **GigWho**. To do that it's recommended to answer the following questions:

1. What is your project?
2. Who is your target costumer? Or who you provide the value?
3. What value does the solution provide?
4. Why is the product unique?

The market of digital labour is growing rapidly, partially because of the rise of platforms such as Uber. These type of platforms can either attain value by creating services (or products) and sell them to the consumer or by connecting service providers (or product suppliers) with their customers.

The latter is the one adopted by **GigWho** and concerns three key entities, the Platform, the Worker and the Customer (Client). The value generated is directly tied to how many connections it makes and how well does it make. More connections translates to more value. However, whatever the model adopted by any of the platforms, for the worker it only matters if the value generated can, ultimately, reflect on more profitability. Even though value should generate revenue from the connection between the worker and his clients, GigWho provides them value in the form of:

- Consistent and responsive platform.
- Effortless process for clients for find their favourite worker.
- Client management system to conveniently manage clients.
- Provide a platform that allows media content to be displayed.

In general, the number of features and what they enable should be considered according to the targetted market. If it is self-employed drivers, the GPS tracking software should work flawlessly; on the other hand, if the focus is beauty/aesthetic services, the worker would be better off with a capable system to manage appointments, and have a profile gallery to publish his work to attract more clients and to clarify doubtful ones. The value of a platform is relative to how much a certain process gets simplified for everyone involved, in addition to the value it can add on top of the current model. It's possible to relate the value perceived by the worker by identifying how much they rely on the platform. According to the study written by Delloit — The Rise of Platform Economy [Delloit, 2018] — it is possible to categorize platform workers by archetypes:

- **Primarily Dependent**: The worker fully relies on the earnings of the platform.
- **Partially Dependent**: The worker uses the platform as a part-time job.
- **Supplemental**: The worker uses the platform to create supplemental earnings.

Based on these three tiers it's safe to assume the value perceived by the three groups is distinct, whereas one group clearly perceives more value if totally relying on the platform to generate revenue, others would perceive it less if the platform was only used for supplemental earnings. The primarily dependent group would be much more involved in the development of the app, whereas the others not so much.

3.1.1 What is the Project?

An all-in-one mobile platform that joins clients with self-employed workers. The project is a platform aimed for anyone ready to work within this specific set of categories: Barber, Hairdresser, Nails, Make-up, Tattoo Artist and Body Piercing. The focus is solely on the beauty and body-art work that is typically momentary, low-cost and can be self-promoted through the display of pictures. Many people are adhering to self-promotion on social-media because it has an excellent way to visually display content and can attract new customers that also use the same platform.

The main reason for addressing this sector is because they allow for it to be performed **On-Location**, **On-Demand** and multiple times a day, as they are usually **short-lived**.

3.1.2 Who is the target customer?

The target customer are workers who are currently working or planning to work independently within the supported categories. Specifically people who are comfortable displaying their work online to promote their services and are looking for an effortless way to handle their agenda and acquire more clients. Not only that, but also workers who do not own a physical space and prefer to work where the client requests.

3.1.3 What value does the solution provide?

GigWho provides workers the ability to choose when and how they want to work while granting them the opportunity to add new sources of income. The platform is aimed to be an all-in-on solution where they can manage their online store. Even though clients are not the main target, they also benefit

from the convenience and flexibility provided by the platform allowing them to request services when and where (per worker availability) they want. They are able to examine prior work to see if it is aligned with what they want and choose the worker that they think is the best fit.

3.1.4 Why is the product unique?

GigWho is unique in the way it joins two types of platforms (social media and platform work) in favour of the self-employed workforce while being completely free to use.

3.1.5 Lean Canvas

The Lean Canvas below provides a clear view of the **GigWho** platform and its underlying business model.

Problem:

- Lack of solution that joins Platform-Work specific features with Social Media ones.
- Beauty and Body Art workers are resorting to social media as promoting channel for their work and reach possible clients and don't have the right tool to manage them.

Solution:

- A platform with social-media and platform-work features.
- Clients can interact with content of a worker.
- Tool to manage clients (book, reschedule or cancel appointments, in-app payments, push notifications about the status of an appointment, content suggestion based on the client preferences).
- Tool to manage on-location and on-demand work.

Customer Segment:

- Self-employed workforce who wants to improve the way they operate.
- Anyone who wants to book, request on-demand and on-location (per worker availability) their favourite worker in the beauty and body-art sectors and/or enjoy watching related content.

Unique Value Prop

- Improving efficiency and profitability of the self-employed workforce having a centralized all-in-one solution.

Unfair Advantage

- Completely free to use.

Early Adopters

- People who already use social-media to reach their clients.

High Level Concepts

- Working platform for the new generation of workers.

Existing Alternatives

- Facebook for Business.

Channels

- Social-media solutions to promote.
- Search Engine Optimization.
- User references.

Key Metrics

- DAU/MAU.
- Most successful category.
- Most requested workers.

Cost

- Marketing.
- Web Infrastructure.
- Store presence.

Revenue

- Advertising Revenue.
- Outside investments.

Figure 1: GigWho Lean Canvas

3.2 Value as a Network

Value can be contextualized as a Network, a node connected representation of all stakeholders and their respective roles that formalize a chain of value. The network can be categorized as internal (when value is traded within a company, among all employees), or external (between a company, its suppliers and its customers) and value can either be tangible or intangible. Tangible value is something concrete, money, documents, while intangible is usually something not concrete, like knowledge, influence or even favours. Either way, value can only be perceived when converted into deliverable. From a stakeholder (nodes) perspective this happens when the input of one transaction, from another stakeholder, with a derivable, is converted into actual gains. [Allen, 2012], [Allen, 2008]

With **GigWho** value is external, meaning value is not traded from within the platform but between all participants of it: the workers, their clients (both users of the application) and the platform itself. Between the registered workers and clients, value is classified as tangible, they trade services for money respectively; however, what **GigWho** provides is intangible. To the workers it offers the tools for them to better manage their clients and promote themselves. To the clients, it offers a platform for them to find the best fit for the service they are looking for as well a platform to interact with content they enjoy. The following image provides a view on how value is traded:

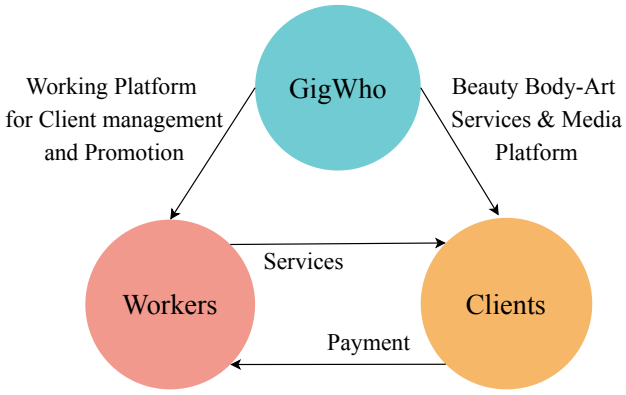

Figure 2: Value Network GigWho Platform

3.3 Development Framework Value Analysis

Before starting the development of the platform one must analyse which solutions are currently available and find the one that suits best. One major factor to consider is **Performance**. It's mandatory to provide a performant experience to all users and no amount of features is significant if the provided experience when navigating through the app is not consistent and efficient. According to the Android development guidelines: *100ms to 200ms is the threshold beyond users will perceive slowness in an application* ([Android, 2019]). Performance and responsiveness are key components that create great mobile experiences. The perception of performance is based on elements like start-up time, page-loading, page-transitions, animations, errors and waiting times. The app must deliver a good first experience since a first bad experience might drift away the user entirely.

Next parameter to look at is the **Learning Curve** and the time it takes to develop. Usually the degree of expertise is related to the time it takes to develop, but it might not be the case for **GigWho**. The framework must be straightforward and follow current development paradigms, have an either known programming language or use one easy to get into, finally be simple enough for basics such as page routing, page layout, connection to external services and clear access to native device functionalities.

Documentation & Community is another considered factor. The framework must have a rich documentation supporting its implementation and examples, and have an active community behind it. Documentation is mandatory and a cooperative community is ideal to guarantee its development is not forgotten within the next few years, or ever, and it frequently gets updated.

Integration & Features also play and critical role. If the framework has already a great integration major mobile features and only require its adaptation to the current business model, it is already ahead.

Finally, **Native vs Cross-Platform** development. Native development will cover 100% API access to all mobile functionalities and grant us more performance with less overhead, notwithstanding cross-platform means it is

not required two separate code bases for the same app (one for Android and another for iOS). At the time of writing, it is already decided to move forward with a cross-platform solution, especially when all considered frameworks offer almost, if not complete, access to the device features. Even though all alternatives are cross-platform, some require less code to work properly on the desired platforms whereas others might require more. One framework might already provide a broad range of styled components while another requires different internal implementations to achieve the desired experience on both platforms.

To summarize the list of attributes considered is:

- Performance
- Documentation & Community
- Integration & Features
- Learning Curve & Basics
- Native vs Cross-Platform

3.3.1 Value Attributes, Value for the Developer and Perceived Value.

Based on solution proposed by T. Woodall — *Conceptualizing Value for the Customer: An Attributional, Structural and Dispositional Analysis* [Woodall, 2003] a new table was built to summarize the notion of value for the developer. Value is obtained if somehow the outcomes achieved from the attributes overcome the sacrifices are:

Attributes	Outcomes	Sacrifices
Performance	User Experience	Time Optimizing
Learning Curve & Basics	Development Time and Experience	
Documentation & Community	User Experience	Time reading through the documentation
Integration & Features	Development Time	Time Integrating developed features and Additional Overhead
Cross-Platform	Development Time and Code Maintenance	Performance

Table 1: Contextualization of the Perceived Value

And the reasons why the following attributes are able to overcome the sacrifices:

- **Performance** leads to better user experience. Even if the framework can handle all the existing scenarios perfectly, the developer should use good coding practices optimizing the application. Those practices should be implemented from the start even though they are often perceived in later development stages when using the application altogether or during comprehensive test scenarios. Therefore, one must sacrifice time to optimize the code independent of the project stage.
- **Learning Curve & Basics** leads to better development experiences. Implementing basic functionality should be easy. Easy to start frameworks with smoothly managed implementations of the basics are welcome by any.
- **Documentation & Community** leads to better development experiences as they cover how-to-build examples. When lacking documentation, one can ask for help from the community. However, before starting developing it is recommended reading the documentation about the framework, it will consume time early-on but might save a lot more in the future.
- **Integration & Features** reduces development time. A Framework that easily integrates existing features such as payments, push notifica-

tions, GPS and more, is a valuable one. The downside of integrating those features versus implementing your own is that for simple uses cases this might carry unwanted overhead.

- **Cross-Platform** means one single code base to maintain. The downside of cross-platform is that it is usually one step away from the performance granted by their native alternative.

Below are the four frameworks considered for the development of GigWho:

1. Ionic + Angular
2. NativeScript + Angular
3. React Native
4. Flutter

3.3.2 The Analytic Hierarchy Process

Analytic Hierarchy Process (AHP) *is a method for multi-criteria decision-making that breaks the problem down based on decision criteria, sub criteria, and alternatives that could satisfy particular goal. The criteria are compared to one another, the alternatives are compared to one another based on how well they comparatively satisfy the sub criteria, and then the sub criteria are examined in terms of how well they satisfy the higher-level criteria.*

The following steps are required to apply the Analytic Hierarchy Process: [Saaty, 2008]

- *Define the problem and determine the kind of knowledge sought.*
- *Structure the decision hierarchy from the top with the goal of the decision, the objectives from a broad perspective, through the intermediate levels (criteria which subsequent elements depend on) to the lowest level (which usually is a set of the alternatives).*
- *Construct a set of pairwise comparison matrices. Each element in an upper level is used to compare the elements in the level immediately below with respect to it.*

- *Use the priorities obtained from the comparisons to weigh the priorities in the level immediately below. Do this for every element. Then for each element in the level below add it weighed value and obtain its overall or global priority. Continue this process of weighing and adding until the final priorities of the alternatives in the bottom most level are obtained.*

The AHP method answers which alternative (blue) is the most adequate to use to develop **GigWho**. The alternatives were introduced, explained and the criteria on which they will be evaluated chosen. The following diagram reflects the AHP ecosystem considered:

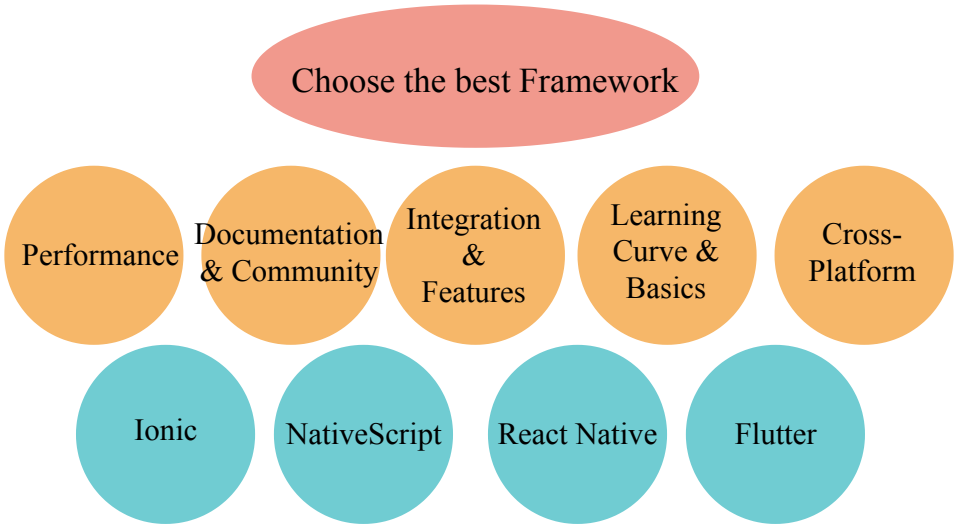

Figure 3: AHP Diagram

After choosing the goal (red), each of the criteria (yellow) must be qualified based on its importance. The assigned weight represents the priority of one over the other. This step is known as Priority Definition and will contemplate the *Fundamental Scale of Pairwise Comparison* [Saaty, 2008].

Classify the level of importance a criterion has over the other. These are the classifications:

- Cross-Platform:

Winner	Weight	Looser
Cross-Platform	[3]	Documentation & Community
Cross-Platform	[4]	Integration & Features
Cross-Platform	[4]	Learning Curve & Basics
Cross-Platform	[5]	Performance

- Documentation & Community:

Winner	Weight	Looser
Documentation & Community	[3]	Integration & Features
Documentation & Community	[3]	Learning Curve & Basics
Documentation & Community	[4]	Performance

- Integration & Features:

Winner	Weight	Looser
Integration & Features	[1]	Learning Curve & Basics
Integration & Features	[3]	Performance

- Learning Curve & Basics:

Winner	Weight	Looser
Learning Curve & Basics	[3]	Performance

The rank assignment between the criteria results in a comparison matrix with the computed weight each criterion has over the other. The matrix was computed using the AHP matrix website.

	A	**B**	**C**	**D**	**E**
A	1	3	4	4	5
B	0.33	1	3	3	4
C	0.25	0.33	1	1	3
D	0.25	0.33	1	1	3
E	0.20	0.25	0.33	0.33	1

Table 2: Criterion Comparison Matrix

Whereas:

- Cross-Platform: **A**
- Documentation & Community: **B**
- Integration & Features: **C**
- Learning Curve & Basics: **D**
- Performance: **E**

The way the criteria was classified is based on general experience developing software applications and is aligned with the goals intended for this thesis. **Cross-Platform** is the most relevant as it is mandatory the development of a single code-base that works on both Android and iOS. Even if the framework has a large community, is well documented and with a flat learning curve, not being cross-platform is not an option.

Documentation & Community comes second because they precede all other criteria during the development process. If the idea is to integrate an already developed solution or implement it yourself, one must read the documentation and gather information on how to do so. The **Learning Curve** is also mollified with documentation and assistance from an active community.

Integration & Functionality as well as **Learning Curve & Basics** have the same priority. Being able to quickly build something from scratch is as important as integrating functionality into the app.

Finally, **Performance** was ranked the least. Even though it's one crucial factor to provide quality user experiences, it is only relevant once full implemented, with users using it. This thesis proposal is about the development of a concept capable of improving the current work-flow for beauty and body-art workers.

All the criteria play an imperative role for a successful development, and ideally no aspect is disregarded in favour of another. The Comparison matrix is followed by the Priority matrix based on the pairwise comparison and the Relative Weight matrix:

- Priority Matrix:

Category	Priority	Rank
A	46.4%	1
B	25.3%	2
C	11.4%	3
D	11.4%	3
E	5.5%	4

Table 3: Priority Matrix

- Relative Weight Matrix:

Category	Weight(Eigenvalue)	Level of Importance
A	0.464	1
B	0.253	2
C	0.114	3
D	0.114	3
E	0.55	4

Table 4: Relative Weight Matrix

After the computation of the Relative Weight Matrix(4), the alternatives come into the mix. Because there are four of them (Ionic, React Native, NativeScript and Flutter), it is required compare them against one another on every criterion. This comparison will be qualified using the same scale used before, from the *Fundamental Scale of Pairwise Comparison*. A brief

explanation is given on why these values are accredited this way. Afterwards, the weights of each alternative for every criterion is used to compute a weight matrix, following the same process as Table 4.

Cross-Platform

Comparison between the winner and other alternatives:

Winner	Weight	Looser
Ionic	[2]	Flutter
Ionic	[3]	NativeScript
Ionic	[4]	React Native
Flutter	[3]	NativeScript
Flutter	[4]	React Native
NativeScript	[2]	React Native

Relative Weight Matrix between the winner with other alternatives:

Category	Weight(Eigenvalue)	Level of Importance
Ionic	0.455	1
Flutter	0.32	2
NativeScript	0.139	3
React Native	0.086	4

Table 5: Cross-Platform — Relative Weight Matrix

This stands for how much of the same code can be reused for multiple platforms (Android and iOS). All the frameworks are known for their cross-platform capabilities and can reutilize code in a decent way, however some are more capable than others. This does not make the platform superior in any way and instead reflects how they were build. With **Ionic**, you're also building for the web. Ionic also comes with a flexible style based component library for both Android and iOS.

Flutter provides Material Design styled components for the Android platform and Cupertino styled components for iOS. It's required to runtime check the device platform to apply the appropriate styled components. This

runtime check is why Flutter is not rated as well as Ionic. **NativeScript**, unlike Flutter, does not apply a styled theme and instead compiles the components into their platform representatives. However, components that do not exist on both platforms must be managed (the UI) by the developer. Finally, **React Native**, like NativeScript, also compiles components to their UI native equivalents but it only provides a basic set of components and requires additional platform specific styling for custom components.

Documentation & Community

Comparison between the winner and other alternatives:

Winner	Weight	Looser
React Native	[2]	Flutter
React Native	[4]	Ionic
React Native	[6]	NativeScript
Flutter	[3]	Ionic
Flutter	[4]	NativeScript
Ionic	[3]	NativeScript

Relative Weight Matrix between the winner with other alternatives:

Category	Weight(Eigenvalue)	Level of Importance
React Native	0.499	1
Flutter	0.299	2
Ionic	0.137	3
NativeScript	0.066	4

Table 6: Documentation & Community — Relative Weight Matrix

As far as Documentation goes, all have plenty of and are equipped with well documented examples and use cases. Personally, the **NativeScript** docs where the most difficult to consult and understand with a not so great overall experience. The core concepts section and the one section dedicated to the different front-end flavours supported was convoluted, especially for someone with no experience. So far, and also on a personal note, **Flutter** has

the best documentation with rich examples and clear descriptions properly explaining how to use. The official Flutter YouTube channel has quick and straightforward overviews on its enormous pool of components (called widgets in Flutter). Both **React** and **Ionic** have their framework well documented with plenty of examples.

Regarding the Community, **React** is by far the most popular one. It has a massive presence on GitHub with 85k stars, is used as dependency by 324k GitHub projects and has, at the time of writing, 2057 total contributors. **Flutter** has about the same stars but fewer contributors. **Ionic** and **NativeScript** (the core package) have 40k stars with 354 contributors and 18k with 159 contributors respectively.

