Voxreel iOS Application

A mobile application for recording and sharing short but insightful video impressions at various businesses

Andreas Lukas Rauter

MEng Computer Science

Supervisor: Dr. Graham Roberts

Abstract

The aim of this project was to develop a video sharing, mobile application, where users can record and share real experiences at various businesses with other users, culminating in a user generated film about a business. Most notably, my application allows users to discover businesses and record short videos about different aspects of the venue, which are then compiled into searchable and insightful showreels. With this approach, I firstly hope to overcome the challenges in the landscape of current reviewing based applications by moving away from the concept of often corrupt, misleading and reputation damaging, written reviews and ratings. Secondly, I anticipate transforming the conventional method of discovering venues into a quicker and more engaging experience, by providing users with rich, informative and transparent video insights, consequently facilitating users' decision making.

My work for this project was primarily focused on two aspects: (1) background research on the negative effects of written reviews and the alternative of using videos as well as identifying the shortcomings of competitor applications, (2) development of an iOS application and a RESTful API providing users with creative features to discover businesses through various search mechanisms, capture user experiences by recording and sharing videos, and stream and browse through video contents via customisable showreels.

The project delivers a mobile application and supporting API service, together offering an alternative approach to business reviews focused on convenience, simplicity and user engagement. With further feature improvement and the raising of funds, a beta launch will soon follow.

1 Introduction

1.1 Problem Description & Motivation

Nowadays more and more businesses expand into the world of online review by exposing themselves on popular reviewing applications such as Yelp, Foursquare or even Facebook or Google. Their general goal is to increase their users' reach as well as provide the invaluable opportunity for customer interaction [1]. More importantly, customers themselves obtain insights into owners' businesses, thus facilitating their decision-making process when visiting new places.

The majority of current reviewing systems are usually based on users' personally written text and star ratings. One major issue emerges from the fact that while businesses can hugely benefit from positive online ratings and reviews, negative reviews can destroy a business's reputation. For instance, according to Yelp, a negative review can cost a business up to 30 customers whereas between 1 and 3 negative reviews could deter up to 67% of a business's customer base [2] [3].

Furthermore, up to almost one third of current reviews are made up, thus posing a considerable cause for concern regarding the expectation gap between consumer reviews and actuality. This raises important questions about systematic bias and the ease of data manipulation [5]. This leads to another related problem referred to as ratings herding/bubble, a phenomenon of users being more likely to share positive or negative feelings towards a product or experience due to other people's reviews without necessarily even experiencing it for themselves [6]. As a result, online reviews should be deemed untrustworthy as a whole, consequently creating big opportunities for new, innovative solutions.

Lastly, the medium of text dominates the online review space even though it provides little actual evidence and insights. Text reviews are only truthful to the people that wrote them, whereas others can read and interpret them in many different ways. In other words, reading reviews is focused on rethinking and reimagining the experience of another consumer, while in contrast watching a video illustrates the experience as it occurred [7]. The latter leaves less room for interpretation and the information a user requires is more readily available and consumed in a more efficient way.

Voxreel's solution of using videos for a more realistic, more trustworthy approach of sharing experiences that does not involve the reading of written reviews, is the future way of tackling the issues of traditional reviewing systems.

1.2 Proposed Approach

Recording transparent videos: Voxreel's main idea is to generate user experiences at various businesses by recording insightful videos as they occur, resulting in the capturing of pure information for others to watch, consequently closing the expectation gap and preventing users from being misinformed. Once users see something they fancy or find interesting during their stay at a business, they can easily record short video clips to show their insights within the business. For each recorded video the user has to select a common theme (e.g. "Food", "Drink", "Inside", "Outside") appropriate to the venue type in order to label the content of the Vox, followed by providing useful metadata information to further describe the video. Ratings are completely ignored and abandoned to limit herding and the highly damaging effects of negative reviews, therefore encouraging users to capture positive experiences only.

Searchable and engaging showreels: Voxreel introduces a quick and highly engaging way of demonstrating other users' generated videos, by compiling them in reels. Each business has its own tailored showreel, whose videos can be searched for specific information as well as sorted by any criteria the user might have. For instance, users might require details about the business's internal or external view or want to gain insights about the sort of food it offers, so they could search specifically for these videos by narrowing down the reel. This searchable video function clearly prevents users from browsing through irrelevant content before committing to a decision as well as offering a fun, new way of searching through content.

Discover businesses: Without the ability of finding businesses, the concepts of recording videos and watching businesses' showreels would be groundless. Even though it is not explicitly stated as a challenge in the problem description, an easy way of discovering venues is crucial for my application's existence. Voxreel provides several searching capabilities that enable users to find venues by location, keyword or category, depending on their intentions.