Chronologically, Ionic was released first in 2013, NativeScript in 2014, React Native in 2015 and Flutter in 2018 making it, by far, the most trending framework. In terms of developer preferences from the GitHub developer survey in 2019[2] React Native was the most popular in *Other Frameworks* within the category *Most Popular Technologies*. Flutter is also mentioned in this list but not as well ranked as React; nonetheless in an impressive placement due to how long has it been in the game. In the same category, but for the title of *Most Loved, Dreaded and Wanted Other Framework, Libraries and Tools*, Flutter is ranked 3º as the most loved and React Native the 3º as the most wanted. In Slant[3] (a product recommendation community website) React Native, Flutter and Ionic are very well considered.

Integration & Features

Comparison between the winner and other alternatives:

[2]https://insights.stackoverflow.com/survey/2019
[3]https://www.slant.co/

Winner	Weight	Looser
Ionic	[2]	React Native
Ionic	[2]	Flutter
Ionic	[2]	NativeScript
Flutter	[1]	React Native
Flutter	[1]	NativeScript
React Native	[1]	NativeScript

Relative Weight Matrix between the winner with other alternatives:

Category	Weight(Eigenvalue)	Level of Importance
Ionic	0.4	1
Flutter	0.2	2
React Native	0.2	2
NativeScript	0.2	2

Table 7: Integration & Features — Relative Weight Matrix

This topic evaluates the amount of features (core and third-party) each platform has available and how easy is to integrate them. These features might be for native device access as well as for other functionalities such as payments, requesting data from external sources, maps or even when using BaaS (Backend as a Service) systems. Because **Ionic** uses JavaScript and runs as a web app it has access to most of the functionality on the NPM repository for most of the problems a developer might face. NPM is the second largest package manager and the largest one for the JavaScript ecosystem, with over 1M packages. There are also some Ionic specific packages developed by the community and available via NPM. With Ionic, chances are there's always a solution for most cases. As far as native access goes, Capacitor (the newer runtime engine capable of providing access to the device) covers common use cases such as device storage, background tasks, camera... and any missing native functionality can be implemented and injected through it. Most APIs have fall-backs for each platform (Android, iOS and Web) which makes their integration effortless.

React Native is a JavaScript framework and can use the NPM repository. As for native access goes, it only provides core features and relies heavily on its community to provide third-party tools. The same applies for **NativeScript** but with a higher range of core features available and less third-party developed by its smaller community compared to React.

Flutter does not use JavaScript and instead it uses its own package manager known as Pub. The size of the repository does not come close to NPM, but the most notorious packages were already, or are currently being, ported. As far as native features goes, Flutter also provides multiple from the get go as well as the means for any developer to implement new ones. Even if some alternatives come with more out-of-box functionalities, there's also the ones with larger communities that provide via third-party, so the weight assigned was not much different.

Learning Curve & Basics

Comparison between the winner and other alternatives:

Winner	**Weight**	**Looser**
Flutter	[3]	Ionic
Flutter	[4]	NativeScript
Flutter	[5]	React Native
Ionic	[2]	NativeScript
Ionic	[4]	React Native
NativeScript	[1]	React Native

Relative Weight Matrix between the winner with other alternatives:

Category	**Weight(Eigenvalue)**	**Level of Importance**
Flutter	0.541	1
Ionic	0.252	2
NativeScript	0.114	3
React Native	0.093	4

Table 8: Learning Curve & Basics — Relative Weight Matrix

How easy is to start and implement the basics with each of the alternatives was the considered factor for this criterion. Personally, from prior experience, **React Native** is the most difficult. It uses JavaScript, a language used by most web developers, but its declarative paradigm makes it quite confusing to read without prior experience and especially without type-safe features. On the contrary, **Ionic** and **NativeScript** have a wide range of flavours to work with, leading to smother transitions for web developers who, most certainly, already used some. Also, both support TypeScript, which enforces some type-safety mechanics that JavaScript lacks.

Flutter was considered the easier to start with. Dart syntax is very similar to JavaScript and it is both a dynamic-typed language as it is strong-typed. Instead of interpreted, it's compiled, which increases development speed by providing errors on compile. Flutter is the clear winner, with React Native as runner-up, both use a declarative approach making the code easier to maintain and develop. The weight between Ionic and NativeScript, when comparing both directly, is low; however, they have a similar learning curve because they can be used with the same frontend engines (Angular and Vue) flattening the curve when transitioning from one to another.

Performance

Comparison between the winner and other alternatives:

Winner	**Weight**	**Looser**
Flutter	[2]	React Native
Flutter	[3]	NativeScript
Flutter	[9]	Ionic
React Native	[3]	NativeScript
React Native	[8]	Ionic
NativeScript	[8]	Ionic

Relative Weight Matrix between the winner with other alternatives:

Category	Weight(Eigenvalue)	Level of Importance
Flutter	0.467	1
React Native	0.325	2
NativeScript	0.172	3
Ionic	0.036	4

Table 9: Performance — Relative Weight Matrix

Ionic is the least performant of all. **NativeScript** comes next. **Flutter** and **React Native** are placed quite nearly. Flutter takes the first place as it compiles all code base to native machine code striving for that extra 60fps performance.

The following step requires the sums of the products between each alternatives' criterion weight and the criterion weight relative to the others. The value *Global Weight of the alternative* can be obtained via the following process: If alternative μ scored the values x and y for criterion A and B, respectively, and the relative weight of A and B criteria is α and β then the global weight of the alternative can be calculated via the given formula:

$$GW(\mu) = x * \alpha + y * \beta \tag{1}$$

Therefore:

- Computed Global Weight Matrix:

	A		**B**		**C**		**D**		**F**		GW
Flutter	0.1491	+	0.0756	+	0.0228	+	0.0617	+	0.0257	=	0.3349
React Native	0.0401	+	0.1262	+	0.0228	+	0.0105	+	0.0179	=	0.2175
Ionic	0.212	+	0.0347	+	0.0456	+	0.0287	+	0.002	=	0.323
NativeScript	0.0648	+	0.0167	+	0.0228	+	0.013	+	0.0095	=	0.1267

Table 10: Global Weight of the Alternatives

- Ordered by classification:

Alternative	Placement
Flutter	**1º**
Ionic	2º
React Native	3º
NativeScript	4º

Table 11: Framework Ranking

The winner is **Flutter**.

4 Evaluation Planning

4.1 QEF

To develop the solution proposed for this thesis it is required to establish a comprehensive view of the system and assess its quality to evaluate the processes involved, over time. The **QEF** (Quality Evaluation Framework) is a framework capable of measuring the quality of a certain system during its development life-cycle and provides a communication layer for all parties involved, from the stakeholders/clients, product owners to the development team. The QEF model will be used to:

- Split all features into application' **Dimensions**.
- Identify **Quality Factors** that are relevant for each **Dimension**.
- Identify **Requirements** for each **Quality Factor**.
- Define **Metrics** for those **Requirements**.
- Measure the quality of the system as a whole.

In order to identify the Dimensions and its Quality Factors it is required to build a detailed list of Requirements (functional and non-functional) about what the system should do and how it should behave. A weight (2, 4, 6, 8, 10) is assigned to each requirement (Requirement Weight) to represent its relevance, alongside the metrics capable of measuring its completion. Those metrics are levels of fulfilment (%). All items are then grouped into Quality Factors which in turn are grouped into Dimensions. It is critical to keep track of each requirement fulfilment level using a model such as QEF to ensure the members of the team make good overall measurements of the current quality of the system.

From the analysis on what the application must do and how it should behave, the following **Dimensions** were created alongside their quality factors:

- Functional:
    1. General

2. Social
3. Work
4. Settings
5. Search
6. Static

- Non-Functional:
  1. Adaptability
  2. Scalability
  3. Performance

GigWho quality factors and requirements are the following:

Requirement	**Metric Evaluation**	**100%**
FG01 — User creates a User Account.	User creates a User Account to use **GigWho**.	User creates a User Account.
FG02 — User creates a Store Profile.	User creates a Store Profile becoming a worker for the platform.	User creates a Store Profile.

Table 12: ***Functionality*** *— Quality Factor: General*

Requirement	50%	75%	100%
FS01 — Worker creates, edit and removes posts from their store profile.		Worker creates and removes posts.	Worker is able to edit a post.
FS02 — User "likes" a post from a worker's store profile.		Double-taping marks a post as "liked".	Workers are notified when a user "likes" one of their posts.
FS03 — User "shares" a post to outside the platform.			User "shares" a post to outside the platform.
FS04 — User "follows" a worker's profile page.	Users "follow" a worker's store profile page.	Workers are notified when a user starts following their store profile page.	Users are notified when the store profiles they follow have new posts.
FS05 — User applies category filters to the content being shown by the app by applying a category filter (based on the supported categories).			Users combine and apply filters to get only the desired content.

Table 13: ***Functionality*** *— Quality Factor: Social*

Requirement	Metric Evaluation	50%	75%	100%
FW01 — Users make proposals to Workers.	Users make proposals by selecting the service(s), a place and suggesting a time-frame.		Users can create proposals.	Users can reply with shorter time frames to responses from the workers.
FW02 — Workers create, edit and remove services from their service menu.				Create, edit and remove services from their service menu.
FW03 — Workers create, edit and remove work addresses.			Workers create, edit and remove work addresses.	Users are notified when a worker they follow creates a new work address.
FW04 — Users (workers and clients) can signal movement related activities.	Workers and clients can signal each other about movement activities related to the appointment, notifying each other when doing so.		Workers can signal a *ready* state to their clients, and clients a *going* state. Both imply sending a notification to the other.	Within a certain radius from the appointment location, clients can signal workers they have *arrived*. Also implies notification being sent.
FW05 — Users change the state of an appointment.	Workers and clients are able to cancel appointments. Workers are able to change the location of an appointment. When the state of an appointment changes, both get notified.	Workers and clients are able to cancel appoint-ments.	Workers are able to change the location on their appointments.	When the state of an appointment changes, both parties are notified.
FW06 — Users can filter appointments by multiple parameters.	Users are able to filter and sort appointments.		Sort by price, duration, distance.	Filter by service name.

Table 14: ***Functionality*** *— Quality Factor: Work*

Requirement	Metric Evaluation	50%	75%	100%
FSS01 — Worker mute proposals.	Workers can mute proposals for specific periods or mute altogether disabling the users' capability to make new proposals from their store profile.	Worker mutes proposals (*enable-disable*).	Worker mutes proposals for individual defined time periods.	Worker mutes proposals for multiple periods. Periods have a highly customizable pattern that can be manipulated by the worker as he seems fit.
FSS02 — Workers customize the type of work they perform.	On **GigWho** worker can specify if they perform work at their place, if they are able to meet the client at theirs, or both.			Worker has a setting to customize this.
FSS03 — Worker customizes a rate per kilometre.	If the worker is the one meeting the client, he can customize a rate per kilometre.			Worker is able to customize this value.
FSS04 — Worker customizes the max number of proposals.	To avoid proposals congesting, workers are able to specific a max number of proposals they can keep active at the same time.			Worker is able to customize this limit.
FSS05 — Worker customizes the max number of services per proposal.				Worker is able to customize this limit.

Table 15: ***Functionality*** *— Quality Factor: Settings*

Requirement	**Metric Evaluation**	**50%**	**100%**
FSH01 — User search for Store Profiles.	User is able to search for Store Profiles by their name, owner/worker's name or specific locations or points of interest.	Search by Store and Worker name.	Search by Location and Points of Interest.

Table 16: ***Functionality*** *— Quality Factor: Search*

Requirement	Metric Evaluation	50%	75%	100%
FST01 — Mobile Application must show a marker on top of each Store location.	Mobile Application must show a marker on top of each Store location and cluster these based on the zoom level applied and the distance between themselves.	Displays a marker for each store located in the map.		Cluster store locations to provide a clear map view to the user.
FST02 — Content is shown in a suggestive manner.	Content is shown based on parameters such as the current user location and the most trending stores.	Content is shown considering the users' current location.		+ Considers the most trending stores (more likes, shares and visits between a certain period).
FST03 — Content is requested and shown automatically, as the user scrolls further.	Application implements infinite scroller technique for the majority of its screens/views, where data is processed in blocks (requested to the server and rendered onto the screen).			Processing new blocks is triggered by user scroll movements.

Table 17: ***Functionality*** *— Quality Factor: Static*

Requirement	**Metric Evaluation**	**75%**	**100%**
A01 — Mobile Application must be implemented for both Android and iOS platforms.	Mobile Application runs natively on each platform.		Apps runs natively on both platforms without UI differences.
A02 — Interface language according to selected language.	Mobile Application interface must support, at least, two languages: Portuguese and English.	Only Portuguese or English is available.	Both languages are available.
A03 — Application must adjust itself to different screen resolutions.			Application is capable of adjusting its UI to different screen sizes and, therefore, multiple smartphones.

Table 18: ***Non-Functional*** *— Quality Factor: Adaptability*

Requirement	**Metric Evaluation**	**100%**
S01 — Mobile Application connects to an external server.	Mobile Application communicates, sends and retrieves data from a centralized server infrastructure.	Implementation of an HTTP web server.
S02 — Mobile Application should only store data required for business logistics.	Mobile Application should not store any sensitive information about the user.	No sensitive data stored.
S03 — Backend infrastructure should scale.	The backend infrastructure should self adjust for periods with increased workloads, where the number of connection is high.	Backend infrastructure is implemented in a way that allows it to scale horizontally.

Table 19: ***Non-Functional*** *— Quality Factor: Scalability*

Requirement	**Metric Evaluation**	**75%**	**100%**
P01 — Mobile Application should load low resolutions assets as a placeholder on low-quality connections.	Loading low resolution assets on low quality connections to increase app responsiveness. The app proceeds to load the original asset if the user remains with it displayed.	Always loads the low resolution version before the original.	The app decides whether to load a low resolution asset or not based on the quality of the connected network.
P02 — Assets and data are cached locally to improve app responsiveness and network usage.	Data that relies on external services should be cached locally, especially for data that is expected to be static most of the time, e.g: image data.		Assets are cached.

Table 20: ***Non-Functional*** *— Quality Factor: Performance*

5 Design

The Design section explores the most relevant features that make **GigWho** a capable platform to be used nowadays in the work and media sectors. It has an overview of the solution and the technicalities associated with. Also, it showcases how the solution is designed resorting to UML (Unified Modelling Language), a general purpose modelling language used in software engineering to provide a standard way to visualize how software works and how it is structured.

This project covers the development of a complete framework, from the frontend to the backend and database administration, while also using remote services for image storage, provide a 2D map view and manage push-notifications. The main focus, however, is mostly aimed towards the client frontend application. The approach taken was the presumed best to expose the application with all its intertwined elements. For its development, **GigWho** was divided into core features, each one with a dedicated written subsection. A single feature raises a specific set of use cases and it might also be associated with one or more specific app views.

5.1 System Architecture

A deployment diagram describes the topology of a system, physically. What software components are running on which hardware infrastructure (nodes). **GigWho** has the following deployment architecture:

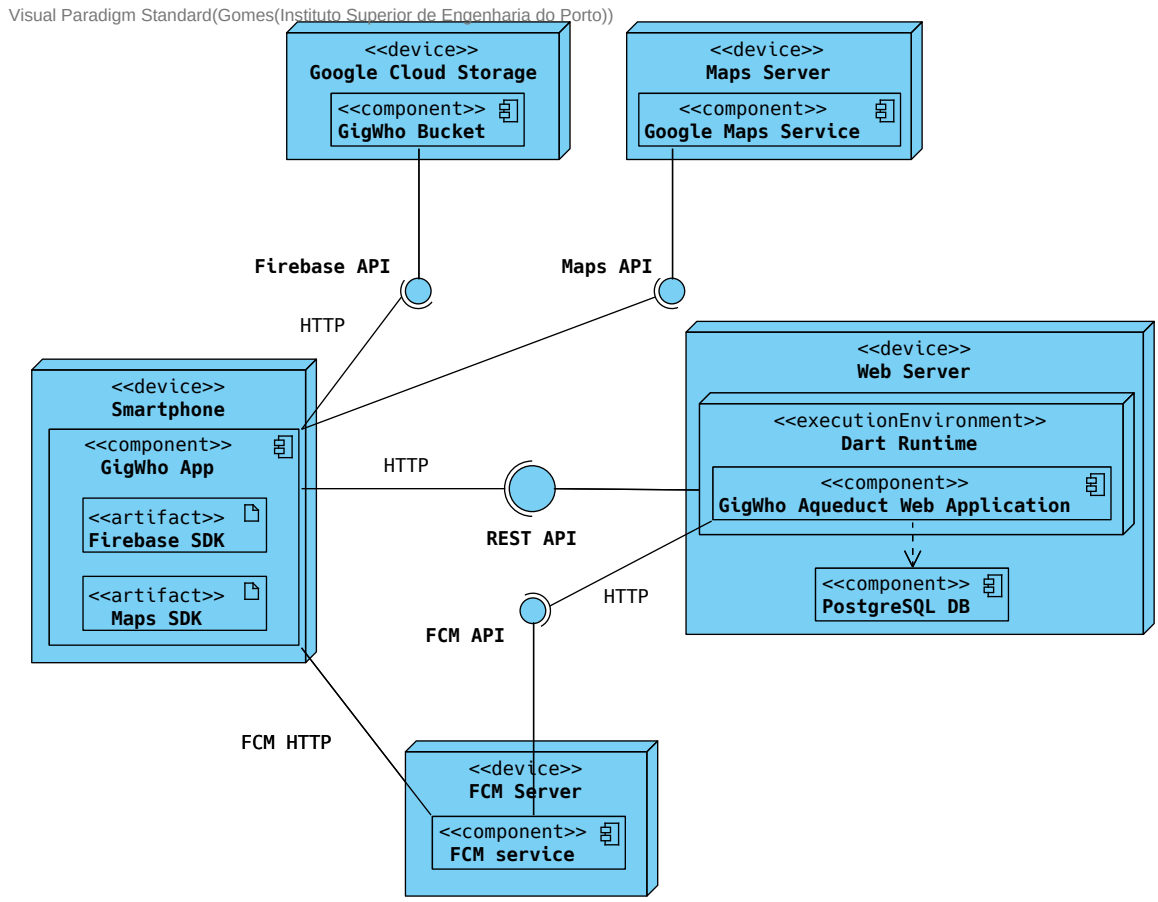

Figure 4: Deployment Diagram

The system is divided into two main components, the **GigWho** app, running in the users' smartphone and the backend Aqueduct[4] server connected to a PostgreSQL database to handle data persistence. The Aqueduct service

[4]Framework used to build server applications with Dart — https://aqueduct.io

exposes a REST API to be consumed by the client. Data flows through this API using HTTP and exchanging JSON payloads.

For image storage **GigWho** utilizes a specialized service provided by Google, the Google Cloud Storage, and uses the Firebase SDK to access its API with functionality to upload, download and allocate assets. Data is sent and fetched via HTTP, and the URL assigned to each item is stored in the **GigWho** web server. The decision behind using an external dedicated server for media is attributed to how simple it is to integrate with **GigWho**. It provides notorious capabilities to handle uploads, downloads and storage or large files, guaranteeing security even for unstable connections. It has the ability to scale automatically and allows user-specific rules for each access type (read, write, delete). Also, storing image data in the same relational database used for the business logic would massively increase its size and reflect on its cost and performance.

To integrate the map view and some map/location related functionalities **GigWho** uses the Google Maps Platform for Maps and Distance Matrix API. The first provides map tiles and information to identify locations (street name, places...), the latter is for path computations, such as the shortest path, its distance and the time it takes to go from one point to another, using multiple travel options (by foot, bicycle, car).

Finally, for the push-notification functionality, **GigWho** uses a service provided by the same Firebase suite, the FCM (Firebase Cloud Messaging), easily accessible through the Firebase SDK. For each device a token is generated and sent to the **GigWho** web server to be stored. Once a certain event that triggers a notification occurs, the server uses that device's token to send a message, with limited payload, via the FCM server, to the device associated with that token.

5.2 Non-Functional Requirements

This subsection covers non-functional requirements outside the scope of the individual core features:

- **Interface** — **GigWho** must properly convey simplicity while being appealing and identical to solutions that are used often. The UI should adapt to most conventional screen sizes. It should not block the user-interface and display the proper loading indicators, when necessary, such as shimmers, skeleton text or download and render low-resolution representations of the original assets thus decreasing the download size and leading to a faster render process.
- **Performance** — To increase performance when using the app image data should be cached locally, to avoid downloading the exact same image every time it is required. Data fetched from a remote source should also be cached locally, whenever possible, and used prior to further identical requests. This makes the app usable in the absence of any connection, on in the presence of extremely slow ones, always displaying data to the user, even if it's not the last version of the resource.
- **Implementation** — **GigWho** must be implemented for both Android and iOS systems and not contain platform specific behaviour, making it equal on both platforms.
- **Scalability** — **GigWho** backend must have the ability to adjust the current system according to the demand, capable of handling heavy workloads when required, with no stalling.

5.3 External Packages used in GigWho

GigWho was developed using Flutter, a Google framework built with Dart and thin layer of C++. The Dart ecosystem has a diverse open-source repository[5] with multiple packages to extend the functionality of Flutter mobile apps as well as any other piece of software that uses Dart or Flutter. As such, several packages were used to build **GigWho**, speeding up the development process while imposing Dart language best practices.

The following is a list of dependencies used by **GigWho**, from the ones that made it to the final solution, to the ones only used in a development environment:

Dependencies — Development and Production:

- Cached Network Image.
- Device Info.
- Equatable.
- Fluster.
- Flutter Bloc.
- Flutter Cache Manager.
- Flutter DotEnv.
- Flutter Secure Storage.
- Geocoding.
- Geohex
- Geolocator.
- Google Directions API.
- Google Maps Flutter.
- Image.

[5] https://pub.dev/

- Image Picker.
- JSON Annotation.
- Local Storage.
- Mask Text Input Formatter.
- Shimmer.
- String Validator.
- Provider.
- Share.
- Uuid.
- Firebase Core.
- Firebase Storage.
- Firebase Messaging.

Development only Dependencies:

- Faker.
- JSON Serializable.
- Logger.

Throughout this and the following section (Implementation) it's explained how were this packages used.

5.4 GigWho Core Features

To proper develop **GigWho** it is required an acute clarification of the core app features and their purpose. The underling design was built taking into consideration the following major components:

1. Store Profile.
2. Store Preferences Customization.
3. Client-Worker communication.
4. Appointment & Proposal Management.
5. Search Functionality.
6. Feed View.
7. Map View.
8. Follow View.
9. Payment Methods.

Before delving into the clarification of the items above, it's essential to describe the following concepts used by **GigWho**:

- **User** — A user of the **GigWho** app. Clients and workers are users. When a user signs up to use the platform, he can perform client related actions. When a user creates a Store Profile, he becomes a worker while still having the client status for other workers on the app.
- **Service** — A work item with a cost and duration, services belong to a Worker (e.g.: Barber: Simple cut — €7 — approx: 30min).
- **Client** — A user who requests for services.
- **Worker** — A user who provides services (e.g.: Barber, Hairdresser).
- **Category** — Categorizers for the type of work performed: Barber, Body Piercing, Hairdresser, Makeup, Manicure, Pedicure, Tattoo Artist.

- **Proposal** — Created by clients, through a store profile, and exchanged between clients and workers until they both settle for an **appointment**. Has predefined forms with information such as the requested services, duration estimate, total cost, initial and final dates (points in time) and a work location.
- **Appointment** — Denotes a commitment between the client and the worker. Through a successful **proposal** comes an **appointment**, and they mostly share the same data.
- **Time-Frame** — Time interval between two points in time.

5.4.1 Store Profile

Workers are able to create and maintain a profile for their store. The profile has identifying information as well as locations, image gallery and the services it provides. While workers are responsible for what is maintained in their profile, **GigWho** has the obligation to make it clean, usable and identical to something they already know — requirements that are driven by the **Aesthetic Usability Effect**, *Users often perceive aesthetically pleasing design as design that's more usable*, and the **Jakobs Law**, *Users spend most of their time on other sites. This means that users prefer your site to work the same way as all the other sites they already know* — ([Yablonski, 2020]).

GigWho store profiles are inspired by some of the most used apps nowadays (Instagram, Uber). Users become more prone into using the app, and using it correctly, if acquainted with how it works.

Workers are able to upload new photos their profile gallery and build their service menu with the services they perform.

Users can socially interact with workers by *liking* their gallery photos, *following* their store profile and *sharing* their content with external platforms.

The Use Cases associated with this feature are:

- **UC01** — User creates a User account.
- **UC02** — User creates a Store profile, becoming a Worker.

- **UC03** — Worker creates, updates or removes services from their service menu.
- **UC04** — Worker creates, updates or removes a post from their profile.
- **UC05** — User "likes" a post from a worker's image gallery.
- **UC06** — User "shares" a post from a worker's image gallery to outside the **GigWho** platform.
- **UC07** — User "follows" a worker store profile.

These use cases are mostly associated with a specific app view, the Store Profile View.

The following are the functional requirements that result from **UC04**, **UC05** and **UC07**:

- User is notified (by push-notification) if the worker added a new photo to its image gallery (only applicable if the user is "following" the worker's store profile).
- Worker is notified when a User "likes" one of their images or "follows" their store.
- Worker is notified when a User "follows" their Store Profile.

Use-Case Diagrams for this feature:

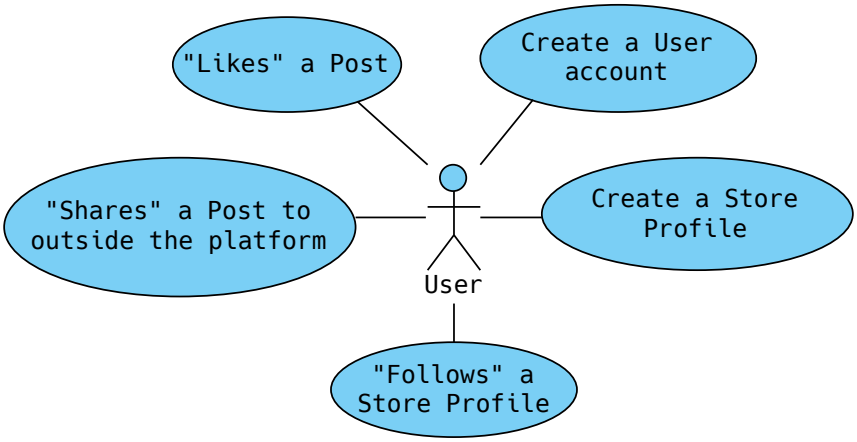

Figure 5: Store Profile User User-Cases

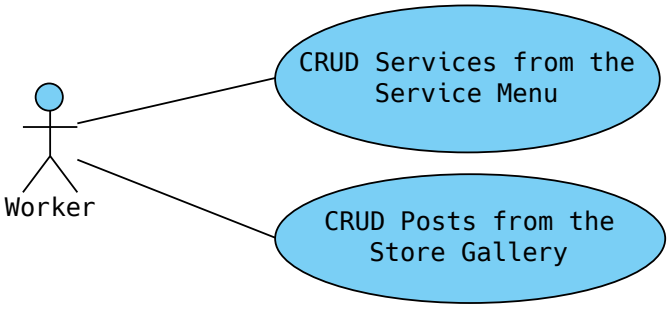

Figure 6: Store Profile Worker User-Cases

5.4.2 Store Preferences Customization

Workers have full control over their workflow. They are able to block/mute incoming proposals between specific time frames or mute proposals altogether, choose a max number of active proposals per day as well as define a max number of services a client can select per proposal. Mute periods can be repeated daily, weekly, monthly or yearly.

Workers can maintain multiple work places while having one as default to welcome their costumers, and state if they perform on-location work, meeting the client at their preferable place. In **GigWho** workers can choose between the following work options: **fixed** — only works through their own places; **on-location** — is able to meet the client; or **both**. When **on-location** or **both** is the option selected by the worker, in their store preferences, they also must set a rate (in €) per kilometre.

These work accessibility features exist to provide convenience and control.

The Use Cases associated with this core feature are:

- **UC11** — Worker mutes proposals.
- **UC12** — Worker mute proposals between a specific time frame.
- **UC13** — Worker customizes a max number of pending proposals per day.
- **UC14** — Worker customizes a max number of services that fit in a single proposal.
- **UC15** — Worker creates, updates or removes work locations.
- **UC16** — Worker indicates the type of work they perform: Fixed, On-Location or both.
- **UC17** — Worker sets a rate per kilometre.

These use cases are mostly associated with a specific app view, the Store Settings View.

Use-Case Diagram for this feature:

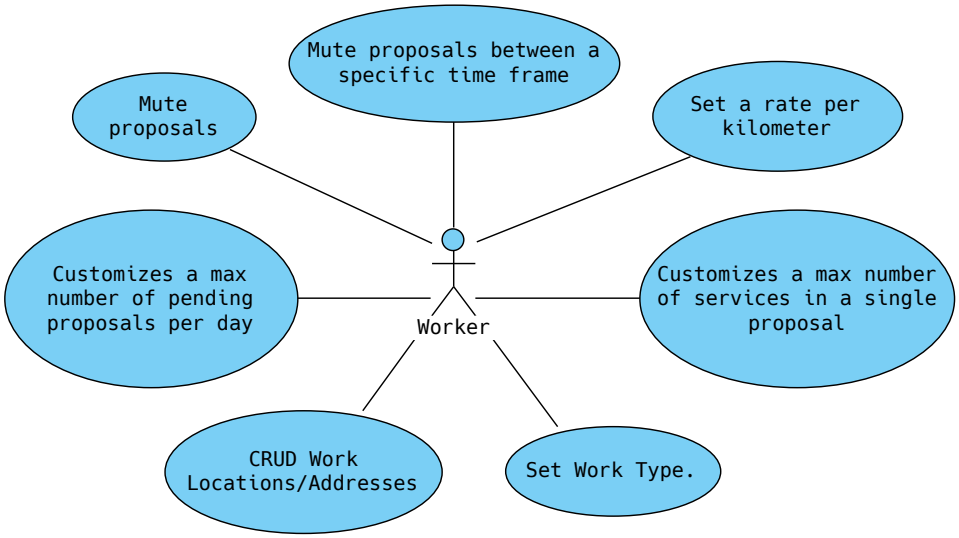

Figure 7: Store Preferences Worker User-Cases

The following are the functional requirements that result from **UC15**:

- User is notified (by push-notification) if the worker added a new address to its address list.

5.4.3 Client Worker Communication

• Making a Proposal

A leading advantage for **GigWho** is the ability to schedule appointments with no client-worker voice or text interaction (unlike other social media platforms, such as Instagram, where clients and workers communicate via text). Communication in **GigWho** is seemingly and suggestive through the exchange of forms with predefined data parameters, between both parties, until they settle for an actual appointment. This group of forms is called a Proposal.

Before scheduling an appointment, clients and workers go through a process of exchanging the following required information, via predefined forms:

- Service(s) the client is interested in.
- Work Location.
- A starting point-in-time from where the client suggests their availability (and/or **4**).
- A final point-in-time to where the client suggests their availability (and/or **3**).

From any Store Profile, users are able to create a Proposal. It begins by selecting the required service(s). To each service is associated a cost and a time estimate for how long it takes to complete. After that, the client proceeds to pick a location by selecting one of the two options: *Meet the worker* at their place or choose their *own location.* The latter is solely available if stated in the worker's store preferences, as mention in the previous feature. Ultimately, the client suggests a valid time frame, restrained by the worker's mute periods and scheduled appointments. In total, there are **three** steps needed to finalize a proposal. Before submitting, the app shows a summary with every detail for the client to confirm.

• **Selecting a time-frame (3rd and final step of a proposal)**

In **GigWho**, working time-frames, or slots, are not delineated by the exact duration of the proposed service(s) but instead need to encompass a big enough interval to allow workers to dynamically manage their availability as they receive more proposals. Valid time-frames are automatically generated as soon as the client chooses the desired date, and are build considering the following two conditions:

- Worker's Mute periods.
- Worker's current appointments with other clients.

With **GigWho** the end goal is not to immediately attain a *Yes* or *No* answer from the worker as the proposal is made. This would require the client to denote exactly the time they wanted to be attended without providing

any flexibility, and the worker would end up rejecting other proposals due to daily time preferences and already scheduled appointments for that same time-frame. Instead, the worker is the one proposing the client the trimmed down slot for him to accept or not.

When choosing a time-frame, clients select an initial and a final point-in-time or can just select one of the two. If only the initial point-in-time is selected, it is assumed, by the worker, the client is available to be attended from that point onwards, whereas if only a final point-in-time is mentioned, available from that point backwards; otherwise, the difference between the final (Tb) and initial (Ta) points-in-time must be equal or higher to the summation of the duration of the selected service(s) plus an additional value (t) to allow workers to trim down the proposed time-frame and properly manage their schedule. If the proposal is *on-location*, the commute estimated time (d) is added to the total duration of the proposal, otherwise it defaults to 0.

$$x = \Delta T = Tb - Ta \tag{1}$$

$$x >= \sum_{i=0}^{n} si + t + d \tag{2}$$

The additional t value is essential and what provides workers with enough flexibility to dynamically fit in multiple clients. If every client sends a proposal from 15:00 to 16:00 the worker would be obliged to accept one, maybe two, and cancel the others. Whereas if those same proposals included longer time frames, such as 14:00 – 18:00, the worker would be able to fit the first between 14:00-14:30, the second between 14:30-15:00 and so on, ending up getting the most out of their schedule.

GigWho proposal model expects clients to provide a considerable time frame capable of accommodating multiple alternatives.

To proper understand this, consider the simplified process below for a particular scenario:

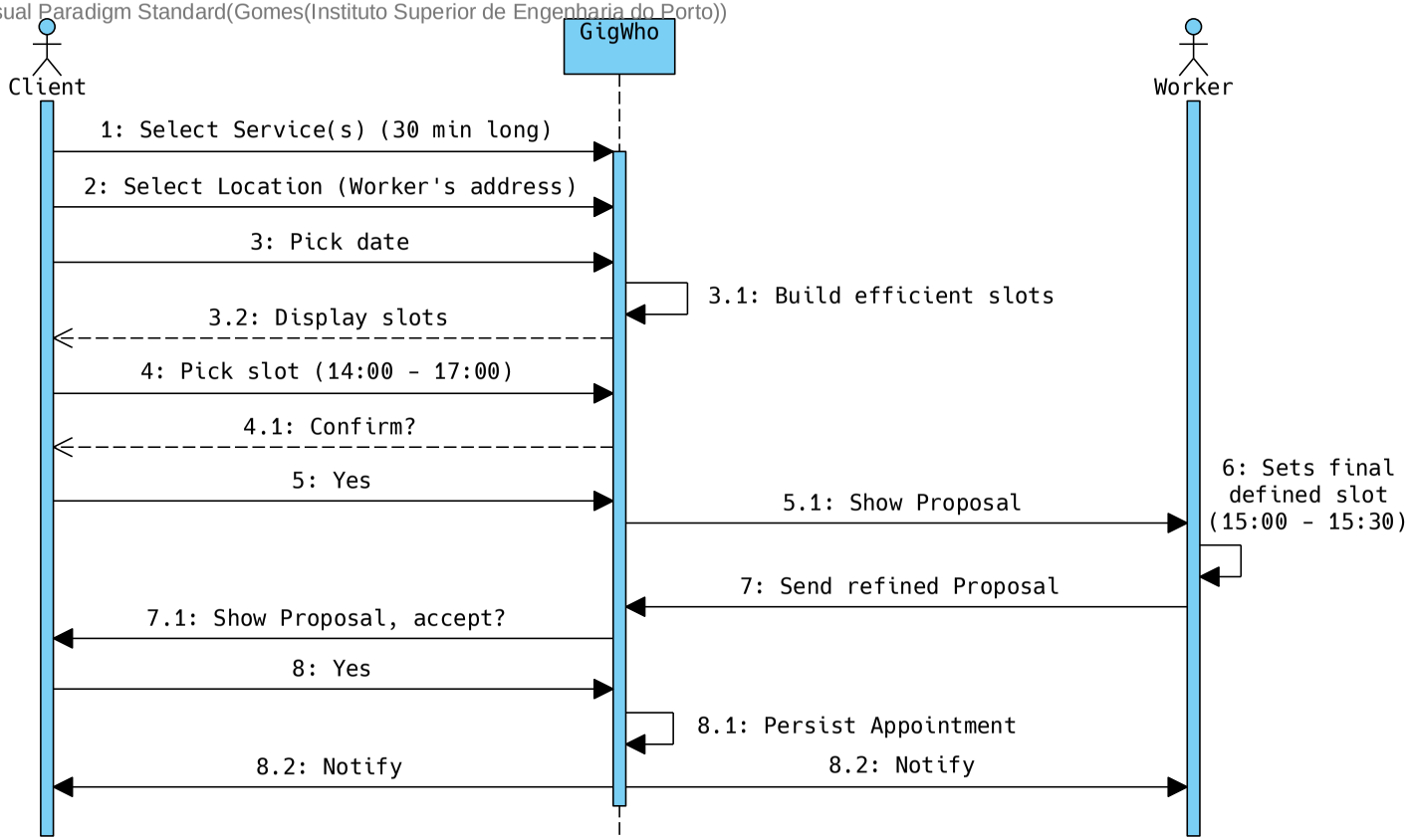

Figure 8: Proposal process simplified overview

After initiating the proposal process:

1. The client chooses the service(s) their interested in (30 minutes long for the current scenario).
2. Client picks a location (Worker's address was picked — *fixed* work type).
3. Client picks a date.
   (a) **GigWho** app fetches relevant data from the **GigWho** backend and builds available time slots.
4. The client selects a 14:00-17:00 slot, meaning their willing to be attended in between that. The total duration of the selected services is only 30 minutes, the app added a t value of 2:30 hours.
5. App displays the proposal overview for the client to review and confirm. Client confirms, the Proposal is created and sent to the server.
6. Server sends the initial proposal to the worker to be analysed. The worker defines a definitive time frame, from 15:00 to 15:30, to accommodate the client.
7. The worker sends the redefined proposal.
8. Client accepts, an Appointment is created and both entities are notified.

On **3.1**, **GigWho** builds slots based on the aforementioned parameters such as mute periods and the worker's already scheduled appointments. Ultimately, the worker is the one deciding when to accommodate the client and the client the one accepting or rejecting. On step **8**, the clients accepts the time-frame proposed by the worker, the app creates an appointment and notifies the client and the worker about its success. Clients can also reject, cancelling the proposal process altogether. With **GigWho** everything is easily managed with a few clicks.

However, the complexity of showing manageable time-slots extends beyond than just adding more time. Workers might have a slot that perfectly fits a proposal between two other appointments; and what if the client dislikes

the slot proposed by the worker and instead want to suggest another more precise one? In the Implementation section this process is comprehensively explained using explicit diagrams.

Why not let the worker define a working schedule divided into slots where the client checks which slots are available and decide which one(s) to select?

GigWho expects workers to manage their own time without being tied to predefined daily agenda, especially if there's on-location type of work involved. Even though it is expected that most would prefer to work from their place within a 9 to 5 schedule, there's also those who don't, and choose unconventional working schedules. **GigWho** model fits both.

- **Time to reply and cancellations**

Clients can cancel the proposal at any given time during the proposal process. However, they cannot make two proposals with overlapping times. In this case, the current active one has to be cancelled in order to start another. Workers can also cancel, however, unlike clients they can keep overlapping proposals if the clients take more time than the expected to reply (this only concerns proposals that were already replied to), until an appointment is settled for.

Clients have a short time limit to reply. When the worker is currently waiting for the client to reply (either by confirming, rejecting or selecting a new slot) they have to respect that short limit and cannot overlap that slot with new proposals. If the client exceeds the limit defined, the worker can start consider proposals for that same period, until an appointment is created. Clients can still reply outside the duration limit, but without guarantees the spot will be available and getting notified if so. This way, the worker is not left hanging for a reply that might never come, possibly losing other clients.