1.3 Aims

- Learn Swift and iOS development.
- Learn the PHP backend framework Laravel for building a RESTful API and gain knowledge in building scalable and robust backend solutions.
- Enhance PHP skills by learning Laravel's PHP structures.
- Acquire experience in creating animated user interfaces, dealing with HTTP networking, applying caching strategies, recording and streaming videos, operating within app databases and developing custom search algorithms.
- Investigate the negative impacts of written reviews and ratings and how the use of videos can overcome such issues.
- Investigate existing competitor applications by illustrating their strengths and weaknesses.
- Design and implement a new mobile platform as part of my Startup Voxreel, in the hope of ultimately defeating the landscape of current reviewing based applications, by shifting away from the concept of often corrupt, misleading and reputation damaging written reviews.

1.4 Goals

- Conduct background research on the effectiveness of videos including a critical reflection on today's reviewing culture.
- Produce competitor analysis on current reviewing applications such as *Foursquare* and *Yelp* to identify their existing shortcomings.
- Deliver a working iOS application with the following key features:
  - **a)** Users can record and share short videos, annotating them with useful metadata.
  - **b)** Users can search for businesses using various searching algorithms.
  - **c)** Users can stream and browse through venues' user generated video reels.
  - **d)** Users can search video reels for specific content.
- Deliver a dynamic RESTful API that supports the features of the IOS application including the management and permanent storage of relevant data (most notably business and video related information).

1.5 Development Approach

My project was carried out in several independent stages: (1) research (2) requirements refinement (3) development and testing, and (4) report writing.

I conducted my initial background research during the summer of 2016, right after I decided to co-found the Startup Voxreel. I was mostly concerned with defining a project scope that would allow me to incorporate my Startup endeavours into my individual project, consequently helping me in writing an application for both purposes. Starting from early October to the end of November, I refined most of my application's MVP requirements and investigated the various technologies I would be requiring for the development of the iOS application and the API. Afterwards, my project was carried out in an iterative approach by applying an agile development methodology [8]. Each sprint would be dedicated to an application feature and would usually take between two and three weeks, depending on complexity. An iteration would normally consist of firstly designing the feature, followed by implementing and finally testing it. Note that this stage was assisted using Trello[1], a project management tool I have been applying to other projects throughout my university career due to its simple and effective service in organising workloads. By the end of February I was able to complete my application's key features and demonstrate the final product to my supervisor. During the remaining time, I was primarily focused on drafting and composing this project report alongside completing the code documentation.

1.6 Report Structure

Following this introduction, a brief outline of the report's remaining chapters is described below.

Chapter 2 discusses the research carried out to investigate the nature of current reviewing systems and the power of video as a practical solution to their problems. This is followed by an overview of existing competitor applications using traditional reviewing mechanisms illustrating their shortcomings. Lastly, I reflect on my development approaches including decisions I had to make in regards to frontend and backend technologies, location autocomplete libraries, business sourcing alternatives as well as video streaming and hosting options.

Chapter 3 illustrates the requirements analysis, which firstly focuses on how the initial requirements were derived, followed by a summary of concrete requirements and use cases. Furthermore, an entity-relationship diagram is given, serving as a high-level data storage description for the ultimately implemented webserver database described in chapter 4.

Chapter 4 covers the system's design and implementation. Firstly, it discusses design principles followed by the iOS application and the RESTful API, and highlights notable design patterns applied. Next, a complete description of the system architecture is provided, whose components are each explained in great depth in the following subchapters including their notable implementation details. Lastly, I elaborate on the webserver database structure.

Chapter 5 describes the various testing strategies applied to the iOS application and RESTful API, for each discussing used automated and manual testing methodologies. Ultimately, a testing summary is provided.

Chapter 6 contains the project's conclusions and evaluation. This comprises evaluating the project's outcomes with the initially set goals and a critical observation on how the system was developed and designed. In the end, further development suggestions are given including a personal reflection on the overall project.

[1] https://trello.com/

2 Context

2.1 Business Background

It is important to note that my project emerged from the UK based Startup company named Voxreel I cofounded together with my colleagues: *Oliver Graham*, *William Graham* and *Mark Lloyd*. Voxreels are reels of short and insightful video clips or "Voxes" expressing user experiences at a business.

2.2 Background Research

2.2.1 Online Ratings & Reviews Systems

Online reviews play a significant role in helping consumers make e-commerce decisions [9]. Especially personal recommendations and online reviews from friends and family are trusted by 88% of consumers and read by 90-91% [4]. Hence, it is no great surprise why businesses are so active in the online review space as they have the opportunity to receive massive exposure, increased web traffic via improved Search Engine Optimisation (SEO) as well as the invaluable opportunity for customer interaction [1].

Businesses can benefit immensely from positive online ratings and reviews. However, the effects of negative reviews can be rather detrimental to business as they impose a massive challenge on the venue to recover its image once the damage has been done. Yelp, one of the most popular mobile app, publishing user generated reviews about local businesses, supports this claim by showing that one single negative review on their platform can cost a business up to 30 customers and that multiple negative reviews (between 1 and 3) could deter up to 67% of a business's customer base [2] [3]. This is especially concerning for the sector that consumers are most interested in reading about, i.e. hospitality - the sector Voxreel is targeting [4].