The Use Cases associated with this feature are:

- **UC21** — Client makes a Proposal to a Worker.

This use case is associated with a specific app process, the proposal process, that has multiple views for each stage.

Use-Case Diagram for this feature:

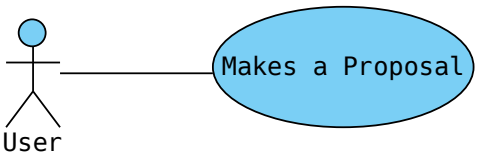

Figure 9: Make Proposal Use Case

The following are the functional requirements that result from **UC21**:

- Both clients and workers (user of the platform) are notified (by push-notification) every time there is a proposal reply from one to the other.

5.4.4 Appointment & Proposal Management

The Proposals and Appointments view should simplify the decision-making process and be as clear as possible. The UI should aid the worker in managing appointments/proposals and display alerts/confirm modals for more sensitive operations such as deleting (cancelling) or change locations for appointments/proposals individually or as a group; while also disposing simple features such as sorting by multiple factors (price, duration, work type, distance) and arrange appointments/proposals through a calendar type view.

When changing default work location, the worker is prompted with an option to either move all scheduled appointments to the new address or not. Clients are notified if their appointments' location change and asked how they want to proceed (cancel, ignore...). The same process applies for ongoing proposals.

• **Movement related notifications**

Workers and clients are able to properly notify each other for movement related activities (*going*, *arrived* location (when inside a certain radius) and *ready*). *Going* applies to the one meeting, if the client is the one *going* to meet the worker, it can signal the state at that moment notifying the worker of its departure. The same client can also notify the worker with an *arrived* state, once inside a certain radius from the appointment location. Lastly, the *ready* state applies only from workers to clients to signal they are absolutely ready to attend the client.

The Use Cases associated with this feature are:

- **UC31** — Worker can change the locations for a live proposal or scheduled appointment.
- **UC32** — User (both client and worker) can cancel an appointment or a proposal.
- **UC33** — User can notify each other on movement related activities. Such as: *going*, *arrived* and *ready*.
- **UC34** — User can filter proposals and appointments through a variety of options.

These use cases are mostly associated with two specific app views, the appointments view and the proposals view.

The following are the functional requirements that result from **UC31**, **UC32** and **UC33**:

- User gets notified if the state of the proposal/appointment changes, such as: location, state.
- User gets notified on movement related activities. Such activities are: *going*, *arrived* and *ready*.

Use-Case Diagrams for this feature:

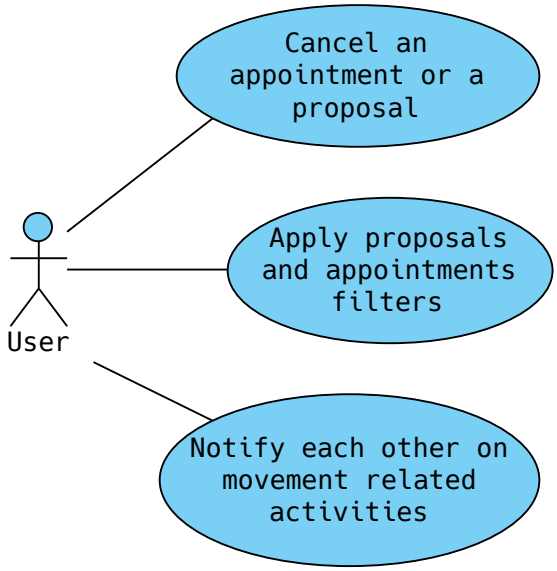

Figure 10: Appointment Management User User-Cases

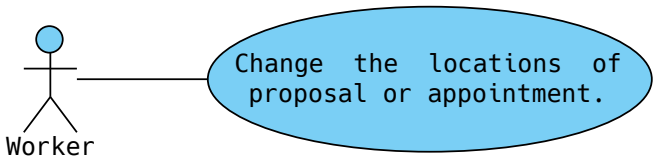

Figure 11: Appointment Management Worker User-Cases

5.4.5 Search Functionality

Users are able to search for stores by its name, worker's name and places (named locations). The search functionality has three options (name, worker name and places), while the Feed and Map View allow filtering through categories.

The Use Cases associated with this feature are:

- **UC41** — Users search for stores by providing its name, worker's name or places (named locations).

  This use case is associated with the Search View.

Use-Case Diagrams for this feature:

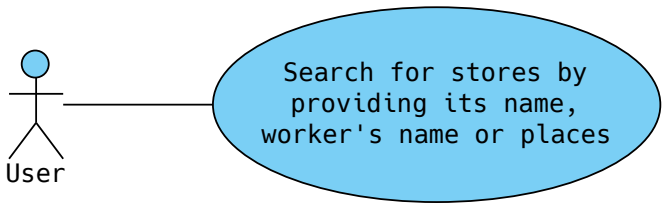

Figure 12: Search Use Case

5.4.6 Feed View

The Feed view is the main view of **GigWho** and the one displayed when opening the app. It exhibits a gallery using images from multiple stores with beauty and body-art related content across the multiple supported categories. Its purpose is to not only entertain users, but also drive them into each store profile. Content is shown sorted from the latest to the oldest, and accounts for the category filters applied. It also considers the user's current location and trending stores.

This is an image driven feature and **GigWho** an image first app. Visual content is one major factor that drives users engagement as human perception is mostly visual, and is more likely to be shared across the respective (or different) platform, between friends, family and business partners — [Manic, 2015].

While **GigWho** is mainly a working platform, it is also a place for users to enjoy and appreciate the art made by others.

The Use Cases associated with this feature are:

- **UC51** — Users use a category filter to only display content they want.

Use-Case Diagram for this feature:

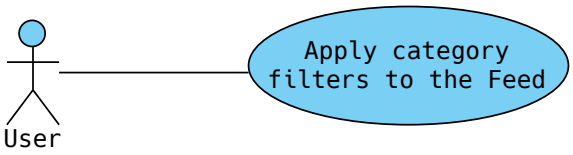

Figure 13: Apply Category Filters to the Feed

The functional requirements associated with this feature are:

- Feed view is influenced by parameters such as the user's current location and trending stores. Content is shown considering both factors.

5.4.7 Map View

The Map view lays out all store locations in a 2D map view. The user interacts directly with the map through drag and zoom movements. Store locations are clustered based on the zoom level applied. Zooming-in splits clusters into smaller ones or reveals actual store locations. The Map View delivers space awareness allowing users to quickly identify which stores are available on certain locations.

The Use Cases associated with this feature are:

- **UC61** — Users use a category filter to only display store locations from wanted categories.

Use-Case Diagram for this feature:

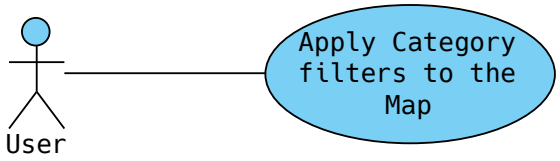

Figure 14: Apply Category Filters to the Map

The following non-functional requirements arise:

- App is expected to cluster locations to not congest the view with multiple Store markers. Clusters are fragmented based on the distance between store locations and the zoom level applied.

5.4.8 Follow View

The Follow View shows users content from the Store Profiles they choose to follow. Content is disposed chronologically, from top to bottom.

5.4.9 Payment Methods

Typically, a client would pay after the service is done either with physical money or by card. Nowadays, people are adhering to more convenient methods such as MB Way. As such, there isn't necessarily a motivation to implement in-app payments whereas the aforementioned are highly used and even preferable, in terms of data storage and security. However, due to the nature of the app, the price is always disclosed beforehand so the client knows exactly how much it needs.

5.5 Entity Relationship Diagram

The following is the Entity Relationship Diagram (ERD) to demonstrate how **GigWho** entities (data objects) relate to one another and how (and what) data is stored in the Database.

The two participants mentioned in the Value-Analysis chapter, that constitute **GigWho** value-chain (the client and the worker), are represented by `GwUser` and `GwStore` respectively.

To use the app it is required an access account represented by a `GwUser` entity. Users become workers by creating/opening a store, represented by the `GwStore` entity, through the same app. Structuring the system this way allows users to be both clients and workers, with the same access account, using the app to its full extent, without having to install a second dedicated worker app.

After the diagram, it is explained the role of each entity in the system.

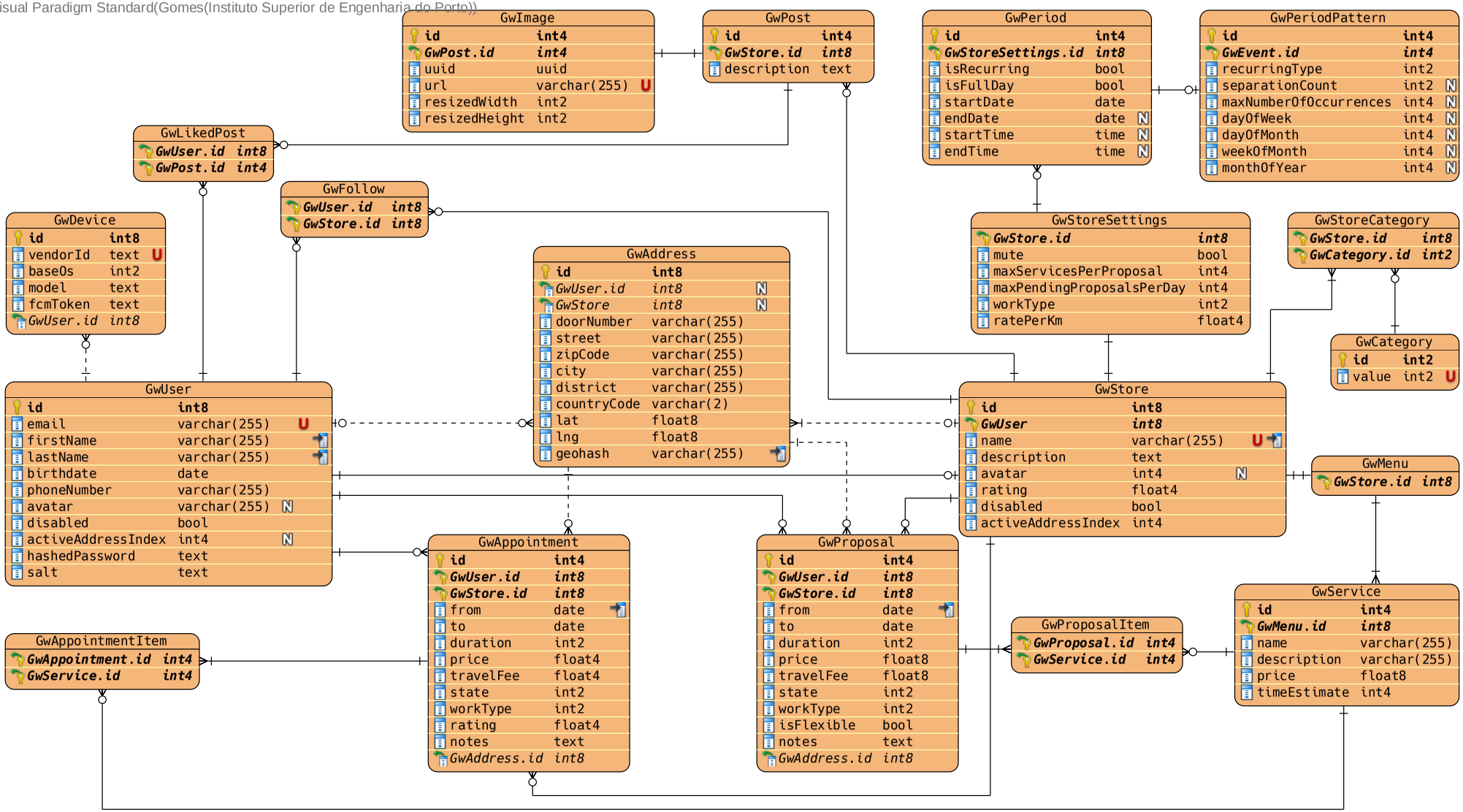

Figure 15: Entity Relationship Diagram

• **GwUser**

The `GwUser` is identified by its *id*. It stores common information for a software service: *email, firstName, lastName, birthdate, phoneNumber, avatar*. The *email* property is unique and what is used, together with a password, as authentication credentials. The *firstName* and *lastName* are non-clustered indexed properties used for the Search Functionality. The *disabled* `boolean` flag allows users to temporarily deactivate their account without removing it from the system, while the *activeAddressIndex* `integer` value is used to mention which address is currently being used as default within the application. The *hashedPassword* and *salt* are properties related to the user authenticity and authentication process to access backend resources. Their purpose is explained with great detailed in the Security section of this document.

• **GwStore**

The `GwStore` is identified by its *id* and its owner ID (*GwUser.id*). The *name* property is a non-clustered index as it will be used frequently to efficiently search a store using its name. *Rating* is the current mean value calculated using all appointments' ratings, it is updated once an appointment finishes and the rating assigned. The *disabled* flag allows to deactivate the store without deleting it, and the mandatory *activeAddressIndex* indicates the current default address being used (its value defaults to 0 when a `GwStore` is created). When creating a store, it is mandatory to provide an `GwAddress`.

• **GwAddress**

An Address is represented by the `GwAddress` entity and both users and stores can have plenty of (To-End cardinality is zero or more [0,*] for the `GwUser`, and one or more [1,*] for the `GwStore`). For the user, having multiple saved addresses exclusively shortcuts in-app operations where it is mandatory to provide an address, by allowing to quickly select one from the list. For the worker, however, the possibility to have multiple addresses is crucial since they might use multiple work locations. One fundamental goal considered

during initial analysis was to provide enough flexibility for the workers to work wherever they prefer by granting them the ability to have multiple addresses simultaneously. It's mandatory for a `GwStore` to have a working address, provided during its creation, as well as for the `GwUser` for **on-location** based services. Users are not required to provide an address during sign-up. The *lat* and *lng* `double` parameters are for rendering map locations in the 2D Map View. Finally, the *geohash* property, a unique hexadecimal value encoded using the latitude and longitude and can be decoded back into that same pair. This property is indexed as it will be used to achieve pagination for search related functionalities, content suggestion and allow for location-based data to be displayed to the user.

• **GwStoreCategory & GwCategory**

A `GwStore` must be associated with at least one and can have multiple `GwCategory`, whose *value* is one of the following categories from `GwCategoryEnum`([6]). A category may also be related to multiple stores. The `GwStoreCategory` entity results from this N – M relationship and connects a single store instance to a category. Assigning a category to a store categorizes it and enables filtered content.

• **GwStoreSettings**

`GwStore` and `GwStoreSettings` have a 1 – 1 relation and are created simultaneously, the latter with the default settings. `GwStoreSettings` is identified by a foreign key, its store primary key, and has settings to allow workers to properly manage their operations:

- *mute* — A `boolean` flag that blocks any user from making new proposals to this store. Defaults to false.
- *maxServicePerProposal* — Maximum number of services the user can pick for each proposal. Defaults to 0, meaning there's no upper limit.
- *maxPendingProposalsPerDay* — The worker might select a max number

[6] [barber, bodyPiercing, hairdresser, makeup, manicure, pedicure, tattooArtist]

of pending proposals per day. Defaults to 0 — no upper limit.

- *workType* — A `GwWorkType` enumerator value with three possibilities:
  - **Fixed**: Worker does not perform on-location work. This is selected as default.
  - **On-Location**: Worker only performs on-location work, meeting the client at their desired location.
  - **Both**: Worker performs both on-location and fixed type work. During the proposal process the client chooses which one he prefers.
- *ratePerKm* — Defaults to 0.0 if *workType* equals `GwWorkType.Fixed`, otherwise it is the value per kilometre the worker requests to meet the client.

• GwPeriod

`GwStoreSettings` may have none to multiple `GwPeriod`s. A `GwPeriod` has multiple properties that define a limited period in time, with an initial and final point in time. It may take place on a particular day, or extend itself for multiple days, e.g: A period between 14:00 and 16:00 for the current day; A period that begins on September 6th, at 12:00, and ends two days later at 16:00.

On **GigWho**, workers are able to define **mute** periods to block proposals whose dates fall between these to enhance their working experience. The properties *startDate* and *endDate* define, each, a date, without time (HH:mm), from when the mute period starts to when it ends. *startTime* and *endTime* `integer` values represent the seconds since 00:00 relative to *startDate* and *endDate*, respectively. The `boolean` *isFullDay* is set if the period consumes an entire day (*startTime* is set to 0 and *endTime* to max number of seconds of a day), and *isRecurring* if the period has a `GwPeriodPattern` associated with.

Non-recurring periods which refer to a past date are removed from the database.

• **GwPeriodPattern**

A `GwPeriodPattern` is assigned to an individual `GwPeriod` to increase its capabilities by extending it with new parameters. With it, **GigWho** is capable of processing multiple instances of a period over time, even though they are bound to specific dates. Periods can repeat daily, weekly, bi-daily, by-weekly, monthly... The *recurringType* has one of the four options: daily, weekly, monthly and yearly, and the *separationCount* is an `integer` value that signifies the deviation between repetitions, ideal for periods that repeat with abnormal recurrence. The *maxNumberOfOccurences* parameter defaults to 0 and sets a limit to how many times the event can occur.

This design is highly influenced by the one from Kher [2016] and is the perfect fit to represent off-days, holidays, work schedule or even unexpected spontaneous events. E.g: The worker wants to mute all proposals for the weekends. The *startDate* would be any day, on a Saturday, with *startTime* set to 0; the *endDate* is set for the next immediate Sunday with an *endTime* set to the max possible value of 86 400[7] seconds. The *recurringType* is set to **weekly** with *separationCount* and *maxNumberOfOccurences* set to **0**. This way, with this configuration, all weekends become muted. A mute period is defined including an entire weekend, and that period is repeated weekly.

The dates and times are separated from each other to facilitate the computation of the multiple same event instances. Mute periods can override each other since they have the same purpose.

• **GwMenu**

A `GwMenu` is created with a store and identified by its store ID. It references a list of services (`GwService`) created by the worker.

• **GwService**

`GwService` represents a service performed by the worker and it is connected to a `GwMenu`. The property *timeEstimate* is an `integer` value provided

[7] 24(hours) * 60(minutes) * 60(seconds)

by the worker for how long the job usually takes to complete.

- **GwProposal & GwAppointment**

`GwProposal` and `GwAppointment` are the objects used between the worker and the client. They are very similar to each other. The proposal has the *proposalState* of type `GwProposalState`[8] that indicates on which side the proposal lies (which user has to provide an answer). The appointment has its own `GwAppointmentState`[9] parameter as well as a *rating* `float` value assigned by the client when the appointment is completed. The *workType* is a `GwWorkType`[10] value.

The reason why most properties are duplicated between the proposal and appointment, instead of the appointment holding a reference (Foreign Key) to a proposal, is because they are meant to be completely independent of one another. The appointment is a long-lived record whereas the proposal is supposed to be a short-lived highly volatile object, always changing and queried multiple times from both the client and the worker. The proposal ends up being removed once an appointment is created, and its properties are carried over. `GwProposal` database table is always kept small (relative to the amount of ongoing proposals) speeding up database lookups. Both entities' primary IDs are composite keys consisting of their own generated ID, the client ID (user requesting for service(s)) and the store ID (the worker), and are indexed by their *from* `date` property due to the fact that they are, usually, queried by it.

- **GwProposalItem & GwAppointmentItem**

`GwProposalItem` and `GwAppointmentItem` represent the services instances selected from the client when creating a proposal.

- **GwImage**

[8] [client, worker]
[9] [scheduled, completed, cancelled]
[10] [fixed, on-location]

A `GwImage` represents an image object uploaded to the Store profile. These images are the backbone of the Feed functionality. It has a *resizedWidth* and a *resizedHeight* `integer` properties for the placeholder asset and a URL for the original asset.

- **GwPost**

The `GwPost` connects the `GwStore` with its `GwImage` objects. It has a *description* property set by the worker and it requires an image.

- **GwLikedPost**

From the N — M relationship between the `GwUser` and `GwPost`, where many users can "Like" multiple posts, results the `GwLikedPost` entity.

- **GwFollow**

From the N — M relationship between the `GwUser` and the `GwStore`, where many users can "follow" multiple stores, results the `GwFollow` entity.

- **GwDevice**

Finally, the `GwDevice` entity is for physical devices the user makes use of to access its **GigWho** account. With each login, not only the credentials are sent but also device related information, such as: *vendorId*, a unique identifier for every device, *baseOs* and *model*; the *fcmToken* is added afterwards, as soon as the user allows push-notifications to be made to that device. If the unique vendor ID is already present in the server, and the associated user is not the one performing the login at the moment, the associated user foreign key is updated. A user is able to login from multiple devices whereas a device can only be associated to one user, which ends up being the one who last logged.

This Entity Relationship model is capable of powering **GigWho** application.

5.6 Flow Diagram

The Flow/Navigation Diagram of **GigWho** presents the flow between main views of the app to properly demonstrate how they relate. Flows with *Drawer* indicate the connected views are only accessible via the main application Sidebar (Drawer), displayed when the user taps the context button or drags the bar from the right edge of the device. Flows with *Tab* (Feed View to Map View or Follow View and vice-versa) denote there's an application Tabbar where it's possible to change back and forth between those views. All flows are reverted by using the application Appbar or the device's back button.

Home is a container of views where it is possible to navigate between each other using a Tabbar. It hosts the **Feed**, **Map** and **Follow View**. **My Proposals** and **My Appointments** split into two other views that render the exact same layout but with different data. Both have one tab for proposals/appointments as clients and another as workers.

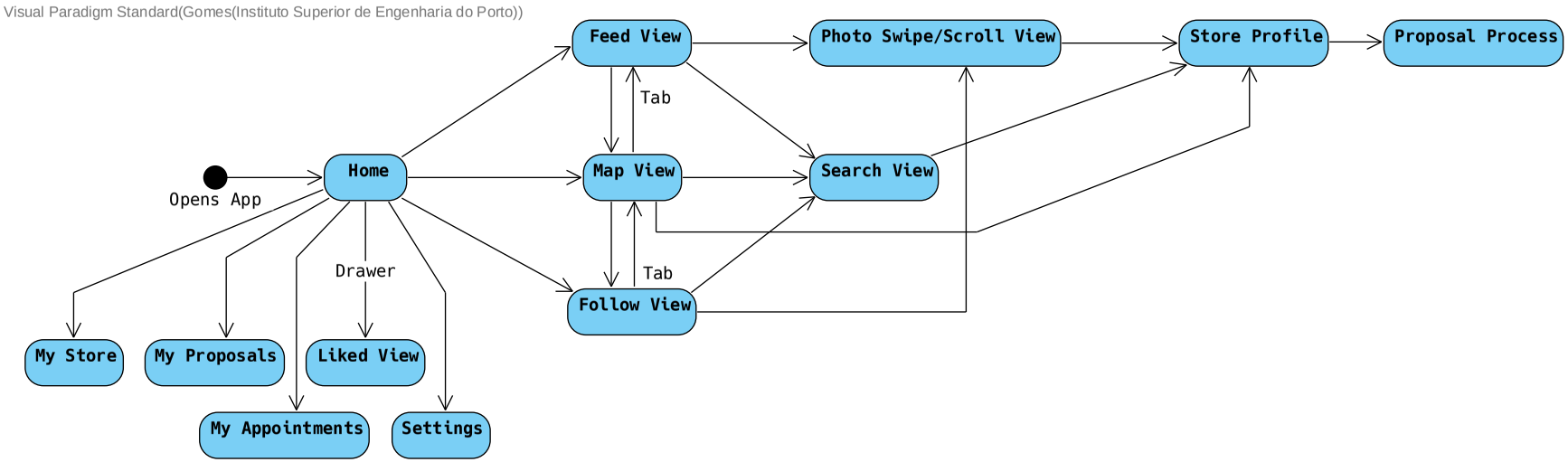

Figure 16: Flow Diagram

6 Implementation

This section covers, through a software engineer point-of-view, the features considered in the Design as well as some other concepts that are fundamental for **GigWho** to work properly and worth mentioning as they are crucial for the development of mobile applications.

6.1 Storage in GigWho

When developing for mobile, it is important to evaluate how the application should use its storage. Users expect a fluid and convenient experience, without password prompts or noticeable loading times. Storage plays a big role when providing such experience, by storing authentication data in a secure encrypted file, or by locally caching resources that would otherwise need to be remotely fetched again and again. This section covers how **GigWho** uses its storage solutions and how they were implemented in a way that, in the future, new implementations can be easily integrated or the current storage usage schema quickly modified.

GigWho has four types of storage:

- **Local Storage** — JSON file-based storage. Allows the applications to create multiple JSON files, each one acting as database for key-value pairs. These files are created and stored locally on the user's device.
- **Firebase Cloud Storage** — Versatile online object storage from Google. Stores user generated content. Integrates customizable access rules and is accessible using the Firebase SDK. **GigWho** utilizes this storage for image related data only.
- **Secure Storage** — Secure Key-value storage. Uses Keychain for iOS and, for Android, it encrypts data using AES, keeping the AES secret key encrypted with RSA and storing the RSA key in Keystore.
- **In-Memory Storage** — Extra fast key-value in-memory storage. Data is stored using device main memory. Differs from the others types of storage in that if the application is closed, the memory is released

and data is lost. This is not a reliable persistence mechanism and is only used for data that is frequently accessed within a specific context.

- **SQLite Storage** — It is used for file cache. While files are stored in a cache directory on the users' smartphone, cache data is maintained in a SQLite local database. Although this is a storage strategy, it requires prior configuration to set up the data model with table and its properties. However, in **GigWho**, this storage is only used to keep cache data (`GwCachedResource` entity) and requires almost no configuration.

Different app components utilize different storage types. Whereas a component might write data to the secure storage, another might use default local storage. **GigWho** abstracts storages through the `GwStorage` interface by using the **Strategy** pattern, a design pattern that implies the abstraction of similar behaviour with different implementations. All storage implementations share the same behaviour however, internally, are implemented differently.

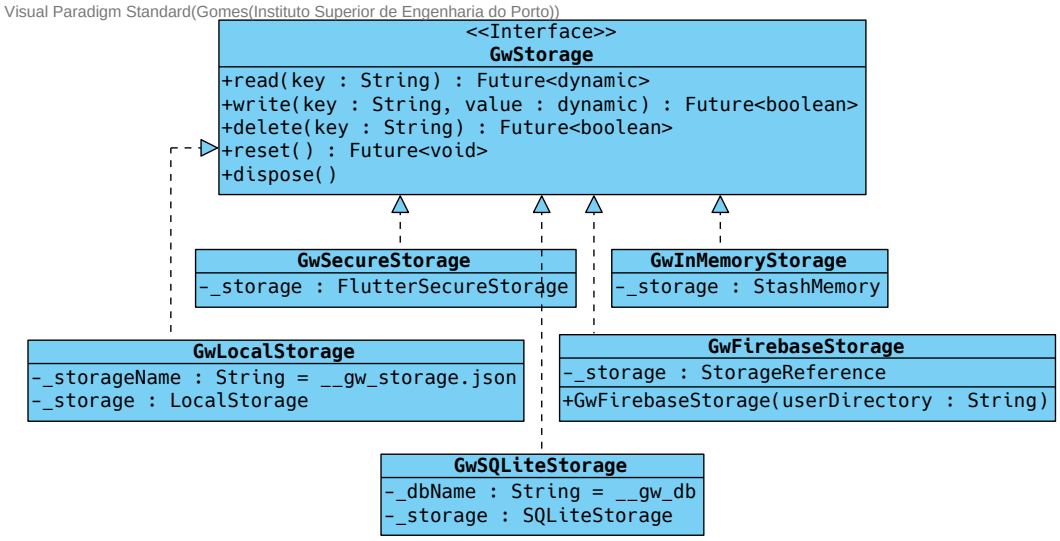

Figure 17: Storage Strategy Pattern

Each storage is a class with concrete behaviour. Components utilize Storages via composition.

Initializing `GwAppSettings` and providing the desired Storage mechanism:

```
1    final appSettings = GwAppSettings(GwSecureStorage());
```

Listing 1: Initializing GwAppSettings

If, for whatever reason, during runtime, `GwAppSettings` must use another implementation, it can be achieved by invoking the setter `setStorate()`:

```
1    appSettings.setStorate(GwInMemoryStorage());
```

Listing 2: Setting new storage during runtime

All stored data can be removed from outside the app, in the device settings.

6.2 Implementing Cache

Caching is a mandatory technique for most software nowadays for improving performance by dismissing unnecessary tasks and making software more responsive. Downloading files is a costly operation, requires CPU power, memory and bandwidth, and it is counter-productive if those resources are exactly the same from the last time the client requested. This becomes much more relevant with image files where, once uploaded, are unlikely to change during its lifetime. Taking Instagram or Facebook as an example, most people edit their photos (resize, cut, filters) prior to the upload, and the same can be expected with **GigWho**.

GigWho implements local cache, storing files in a dedicated cache directory, that can easily be deleted, and all cache control data in a local database using SQLite storage. It uses the Flutter Cache Manager package to properly handle cached resources. The following diagram is the schema of the data properties needed for caching:

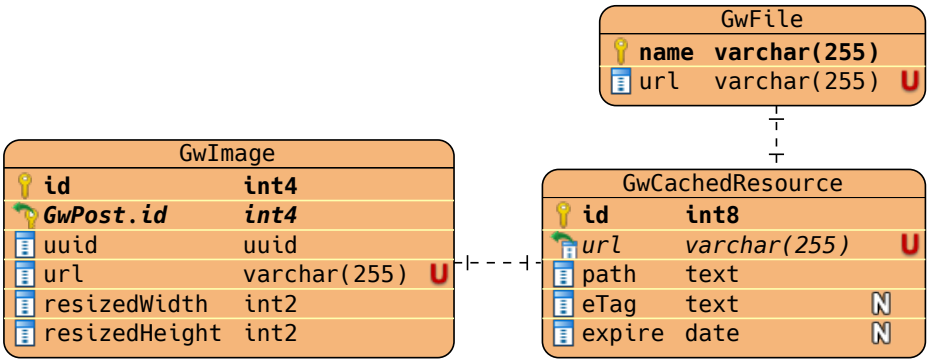

Figure 18: GwCachedResource

The `GwCachedResource` relates to `GwImage` and `GwFile` through their URL property. Neither `GwFile` nor `GwCachedResource` are server stored entities. A `GwFile` represents a text file created from a JSON payload fetched from a GET request to the **GigWho** server and is stored using **GigWho** Local Storage. The `GwCachedResource` is stored using **GigWho** SQLite Storage.

Text File Caching

GigWho caches certain response payloads such as the ones provided by the `api/places` API. Before requesting external data, **GigWho** uses the URL to check if there's a cached copy of the data to be immediately displayed to the user. Regardless if there's cached data or not, the server request is always made. Caching JSON payloads is relevant for scenarios where connected networks perform at incredible low speeds, to scenarios where there's no network at all, guaranteeing the user has always something to visualize while waiting for the most updated version from the server. It becomes even more relevant for the Map View where the user can still check store locations without internet. The main goal is to always have data being shown to the users while the app requests its latest versions to the server, regardless of the time it takes.

Image Caching

A cached resource has an *expire* date and an *eTag*[11]. These are only used when the cached resource is related to an `GwImage` and are provided by the Firebase Cloud Server the first time the resource is requested. Both values are appended when making that same request a second time.

For images the request is always sent to the server but a second download is not imperative. The server compares the resource *eTag* value, computed when the image was first inserted, to the one present in the request made by the app. If they match, it replies with a *304 — Not Modified*; if they don't, the latest version of the resource is downloaded. The latter is extremely unlikely since **GigWho** does not currently support modifications. The server also checks for the *expire* date property. If it is before the current date, the local copy of resource is invalidated and the latest version of the resource is downloaded, regardless of the *eTag* value. T he server also provides the most recent *eTag* value and a new expiration date, which are updated on the `GwCachedResource` with that URL. This is extremely relevant for the Feed View and Follow View that mostly deal with image files.

[11] Md5 hashed value used to verify versions of a resource.

The following two diagrams expose the role of local cache in **GigWho**. On the first, the user opens the Map View and cached data is rendered before connecting to the server. On the second, the remote version of the file is the same to the local one, according to its *eTag*, and the client uses the cached version. The 1.5 message, in the second scenario, has the *eTag* and the *expire* values appended to specific HTTP request headers.

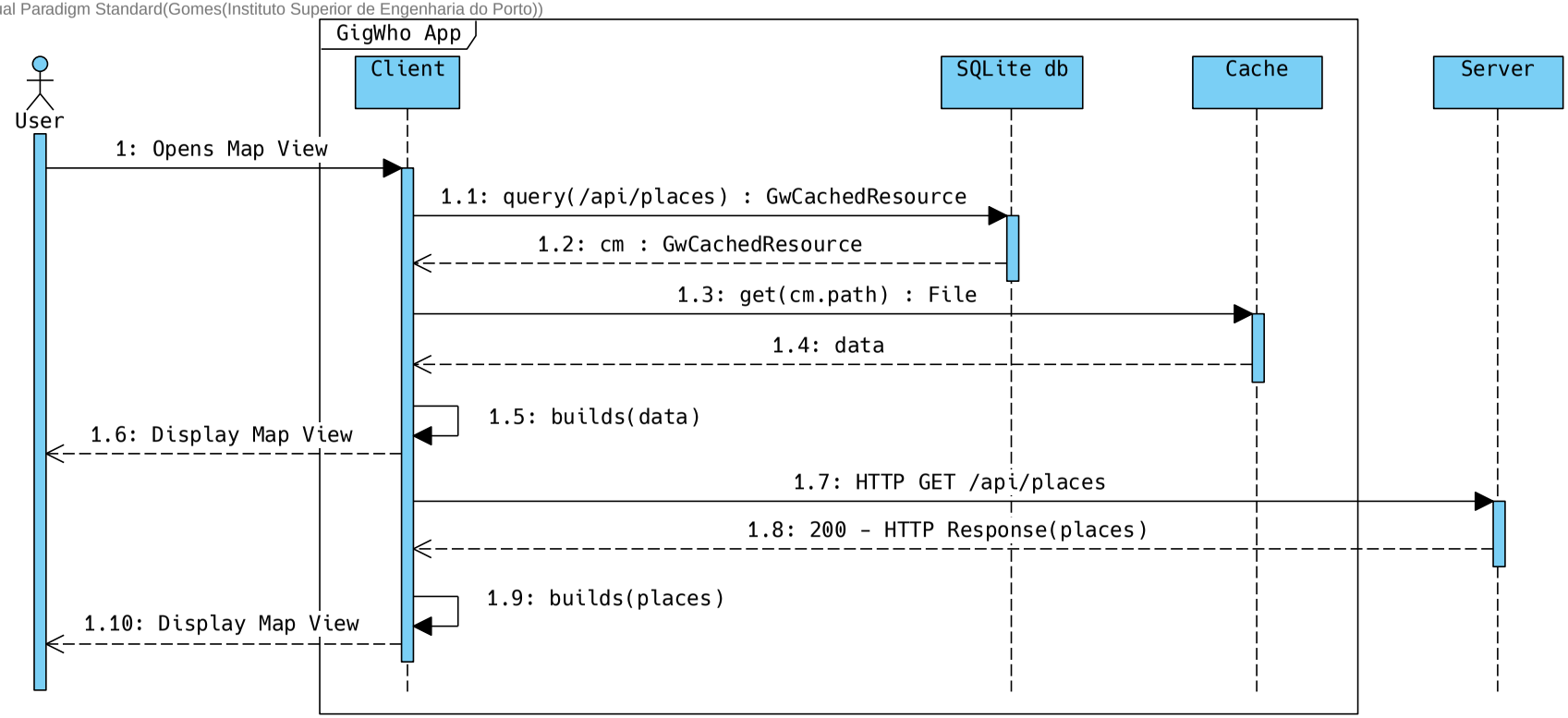

Figure 19: Using cached requested data — Map View

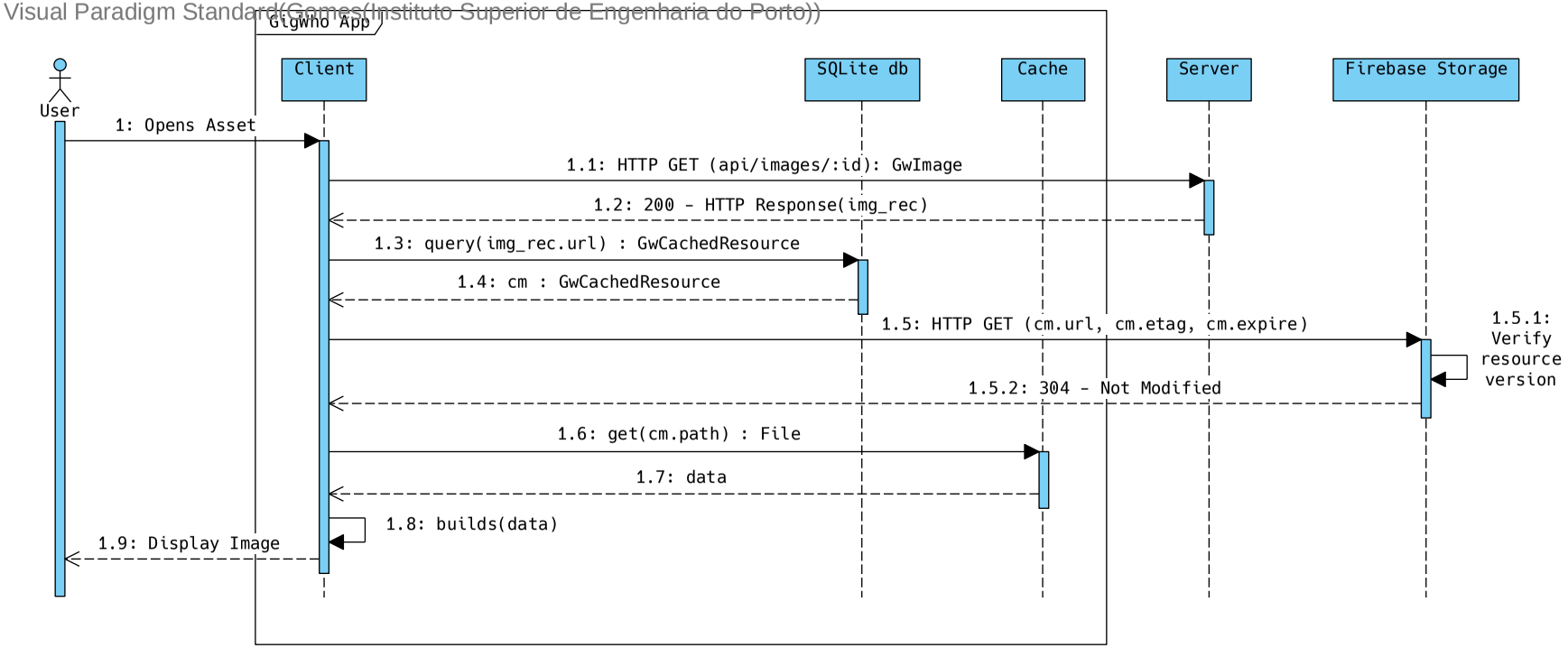

Figure 20: Using cached data to display an image to the user.

6.3 GigWho Media Assets & FCS (Firebase Cloud Storage)

This section explains how assets are imported from the local gallery and uploaded to the remote server (Firebase Cloud Storage) as well as stored in the **GigWho** server to be accessed by every app user. Assets in **GigWho** are imported locally, from the users' smartphone gallery to the app, and uploaded to a bucket on Firebase Cloud Storage. A Bucket is a remote container provided by Firebase where developers can store their application objects.

In **GigWho** storing an image asset into a bucket creates a new smaller version with specific dimensions. The resized copy is used as low quality placeholder (image preview) as the app downloads the original and larger file achieving a more responsive design. This is common practice for image oriented apps (e.g.: Instagram).

6.3.1 Import Asset to the App

The **Image** and **Image Picker** packages were used to simplify the import process from Android and iOS galleries. When users start the upload activity, the app first opens the filesystem for file selection. File raw data is imported and used to generate its placeholder equivalent. The following is an excerpt detailing the entire process, using both mentioned packages:

```
1  const maxDimValue = 400;
2  final ImagePicker _picker = ImagePicker();
3  File _image;
4  int _width;
5  int _height;
6
7  Future _importImageFromGallery() async {
8    final PickedFile file =
9    await _picker.getImage(source: ImageSource.gallery);
10
11    if (file != null) {
12     // Update widget state.
13     setState(() {
14       this._image = File(file.path);
15       Image original = _image.decodeImage(this._file.
             readAsBytesSync());
16       Image resize;
17       if (original.width > maxDimValue || original.height >
             maxDimValue) {
18         resize = (original.width > original.height) ?
19           copyResize(original, width: maxDimValue) :
20           copyResize(original, height: maxDimValue);
21         this._width = resize.width;
22         this._height = resize.height;
23       }
24     });
25   }
26 }
```

Listing 3: Importing from Adroid/iOS gallery

The `ImagePicker` API provides file selection from the local gallery and `PickedFile` is a simplified agnostic wrapper of `File` (from `dart:io`) that holds raw data (`Uint8List`). In this excerpt, the path for the selected file is provided to an actual `File` object to be fed to a widget that, ultimately, renders the actual image. The APIs `decodeImage()` and `copyResize()` are used to generate shorter dimensions of the asset while preserving its original proportions.

6.3.2 Upload to remote

With the displayed picture on the device, the user decides whether to upload it. If so, the Firebase Storage provides the required APIs to connect to the

remote Firebase Cloud Storage and securely upload the selected file. The end result is a URL pointing to the image remote location. The excerpt is a continuation from the previous one and displays the process of uploading to the remote site:

```
GwStorage _storage = GwFirebaseStorage();

Future<void> _uploadToRemote() async {
  final folder = this.widget.storeId;
  final uuid = Uuid().v4();
  final assetLocation = '$folder/$uuid';
  final List<int> byteData = await this._file.readAsBytes();

  try {
    final bool res = await this._storage.write(assetLocation,
        byteData,
      resizeWidth: this._width, resizeHeight: this._height);
    if (res) {
      final locationUrl = await this._storage.read(assetLocation);
      final image = {
        'uuid': uuid,
        'storeId': this.widget.storeId,
        'imageUrl': locationUrl,
        'resizeWidth': this._width,
        'resizeHeight': this._height,
      };
      context.bloc().add(PostImageEvent(image));
    } else {
      // Display Error Modal.
      ...
    }
  } catch (UploadTaskError err) {
    // Display Error Modal.
    ...
  }
}
```

Listing 4: Upload asset to Firebase Storage

The `GwStorage` implementation is defined at runtime, through the Strategy pattern, and the strategy applied is `GwFirebaseStorage`, so it uses the Firebase SDK to connect to the remote storage, asynchronously. It first uploads the asset to `https://$host/$storeFolder/$assetName` and

immediately retrieves the generated URL from the server. It then creates an `GwImage` object storing the URL, a name (UUID), the store ID of the current user and the resize dimensions used to load the placeholder. Every store has assigned a folder equals to its ID, inside the **GigWho** bucket, to keep each store folder separated from one another. Only the user who uploads the object has write access over it.

6.3.3 Generating the low-res version

Firebase Cloud Storage provides extensions that can be deployed directly into a running storage instance. One in particular is the **Resize Images** extension capable of resizing image files, thus allowing **GigWho** to create different sized versions of the original asset. When first uploading to the server the resize dimensions are sent as custom metadata with the image raw data. The resize operation happens remotely as soon as the image is uploaded, the result is stored in the exact same location, with the resize dimensions appended to its name. In the first excerpt, the new dimensions are computed from the original file. The resize version can only have its greater dimension equal to 400 pixels while maintaining the aspect ratio. The process ensures that it is always 400 pixels wide or 400 pixels long.

The following excerpt shows the URLs of both the original and the resize version:

```
1  final storeId = this.widget.storeId;
2  final assetName = '54e4d480-71ab-4b42-a08c-2e662b20888a';
3  String original = 'gs://gigwho.appspot.com/$storeFolder/$assetName.
       jpg';
4  String resize = 'gs://gigwho.appspot.com/$storeFolder/
       $assetName400x400.jpg'
```

Listing 5: Original and resize version URL

The **GigWho** app starts by fetching image data (`GwImage`) from the server and use its URL to check if there is a cached version. If not, the app starts by downloading the low-res version by appending to its URL the resize dimensions from the `GwImage` object. The download of the low-res asset is faster, since it has a much smaller size, improving users' perception

of the app performance. However, the lower resolution asset only acts as a preview and as soon as the first download completes, it immediately queues the download of the original one.

The following diagrams help visualize both the upload and the download, when an image is shown to the user. The latter assumes there is no cached copy of asset (Implementing Cache).

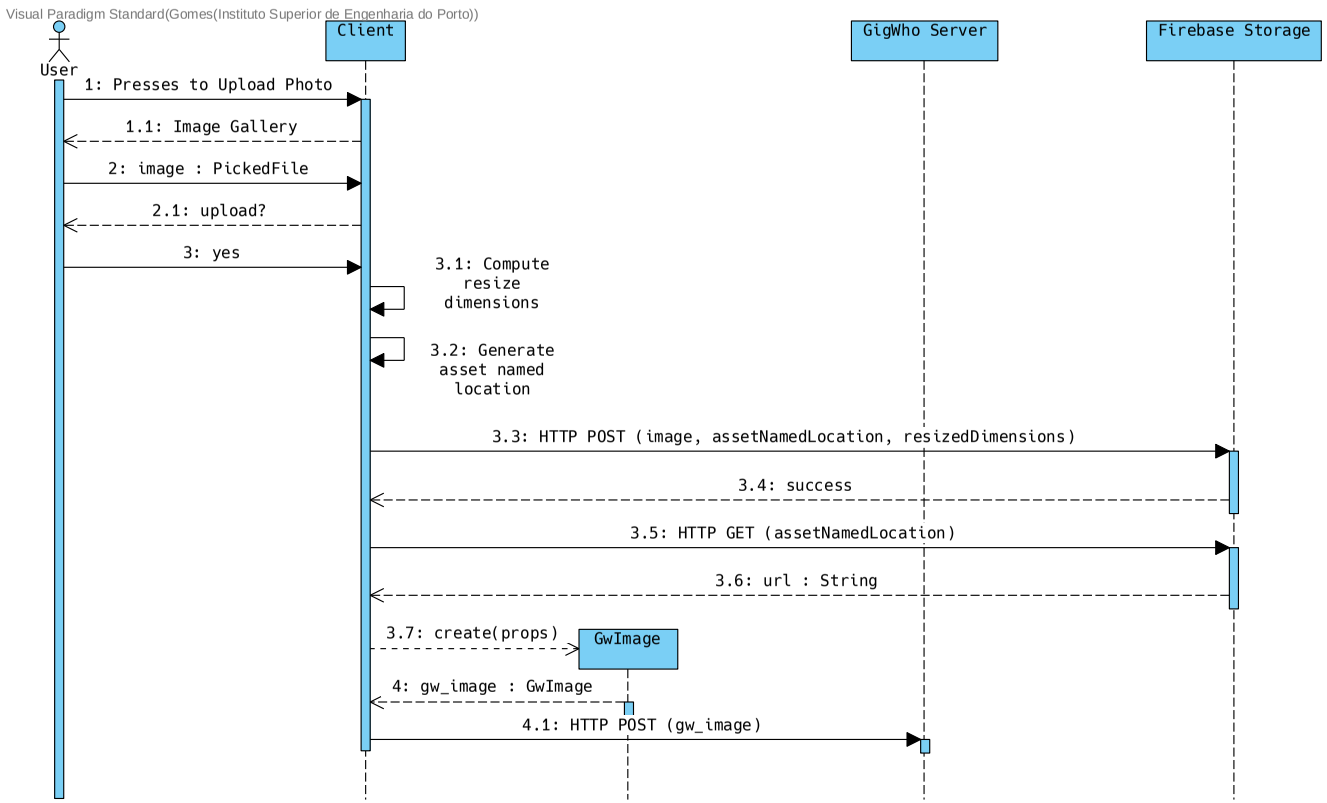

Figure 21: Upload Image diagram simplified

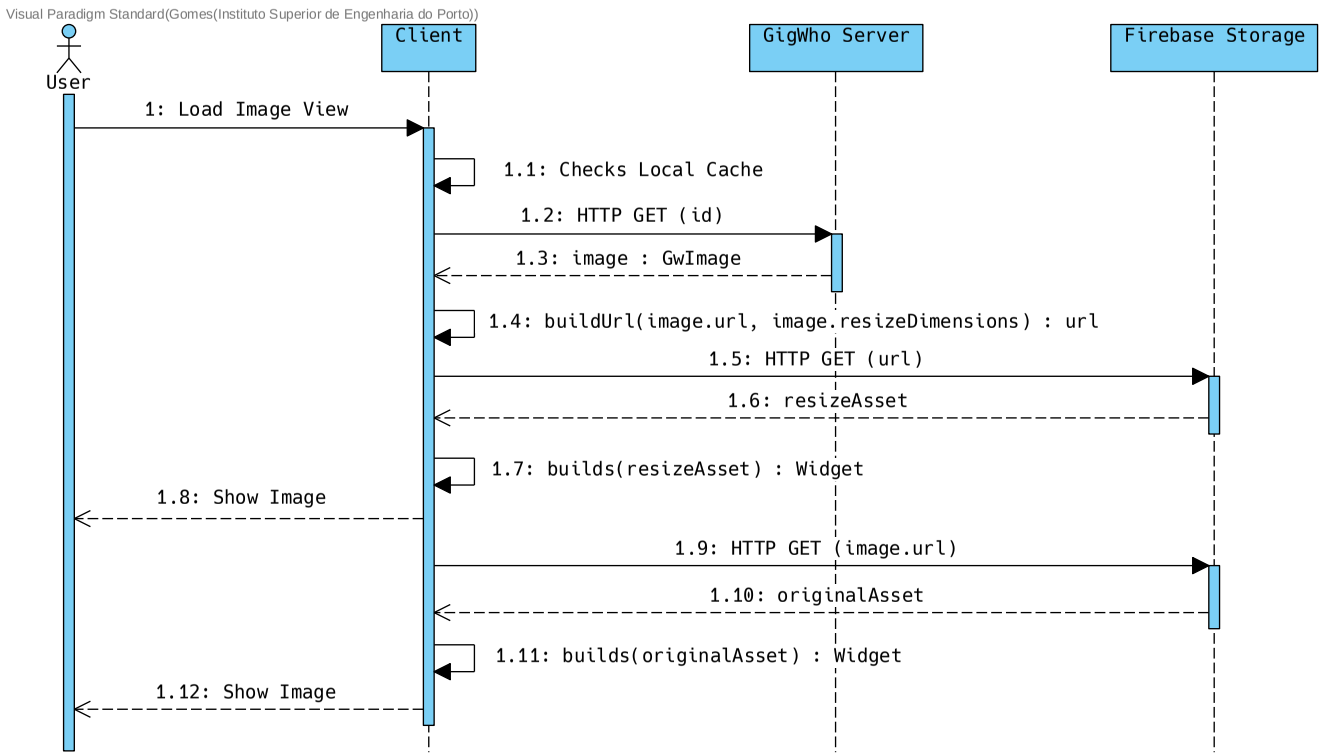

Figure 22: Download Image diagram simplified

6.4 Security

6.4.1 Authentication & Authorization

GigWho server has implemented an Authentication & Authorization service following OAuth 2.0 standards. Only authenticated users with a valid username and password combo are given access. In **GigWho** the username is the email address. It uses the **Password Authorization** grant, where users are given access tokens when providing a valid email and password combination on log in. The token provides verified access without having to authenticate the user for every access request. It also implements the **Refresh Token** grant flow. For each access token there's also a refresh token issued and given to the client, after logging-in, alongside an expiration date. The expiration date pinpoints the validity of the access token and when expired, the client uses the refresh token to generate a new one.

OAuth 2.0 also introduces Clients. Clients are the applications the users utilize to access the server, which are, in this project, exclusively the **GigWho** mobile app, not to be mistaken for the client business definition attributed in previous sections. Clients also require to be validated and therefore recognized by the server, similar to the User. **GigWho** app client requires an identifier and a secret. This data is added to the running server application via CLI. Users are never aware of the client identifier nor secret and both are kept encrypted inside *env* files on the **GigWho** application.

The following diagram provides a clear view on how the log-in is conducted, using the **Password** authorization grant:

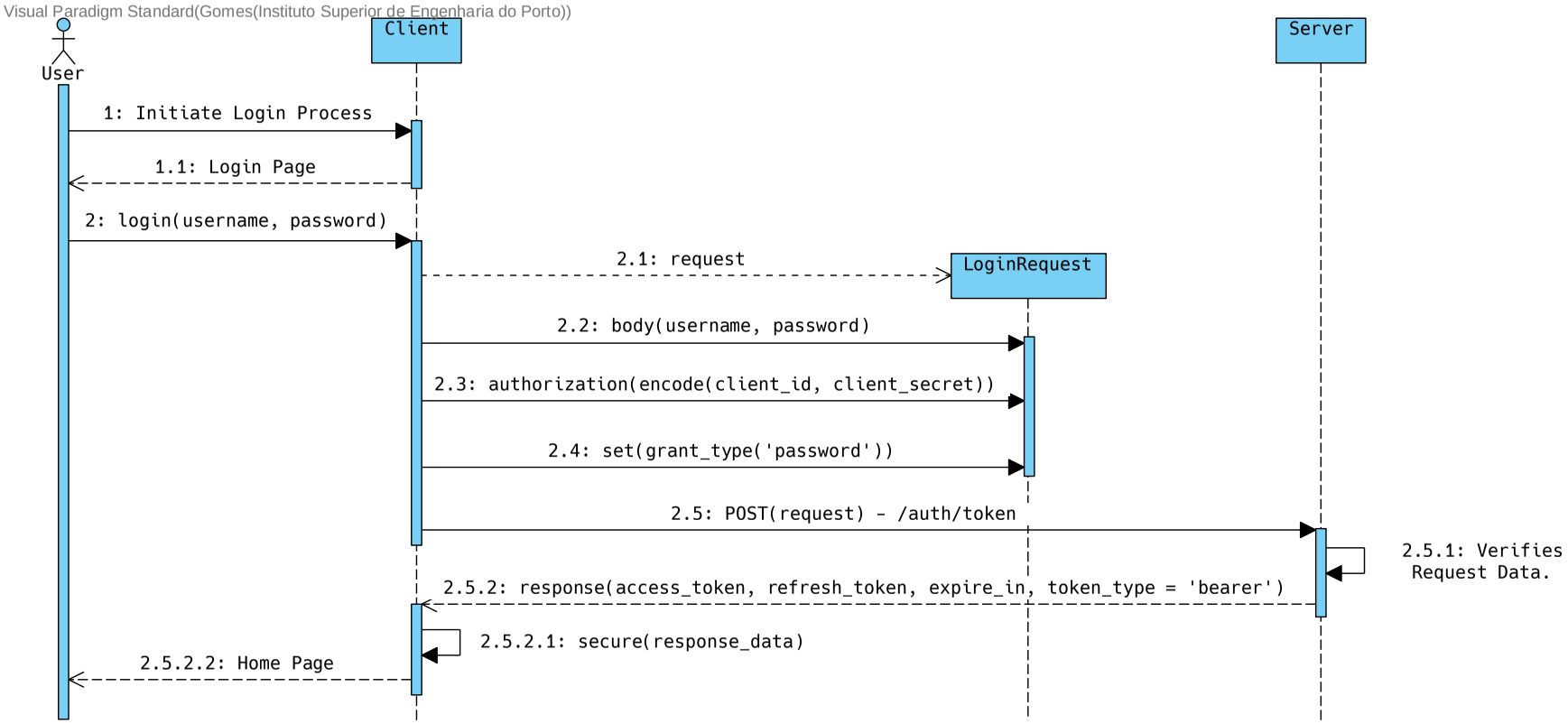

Figure 23: Login Sequence simplified

To obtain a token, users have to provide their credentials to the `/auth/token` API. When doing so, both the client ID and the client secret are decrypted, encoded[12] and added to the *Authorization* header of the Login request. Appended to the body is also the grant type used, *password*. The server replies with a valid access token, an expiration limit, a refresh token and the access token type. The refresh token is for the OAuth2.0 **Refresh grant**. The client keeps track of the expiration limit that, once reached, automatically sends a new request to the same API, this time using only the refresh token and setting the grant type value to *refresh*. The server replies with a new valid access token and the client does not have to log in again to vindicate another.

Token data is always stored encrypted using `GwSecureStorage` on the client application. This storage uses platform specific native secure solutions.

When a user signs up, a `GwUser` object is created saving the email, a salt (random generated value) and a hashed version of the password. From this point onwards the user can log in and start using the app. When the server receives a login request, it computes the hash of the password in the request by mixing it with *salt* property generated during sign up and using an SHA-256 cryptographic function to compare the value to the one stored in the `GwUser` database table. All authorization and authentication data is kept under three database entities:`GwAuthClient`, `GwUser` and `GwAuthToken`.

[12] **Base64** encoded in the following format — *(client_id:client_secret)*

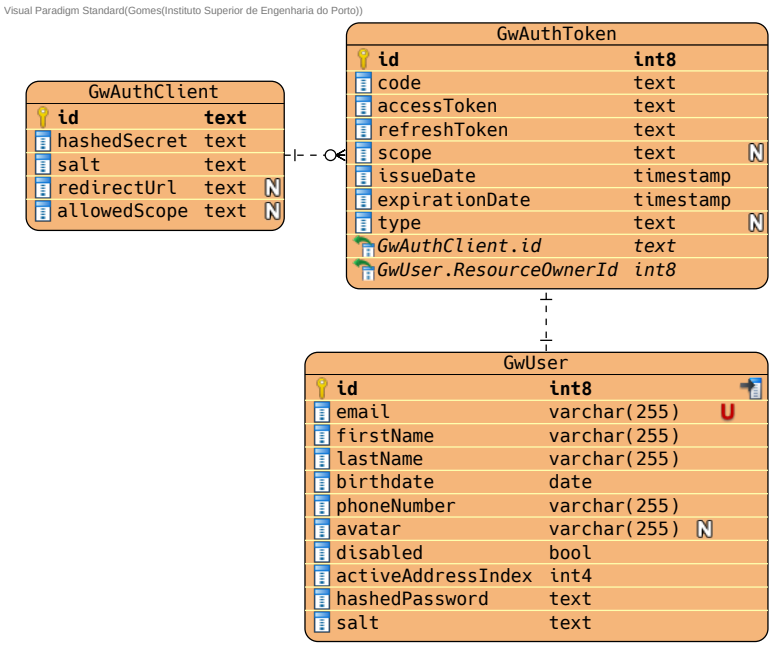

Figure 24: Authentication related Entity Relationship Diagram

Access to resources is also controlled via scopes. A scope is a piece of information that indicates what the client has access to. Since there's only one app and this project does not include the implementation of a back-office for administrators or any other role, the **GigWho** app is the only client and has access to every resource. However, this opens up the possibility to implement more clients, with different access levels, without having to further implement more auth related code, since the server already supports the addition of new clients and access scopes.

6.4.2 Input Validation

Input validation is a major security feature that all frontend services should and must implement to avoid corrupted inputs from reaching the backend server. In the **GigWho** app, every non-restricted value form, such as input fields, gets validated while the user is typing. Using a combination of *RegExp* (Regular Expressions) with enforced input text masks, users become unable

to type invalid inputs let alone submit them.

Input Text Mask Formatters have the two properties mentioned above: a *mask* and a *filter*. The mask establishes the text format whereas the content is enforced by the filter. For a Portuguese phone number input the following would be used:

```
1  String mask = '+351 9# ### ## ##';
2  var filter = {
3    '#': RegExp(r'[0-9]'),
4  };
5  var maskFormatter =
6    MaskTextInputFormatter(mask: mask, filter: filter);
```

Listing 6: MaskTextInputFormatter - Dart code sample

The filter provides meaning for the mask # that can match any numeric character between 0 and 9. The `maskFormatter` is fed to the text input field that re-evaluates its input text on every change.

6.5 Feed View

The Feed View is the main view of **GigWho** and the view users are faced with as soon as they open the app. It mainly consists of images. At the top there's a category filter, which the user can pick apply to select what is shown. The bottom section has the tab bar, providing access to other app compartments (Map View and Follow View), and, on top if it, the search and drawer buttons.

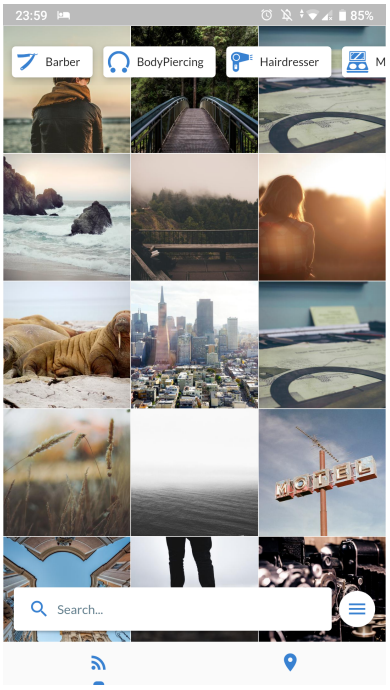

Figure 25: Feed View

Image are disposed in a grid arrangement with a 3 length column and an x length row. The number of rows is defined by the user's current y position on the view. It has implemented an Infinite Scroller technique where only a limited set of results is first loaded until the user reaches a certain *y-axis* position, by scrolling, triggering the app into loading the next set and so on. Consecutive sets are appended to the grid. Users can also force refresh by

pushing down the top of the grid, clearing the grid results and loading a new set.

Infinite Scroller

Infinite Scroller is one of the techniques that improve the app actual performance and leads to a more responsive UI, keeping the users engaged. It's about loading a massive set of objects in batches instead of loading all at once. For mobile, fetching data from external sources has a relying cost on the network quality the user is connected to. The Feed View implies the use of many image resources that need to be downloaded from the server (if not already cached). For each set of resources there's an equal number of *widgets* that process and render it to the user, and having less consecutive renders is more performant than having more. With infinite scroller, resource objects are loaded in smaller sets and only when they are actually required.

This technique requires modifications not only in the frontend but also in the backend. `GwImage` objects are fetched to populate the Feed View, from the most recent to the oldest. For each set, the **GigWho** frontend also requests the *createdAt* `int` timestamp property of each object in the set. These values are stored locally, in-memory, and, when making succeeding requests, the older value is added to the request body, alongside the *pageCount*. The *pageCount* is an `integer` value to limit the number of results to return and it defaults to 30 is not specified. The **GigWho** server expects a *timestamp* property in the request body, for requests to `api/feed`. If there's no value (undefined) it always returns a set with length equal to *pageCount* or the firs thirty if this is also undefined.

In the frontend, the Feed View wraps all the resources in a Scroll View and appends a `ScrollController` object, an API to provide access to the current y position and the current max extent ($ymax$) of the scroll view it is attached to. Once the user reaches a certain y position, it triggers the app into making a new request. For every new set of results added to the view, the vertical extent of the scroll view is updated.

```
1  const thresholdFactor = 0.6;
2  void scrollEvent() {
3    if (yCurrent >= ymax * thresholdFactor) {
4      request.body.append({
5        'timestamp': <x>,
6        'pageCount': 20,
7         filter: [],
8      });
9      context.bloc<FeedBloc>().add(FeedRequest(request));
10   }
11 }
```

Listing 7: Inifinite Scroller simplified sample

The print-screen provided has random images, fetched from the random image API and not related to **GigWho**.

All results are cached locally, as they are not expected to change. This technique is also implemented in the exact same way in the Follow and Search Views.

Category Filtering

Users are able to pick category filters and apply them to both Feed View and Map View. When a filter is applied, **GigWho** app resets the current resource set and asks the server for a new one with only categorized data, without requiring the *timestamp* value. Therefore, for the same endpoint (`api/feed`), the server also checks for a *filter* property in the request body, a list of `GwCategoryEnum` values. Even though resource set is removed, **GigWho** is caching image data.

Force refreshing resets the current set and retrieves data from the latest point available.

6.6 Map View

The Map View renders store locations on a 2D view of the map, with latitude and longitude coordinates. It organizes stores into clusters based on the distance between themselves and the zoom level applied by the user.

The view supports panning and zooming movements. Clusters and store locations are stored in a tree data structure. The nodes have `integer` values equal to the amount of stores it has whereas leaf nodes represent store locations, with no assigned `integer` value. Stores must respect a certain distance in between each other to form a cluster. If the zoom level increases, the clusters break into smaller ones and/or reveal actual store locations. If the zoom level decreases, clusters are formed by smaller clusters and/or store locations.

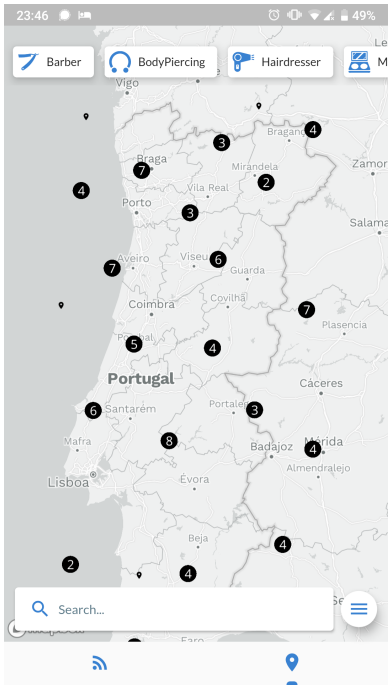

Figure 26: 1 — Map View

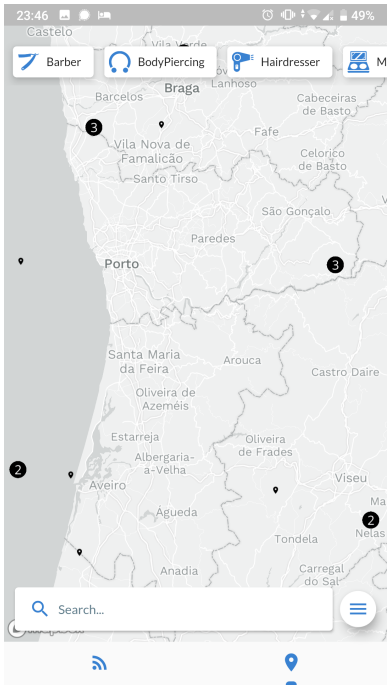

Figure 27: 2 — Map View

Both prints have random generated locations with random latitude and longitude values.

Store locations are fetched from the `/api/places` API all at once and cached locally (process already explained in the Text File caching section). The server replies with an array of `GwPlace` objects, a compact data structure that holds the required data to properly render the location and redirect users to each individual profile. The **GigWho** server queries each `GwStore` and its `GwAddress` through a join operation to assemble the required `GwPlace`.

```
final query = Query<GwStore>(dbContext)
  ..returningProperties((s) => [s.id, s.name])
  ..join(set: (s) => s.categoryList)
  ..join(object: (s) => s.address)
  ..returningProperties((a) => [a.lat, a.lng]);

query.exec();
```

Listing 8: GigWho Server query for GwPlace - Dart code sample.

Clustering is achieved using the Fluster package, a geospatial point clustering library for the Dart ecosystem.

6.7 Follow View

The follow view displays images from the stores the user chooses to follow. Unlike the Feed View this one disposes images in a top to bottom scroll view. Images are laid out chronologically, with the most recent ones on top. At the bottom of each image component there's a description and the action bar for actions such as *Like*, *Share* and visit the store.

Results are fetched from the `/api/content_follow/` endpoint. The server knows which user is making the request through its access token and queries data specific to it. Like other views in **GigWho** it implements an infinite scroller technique, fetching a small collection of images and appending more, as it scrolls down. Force refresh is also available for this view, by pushing down the top of the scroll view.

6.8 Search View

The Search View allows users to search for stores with specific terms that are either present in the store name, the name of the store owner, or from a specific place. Like the Feed and the Follow View, it implements infinite scroller, loading more results as the user scrolls down.

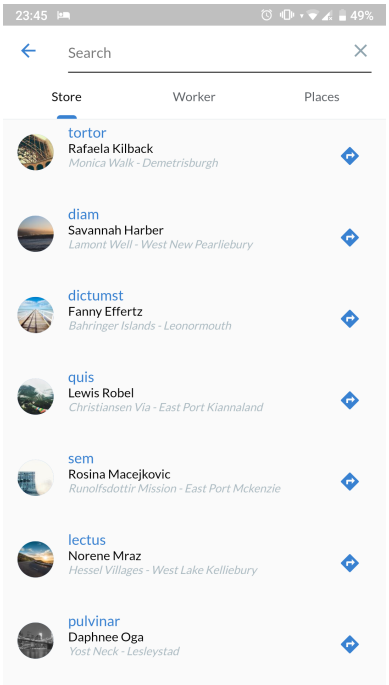

Figure 28: Search View

The view is divided into three tabs that share the same search bar, however, pressing the virtual keyboard search button on each tab requests data from different API endpoints:

- **Store**: Requests to the `api/search?param=store` API and appends the search term to the request body. The server provides sorted results that contain the search term.
- **Worker**: Same as the **Store**, but requests are forwarded to `api/search?`

`param=worker`.

- **Places**: Searches for this tab are processed before being sent to the **GigWho** server. When the user submits, the search term is first converted into latitude and longitude coordinates in a process called geocoding, using the Geocoding package. It can convert address data into coordinates, and vice-versa, using the services provided by both Android[13] and iOS[14] platforms. With a single `String` input, the Geocoding API might return single or multiple results. If there's multiple, the app computes the distance between the user and each result and saves the closest one. Finally, it generates a *geohash* encoded hexadecimal `String` from the stored coordinates and sends it to the `api/search?param=places` API.

The following diagram provides a clear view on the process of requesting search results for the **Places** tab:

[13]https://developer.android.com/reference/android/location/Geocoder
[14]https://developer.apple.com/documentation/corelocation/clgeocoder

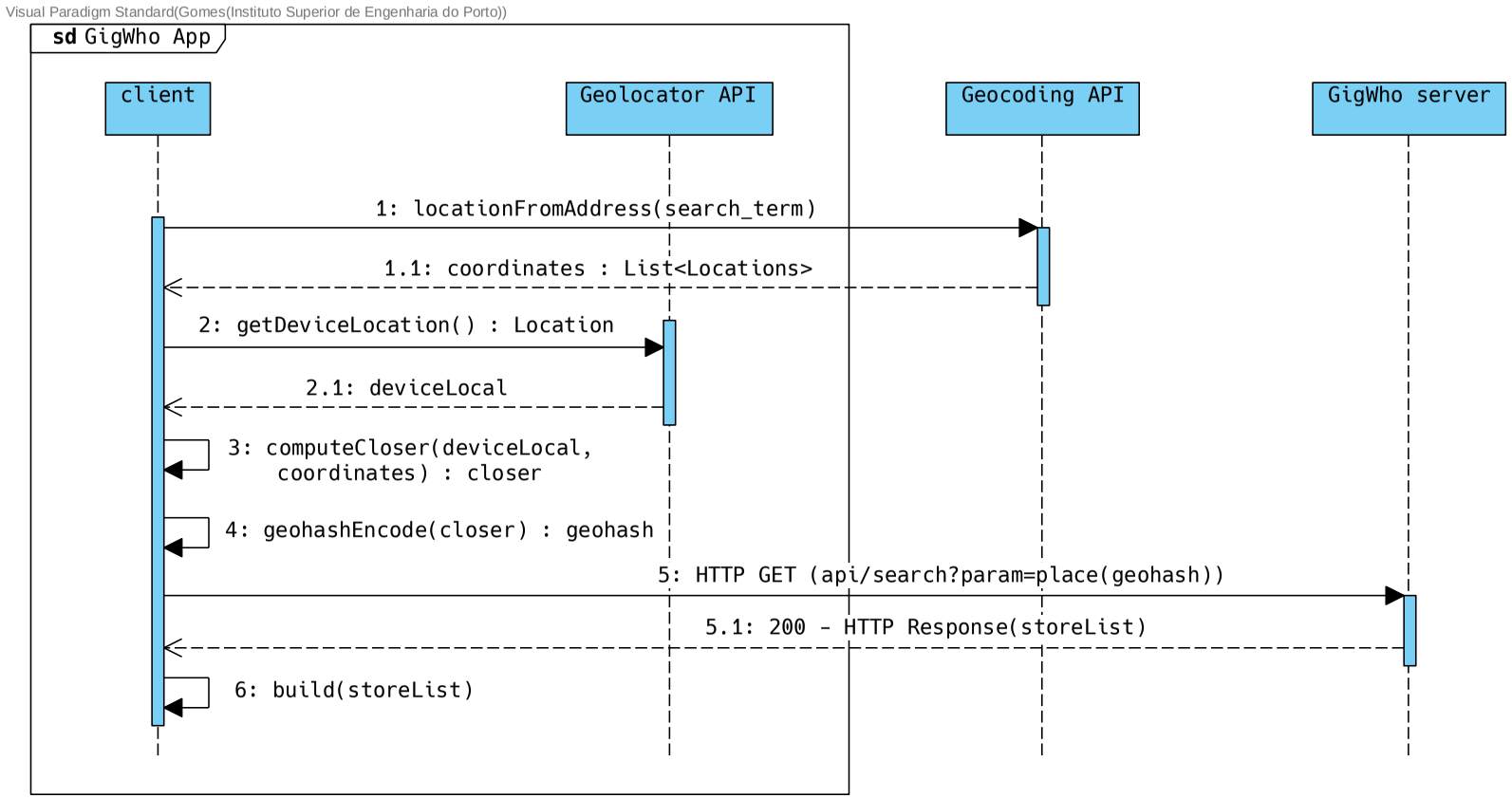

Figure 29: Search by Places diagram

- **1** — From the search term provided, it utilizes the aforementioned Geocoding API to obtain valid coordinates.
- **2** — The client uses the local Geolocator API to get the current device coordinates.
- **3** — From the list of coordinates returned in step **1.1**, it calculates which one is closer to the user device.
- **4** — Encodes the closest pair of coordinates into a *geohash* value. More information about the encoding can be found here[15], although the most relevant factor for **GigWho** is the ability to encode positions relative to one another. This means the closer locations to each other, the more alike is the hash between them, making it the perfect tool to find closer locations within a certain radius. This is possible thanks to the hash also encoding a level of accuracy.
- **5** — Requesting store data to the **GigWho** server that queries for addresses containing the provided *geohash* value.
- **6** —**GigWho** apps builds the store results widgets and shows them to the user.

Every time the user triggers the app to load more results, by reaching a certain y-position on the scroll view, the last character of the *geohash* is removed, decreasing its accuracy thus increasing the search area enclosed by the hash. A new request (step **5**) is made for a wider set of store locations.

For pagination, however, **GigWho** goes a step further and instead of fetching all store locations within the area expressed by the *geohash* accuracy level, it appends a *fetchLimit* `integer` value to the request body and only when the quantity of the store set is inferior to this value, does the app remove the *geohash* last character. This way, even if there's 100 store locations for the first *geohash* value, they are all split into four (4) different requests, decreasing even further the load on the backend.

[15] https://sites.google.com/site/geohexdocs/

6.9 Client-Worker communication

The proposal process has three stages, as already mentioned in the Design section. On the first two, the client chooses the services and the location.

Figure 30: 1st Stage — Menu Service View

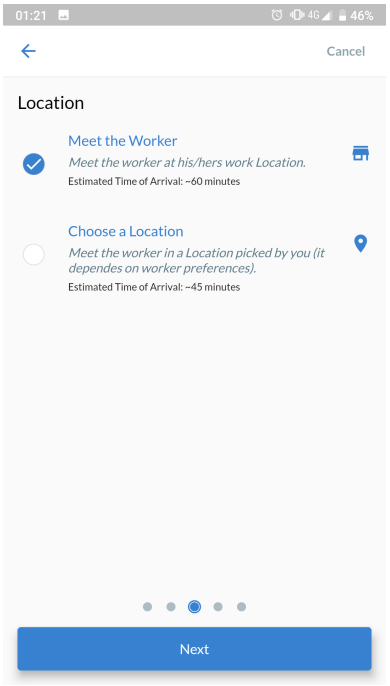

Figure 31: 2nd Stage — Location View

From this point, the minimum time required for the appointment is equal to the summation of the selected services. If the client had instead selected *on-location* type work, the total time would also include an estimate travel time, from the worker's current default main address to the client's provided location.

On the third stage the client is required to pick a date, without time. This is the most crucial process when creating a proposal since the app needs to build efficient slots for the client.

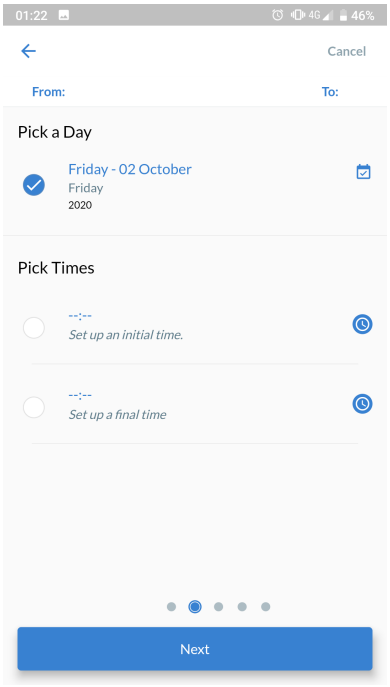

Figure 32: Pick date

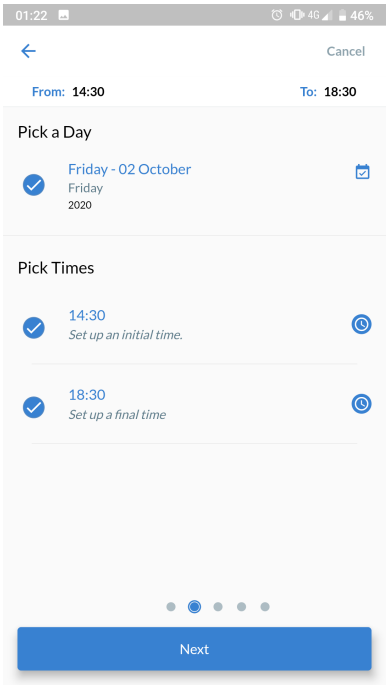

Figure 33: Pick slot

After picking the date, the app proceeds to gather all `GwAppointment`, `GwPeriod` and `GwPeriodPattern` required to build the appropriate slots and display them to the user. `GwPeriod` data is first requested since it must always be accounted for, whereas `GwAppointment` data will not be needed if there are already mute periods for the same time window.

To properly manage empty slots the app makes a request to the server given the selected date, the min duration required for the appointment and the worker/store ID. It requests for the following:

1. `GwPeriod` & `GwPeriodPattern` where:
    - Periods that have a recurring pattern or periods involving the selected date.

```
query.where((x) =>
x.isRecurring ||
  !(x.startDate.isAfter(date) || x.endDate.isBefore(date)));
query.join(object: (x) => x.pattern);
await query.fetch();
```

2. After processing the `GwPeriod`s; and to avoid overlaps between new proposals and appointments, it requests the following `GwAppointment` data:

   - The *from* `Date` property is between the 00:00 and 23:59 of the selected date.

```
  final query = Query<GwAppointment>(context);
  query.where((x) => x.store).identifiedBy(storeID);
  query.where((x) =>
    date.isAtSameMomentAs(DateTime(x.from.year, x.from.month
        , x.from.day)));
  await query.fetch();
```

   - The last appointment of the previous day, but only if it extends to the selected date (e.g.: Appointment starts at 23:00 the previous day and ends at 1:00 of the selected day).

```
  ...
  query.where((x) =>
    date.isAtSameMomentAs(DateTime(x.to.year, x.to.month, x.
        to.day)));
  await query.fetchOne();
```

   - And the first appointment of the next day, but only if it starts between 00:00 and appointment duration ([0, dur]).

```
  ...
  query.where((x) {
  final nextday = date.add(const Duration(days: 1));
  return x.from.isAfter(nextday) &&
    x.to.isBefore(nextday.add(Duration(minutes: dur)));
  });
  await query.fetchOne();
```

To offload the backend and not assign further processing aside from querying the database, all `GwPeriodPattern` objects and respective `GwPeriod`s are processed on the frontend. It will compute a sequence of `GwPeriod` objects using the `GwPeriodPattern` from the original period, and find whether they should be considered for the current proposal. After gathering all relevant mute periods and appointments, their times are merged and reversed, resulting in empty slots as shown in the following image:

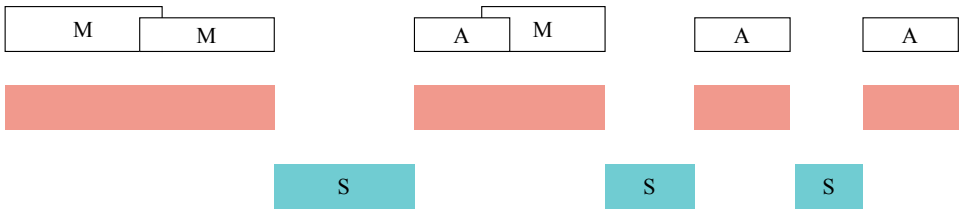

Figure 34: M — Mute Period; A — Appointment; S — Empty Slot

Slots that are shorter than the duration required for the appointment are removed from the final slot list, leaving only the ones capable of fitting it in.

Even though **GigWho** compels clients to pick slots longer than what is actually needed, to provide the worker with enough flexibility to elaborate their schedule, clients are also able to pick slots that perfectly fits their appointment. **GigWho** applies an overall restraint of four times (4x) the appointment required duration, if equal to or less than 1 hour, otherwise three times (3x) longer. This means than if the initial duration is x, the client has to pick slots at least three to four time longer ($3*x$ to $4*x$); but only if there is a slot with a duration equal to or higher than $3*x$ or $4*x$. Consider the following valid examples, for x equal to 1 hour:

- **Slot 1**: Has **1** hour, from **12:00-13:00**. This is a **valid** slot and the client can pick it.
- **Slot 2**: Has **2** hours, from **12:00-14:00**. This is a **valid** slot as well.
- **Slot 3**: Has **6** hours, from **12:00-18:00**. This is a **valid** slot and client can pick the following combinations when applying the **GigWho**

default restraint of four times the x value. The options are:

- **12:00-18:00** (6 hours)
- **12:00-17:00** (5 hours)
- **13:00-18:00** (5 hours)
- **12:00-16:00** (4 hours)
- **13:00-17:00** (4 hours)
- **14:00-18:00** (4 hours)

If clients are not satisfied with the final slot proposed by the worker, they are able to answer with a new slot suggestion with a smaller restraint value. For appointments equal to or less than 1 hour the restraint is now three times (3*x) the duration of the appointment and for longer than 1 hour it becomes two times (2*x), decreasing 1 unit for both the previous contraints. However, replies from the client to the worker are only possible if the worker flagged the proposal as flexible, with the *isFlexible* property from the `GwProposal` object. If *isFlexible* is set to `false`, the client can only accept or reject, completing the proposal process either way.

The restraints defined are not written in stone. The experience, or lack of, from using the type of services supported by the platform has lead to the implementation of such model that appears to be adjustable to all parties involved, prioritizing the worker. However, experience differs between users, and especially when comparing clients to workers. Also, within service types, **GigWho** slot logistics might work better for some but worse for others. Whereas for barber and hairdresser this model can provide simplicity, for tattoo artists and body piercers, where one project might split across multiple appointments, it might not. Overall **GigWho** encourage large slots to give workers room to decide how to best fit their clients.

Proposal Push-Notifications

The communication between the client and the worker happens within **GigWho** but both are notified via push-notification. The implementation uses the Firebase Message external package to access the Firebase Cloud Messaging server API. Each device is listening to notifications from the server, using its own widget class. The widget can process message events provided with three handlers:

- The app is currently opened using the `onMessage` event handler.
- The app is idle using the `onResume` event handler.
- App is closed, and not running, using the `onLaunch` event handler.

The **GigWho** server is the one generating the message payload and triggering the FCM server. Considering the proposal as example, it triggers the message right after a successful insert proposal transaction in the database. It fetches the *fcmToken* property from the `GwDevice` table using the worker ID and sends the newly `GwProposal` object as JSON. The snippet below has a code sample for the scenario above:

```
final query = Query<GwDevice>(context)
  ..where((x) => x.userId == proposal.worker.id));
final deviceInfo = await query.fetchOne();
...
// Performing the POST request.
await http.post(
  'https://fcm.googleapis.com/fcm/send',
  headers: <String, String>{
    'Content-Type': 'application/json',
    'Authorization': 'key=$FcmContext.serverToken',
  },
  body: jsonEncode({
    'notification': <String, dynamic>{
      'title': 'You\'ve received a new Proposal.',
      'body': proposal.toJson(),
    },
    'priority': 'high',
    'data': <String, dynamic>{
      'click_action': 'FLUTTER_NOTIFICATION_CLICK',
      'status': 'done',
    },
    'to': deviceInfo.fcmToken,
  }),
);
```

Listing 9: GigWho Server sending message to FCM Server

The snippet below is how the **GigWho** app handles messages from the FCM Server using the Event handlers mentioned above:

```
class _NotificationWidgetState extends State<NotificationWidget> {
  final FirebaseMessaging _firebaseMessaging = FirebaseMessaging();

  /// Proposal, in its current state, is accessible from the Proposal
      View.
  void _showProposalDialog(GwProposal proposal) {
    ...
  }

  void _openProposalView(GwProposal proposal) {
    ...
  }

  /// Init.
  @override
  void initState() {
    super.initState();
    // Register Firebase event handlers.
    this._firebaseMessaging.configure(
      onMessage: (Map<String, dynamic> message) async {
        // Event handler for when the app is currently opened.
        final proposal = message["body"] as GwProposal;
        this._showProposalDialog(proposal);
      },
      onResume: (Map<String, dynamic> message) async {
        // Event handler for when the app is idle.
        final proposal = message["body"] as GwProposal;
        this._openProposalView(proposal);
      },
      onLaunch: (Map<String, dynamic> message) async {
        // Event handler for when the app is closed.
        final proposal = message["body"] as GwProposal;
        this._openProposalView(proposal);
      },
    );
  }
  ...
}
```

Listing 10: Handle Proposal Events Example

Other push-notification related events, such as when modifying the ap-

pointment state (cancelled, or location changed), or movement-related events (going, arrived...) work exactly the same.

6.10 Scalability (Backend)

GigWho backend was built using the Aqueduct Framework[16] supporting multi-threading out of the box by enabling multiple instances of the server application to run across multiple threads on a physical machine. To each thread is assigned an instance of the **GigWho** backend application. All instances are replicas of each other and capable of handling multiple HTTP requests simultaneously. The number of instances is specified when first deploying and initializing the application.

Since **GigWho** is already capable of spawning multiple instances of itself, it's effortless to scale the service horizontally whenever required, spreading more instances amongst a cluster of machines. This, however, relies on the capabilities the provider hosting **GigWho** server, although, nowadays, most of the cloud service providers support horizontal scaling automatically.

[16] "An object-oriented, multi-threaded HTTP server framework written in Dart." — https://aqueduct.io

7 Evaluation

In this section it is verified the quality of the entire **GigWho** solution. To all requirements it has to be assigned a value from the following set of values: 2, 4, 6, 8, 10, representing the importance they have for the quality factor they belong to.

7.1 Functional Dimension

Represents **GigWho** functional requirements.

7.1.1 General

ID	Description	Weight	Level of Fulfilment
FG01	User creates a User Account	10	100%
FG02	User creates a Store Profile.	10	100%

Both User account and Store profile are **GigWho** basic elements and have a weight value of 10. Both requirements were achieved with success, implementing a state-of-the-art authentication and authorization flow and a secure storing solution for sensitive data specific to the user. Anyone is able to create a user account to use the app, as well as create a store profile and build their online store.

7.1.2 Social

FS01	Worker creates, edit and removes posts from their store profile.	10	100%
FS02	User “likes” a post from a worker’s store profile.	10	100%
FS03	User “shares” a post to outside the platform.	6	0%
FS04	User “follows” a worker’s profile page.	10	100%
FS05	User applies category filters to the content being shown by the app.	6	100%

As for the platform social features, all were achieved with success (100%)

except for the **FS03**. The **FS01**, **FS02**, **FS04** are internally fundamental for the social aspect of the platform and therefore have a weight value of 10. Since **FS03** involves an external component (to where the content is being shared), it was assigned a weight value of 6. Apply category filters (for the Feed View and Map View), on **FS05**, is not fundamental and only enhances the users' usage experience; as such, it has a weight value of 6.

7.1.3 Work

FW01	Users make proposals to Workers	10	100%
FW02	Workers create, edit and remove services from their service menu	10	100%
FW03	Workers create, edit and remove work addresses	10	100%
FW04	Users (workers and clients) signal movement related activities	8	100%
FW05	Users change the state of an appointment	10	100%
FW06	Users can filter appointments by multiple parameters	6	75%

For the Work quality factor, the **FW01**, **FW02**, **FW03** and **FW05** are mandatory and have a weight value of 10. Since **FW04** does not impact the communication between the client and the worker, but would effectively improve the experience for both, it has a value of 8. Finally, the **FW06**, with a 6, is another nice-to-have feature.

The **FW06** is partially completed, missing the application of filters based on a text input that would allow users to filter/search appointments associated with a specific search term (e.g.: appointments from a specific store, appointments that have a certain service). All other requirements are completed using the capable **GigWho** backend infrastructure and an external messaging server for push-notification between devices.

7.1.4 Settings

FSS01	Worker mute proposals	10	100%
FSS02	Workers customize the type of work they perform	10	100%
FSS03	Worker customizes a rate per kilometre	10	100%
FSS04	Worker customizes the max number of proposals	10	100%
FSS05	Worker customizes the max number of services per proposal	10	100%

Since this is a platform for the workers to enhance their working process, all the items above have a weight value of 10 and were all implemented successfully. From the settings view, workers can swiftly personalize these settings.

7.1.5 Search

FSH01	User search for Store Profiles	10	100%

Having the ability to effectively search for exactly what you are looking for is imperative and probably one of the most used features of a platform with search capabilities. **GigWho** search functionality allows users to search for stores by its name, worker's name and, by integrating Geolocator, Geocoding and Geohash solutions, by named places (e.g.: street name, city name).

7.1.6 Static

FST01	Mobile Application must show a marker on top of each Store location	10	100%
FST02	Content is shown in a suggestive manner	6	50%
FST03	Content is requested and shown automatically, as the user scrolls further	6	100%

These requirements refer to the app expected static behaviour. The **FST01** is what allow users to locate stores on the 2D Map view and has a weight value of 10. Regarding the method used for content display, on **FST02**,

the user current location is being considered but without the attributes that would classify the store as trending. The **FST03** was successfully implemented and data is fetched automatically. Both the **FST02** and **FST03** have a weight value of 6 as they don't directly impact the flow between clients and workers.

The **FST01** was implemented using the backend infrastructure, an external map service and a cluster solution, whereas the **FST03** implements infinite scroller detecting user scroll movements and requests data to the **GigWho** backend, that supports pagination, as it is needed.

7.2 Non-Functional Dimension

Represent **GigWho** non-functional requirements. They define attributes outside the scope of the **GigWho** app itself and instead target performance, system hardware, usability.

Adaptability

A01	Mobile Application must be implemented for both Android and iOS platforms	10	50%
A02	Interface language according to selected language	6	100%
A03	Application must adjust itself to different screen resolutions	8	50%

A01 has a weight value of 10 as it must behave the same way for Android and iOS. Since Flutter was the framework used to develop **GigWho**, and every used external extension is platform agnostic, it is safe to assume the mobile app works on both Android and iOS; however, basic functional testing for iOS was not performed and is still required. Despite existing solutions that allow iOS development on machines that don't natively run MacOS, such as VMs or MacOS cloud instances (mostly paid), the project didn't account for iOS builds. Therefore, for this requirement, the fulfilment level reached was only 50%.

The application interface (**A03**) was only tested on three smartphones

and there is not enough data to guarantee the UI would adapt accordingly to the multiple device screens/resolutions on the market. However, tested devices had a common resolution (720p and 1080p). It was given a level of fulfilment of 50% and a weight level of 8 as it does directly the client-worker flow even though it is a non-functional requirement.

Scalability

S01	Mobile Application connects to an external server	10	100%
S02	Mobile Application should only store data required for business logistics	6	100%
S03	Backend infrastructure should scale	8	50%

S01 and **S02** requirements are completed. The backend software to support **GigWho** client application was implemented accordingly to its needs and it only stores the required business data to properly operate, nothing more. **S01** has a weight value of 10 given that it is indispensable component for the GigWho platform.

The backend was developed considering future scaling needs and the Aqueduct backend framework was selected as a result. However, due to the lack of testing, no data was collected to understand how the system would behave when facing massive traffic loads. **S02** has a weight value of 6 and **S03** of 8.

Performance

P01	Mobile Application should load low resolutions assets as a placeholder on low-quality connections	6	75%
P02	Assets and data are cached locally to improve app responsiveness and network usage	6	100%

P01, with a weight level of 6, did not achieve a 100% fulfilment level

because it would require the app to automatically detect if the connection is slow or not.

Due to the implementation of a capable caching system, the **P02**, also with a weight value of 6, achieved 100% level of fulfilment.

Even though performance is vital, the app would still function without these features and, if connected to a fast speed network, the end user might not even feel these optimizations, hence the weight value assigned.

According to the QEF (Quality Evaluation Framework), the importance each dimension (Di) has to the final solution is given by:

$$D_i = \sum_n (p_n \times factor_n), \sum_n (p_n) = 1, p_n \in [0, 1] \tag{1}$$

And the contribution from each quality factor (Fn) is given by:

$$F_n = \frac{1}{\sum_m pr_m} \times \sum_m (pr_m \times pc_m), p_n \in [0, 1] \tag{2}$$

With a global deviation (D) given by:

$$D = \sqrt{\sum_j (1 - \frac{Dim_j}{100})^2} \tag{3}$$

The quality of the entire solution (Q) can be computed using:

$$Q = 1 - \frac{D}{\sqrt{n}} \times 100, Q \in [0, 100] \tag{4}$$

The formulas above were used to compute the importance of each quality factor to the solution and each completion level.

Dimension	Quality Factor	Weight	Contribution (≈%)
Functional	General	0.091	100
	Static	0.136	86
	Social	0.227	86
	Search	0.045	100
	Settings	0.227	100
	Work	0.273	97
Non-Functional	Adaptability	0.38	62.5
	Scalability	0.25	87.5
	Performance	0.38	83

Table 21: QEF — Quality factor weight contribution and fulfilment level

After applying the formulas, the quality score for the quality of **GigWho**, in its current state, is **89%**.

8 Conclusion

GigWho primary goal was achieved — the development of a concept with platform work and social media features capable to provide workers with an all-in-solution that simplifies the way clients and workers connect. Direct messaging, calls, cancellations, appointment keeping outside the platform, all of that removed to make room to a much more convenient communication system. A system that also prioritizes the worker preferences and grants them freedom to work how they want.

GigWho is a capable platform that implemented many mandatory techniques for modern mobile applications. However, it lacks testing as it didn't reach the development stage initially planned.

8.1 Limitations and Future Work

GigWho is still in development due to the considerable size of the platform. Future work includes the testing cycles where each cycle is decomposed into the following stages:

- Generate mock data relevant to the main subject (beauty and body-art). Right now, **GigWho** uses random data from free services, such as the API used to collect images.
- Elaborate a new QEF model with new requirements for a post-development phase.
- Deploy **GigWho** backend on the cloud.
- Deploy the app on a testers program. For Android there is Play Console, and for iOS TestFlight.
- Write surveys, for the testers, to provide feedback over the use of the **GigWho** App.
- Send out invites to users.
- Collect the surveys and analyse the results.
- Compute the overall quality of the solution for this phase.

- Implement the requirements accordingly.

The testing phases might be split into an initial **internal phase**, with only a small and private group of users testing the most basic functionality; proceeded by an **alpha** phase, followed by an open **beta** phase.

From alpha to beta, the app would stop using mocked data and expect actual workers to utilize it as it is supposed to be. In the beta program, the app is released to the public and accessible to almost everyone. Also, in this phase, **GigWho** would have to hire new resources and a marketing team to properly advertise the platform. If the primary goal is to have people using the platform to request for services, **GigWho** must ensure that there are already as many workers as possible signed up and ready to work.

References

Angela Stringfellow. What is the gig economy? how it works, benefits, and more, 07 2019. URL https://www.wonolo.com/blog/what-is-the-gig-economy/.

Intuit. Dispatches from the new economy: The on-demand economy worker study, 06 2017. URL https://intuittaxandfinancialcenter.com/wp-content/uploads/2017/06/Dispatches-from-the-New-Economy-Long-Form-Report.pdf.

Alex Moazed and Nochilas L. Johnson. *Modern Monopolies*. St. Martin's Press, 05 2016. ISBN 9780321617743.

Georgia-Rose Johnson. Uber vs taxi: A comparison of the price of uber and taxis around the world, 01 2020. URL https://www.finder.com/uk/uber-vs-taxi#tab2525.

Sahil Sankhla. The rise of beauty services in the on-demand market, 05 2018. URL https://jungleworks.com/the-rise-of-beauty-services-in-the-on-demand-market/.

Rimma Kats. Consumers are influenced by brands on social, 09 2019. URL https://www.emarketer.com/content/consumers-are-influenced-by-brands-on-social.

Matercard and Kaiser Associates. The global gig economy: Capitalizing on a $500b opportunity, 05 2019. URL https://newsroom.mastercard.com/wp-content/uploads/2019/05/Gig-Economy-White-Paper-May-2019.pdf.

J. Clement. Number of worldwide internet users 2009-2019 by region, 10 2019. URL https://www.statista.com/statistics/265147/number-of-worldwide-internet-users-by-region/.

Marko Milijic. 29+ smartphone usage statistics: Around the world in 2020, 10 2019. URL https://leftronic.com/smartphone-usage-statistics/.

Mobile operating system market share worldwide. URL https://gs.statcounter.com/os-market-share/mobile/worldwide.

Delloit. The rise of the platform economy, 12 2018. URL https://www2.deloitte.com/content/dam/Deloitte/nl/Documents/humancapital/deloitte-nl-hc-reshaping-work-conference.pdf.

V. Allen. Value network analysis, 01 2012. URL https://www.uio.no/studier/emner/matnat/ifi/INF5120/v12/undervisningsmateriale/ValueNetworks2012023.pdf.

V. Allen. Value network analysis and value conversion of tangible and intangible assets. *Journal of Intellectual Capital*, 2008.

Android. Keeping your app responsive, 12 2019. URL https://developer.android.com/training/articles/perf-anr.html.

Tony Woodall. Conceptualising 'value for the customer': An attributional, structural and dispositional analysis. *Academy of Marketing Science Review*, 12, 01 2003.

T. L. Saaty. Decision making with the analytic hierarchy process. *Int. J. Services Sciences*, 2008.

Jon Yablonski. Laws of UX, 01 2020. URL https://lawsofux.com.

Marius Manic. Marketing engagement through visual content. *Bulletin of the Transilvania University of Brasov. Economic Sciences. Series V*, 8(2): 89, 2015.

Shantanu Kher. Again and again! managing recurring events in a data model, 10 2016. URL https://www.vertabelo.com/blog/again-and-again-managing-recurring-events-in-a-data-model/.