However damaging or promoting online ratings and reviews might be for businesses, the hard truth is that up to 30% of reviews are fake [5]. This poses a considerable cause for concern regarding the expectation gap between consumer reviews and actuality, as they do not provide truthful visceral information about venues. Instead, there is a high risk of misinforming users. Additionally, there is the chance that the context of any review or rating may not relate to any one specific user, again falsifying users' expectation.

This leads me to discuss another major issue with ratings and review systems: herding ratings bubbles. Herding, in the context of reviews and ratings, means one is more likely to share positive or negative feelings towards a product or experience other users have, without necessarily even experiencing it for themselves [6]. This was supported by the so-called 'Social Influence Bias: A Randomized Experiment' [6]. By making simple alterations to the "likes" or "dislikes" of reviewers' comments on a news aggregation site, they found that other users were socially influenced greatly in their reviews. Whilst positive social influence had more of a herding effect than that of negative social influence, both positive and negative herding was evident in this experiment. If we know that nearly a third of online reviews and ratings are fake then for those that are not, many of them could simply derive from positive and negative social influences.

Another source of corrupted reviews are consumers deliberately being requested to write them. 70% of consumers will actually leave a review for a business when asked by that business, regardless of whether they have anything worthwhile to say [10]. This can and often does work the other way as well. In an open playing field like this, businesses can ask consumers to leave negative reviews about their competitors. It has become a legal issue regulated by the Competitions and Markets Authority (CMA), who make it clear that writing or commissioning fake reviews

could lead to civil or criminal action [11]. However, this has not prevented fake ratings and reviews from occurring. Also, even if it was possible to police all websites for fake reviews and remove them, their "legacy" lives on in the future and will pose a great challenge for businesses to reconcile [9].

There might be another big issue that researchers and companies are overlooking, that is the medium of text dominating online review. A recent study carried out by [10] demonstrates that 58% of consumers pay most attention to "Overall star ratings", right next to the second most important factor "Sentiment of reviews" with 47% (see figure 2.1). Based on the evidence suggested in this study, users heavily focus on subjective matters, i.e. star ratings and written text. Given the fact that these can be read and interpreted in various ways, there is a great chance that they end up providing little in the way of actual information.

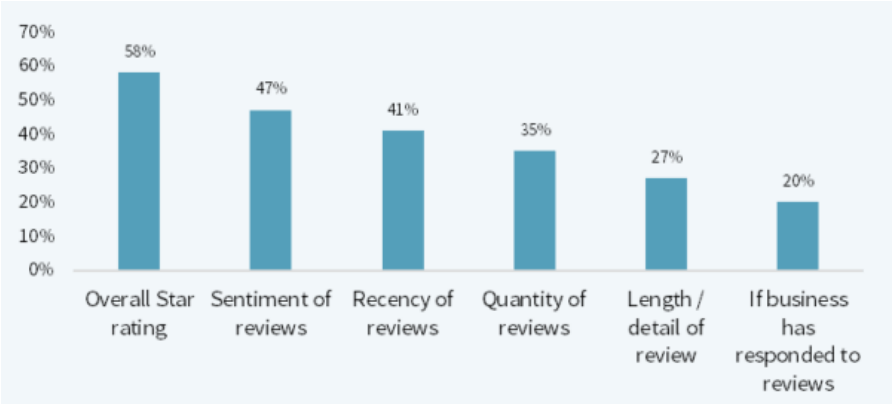

Figure 2.1 Customers attention factors when judging businesses [10]

Despite the ever-growing trust of and reliance on online reviews and ratings, the evidence suggests that they are both highly corruptible and highly irrelevant to the needs and wants of the individual consumer and should therefore not be trusted. They create an end user that is misinformed either by other users that may be demographically irrelevant to their activities, interests and opinions or by fraudulent information.

2.2.2 The Power Of Video In Experience Sharing

McLuhan's work is one of the cornerstones of media theory, as well as having practical applications in the advertising and television industries [12]. He claims that a medium affects the society in which it plays a role not only by the content delivered over the medium, but also by the characteristics of the medium itself [12]. It is important in the context of Voxreel to consider his work when understanding how consumers share and utilise information before making purchasing decisions.

Research carried out by Forrester Research concludes that 1 minute of video is worth around 1.8 million words, which would take 120 hours to read [13]. When humans read something, their cognitive processors are actively involved in processing the information, resulting in heavy and long brain computations [14]. In contrast, videos are processed by the brain 60,000 times faster, thus requiring a lot less concentration [14]. In other words, information you require when watching a video is more readily available and is consumed in greater depths more efficiently.

Great examples demonstrating the popularity of videos can be found in the social media industry. More and more people are using Snapchat, Vine and Instagram to send messages to and share experiences with their friends and family. The onus therefore is on venues to be active in these spaces if they are to be visible and relevant to their potential customers and to help them make purchasing decisions. For instance, businesses on Snapchat can share short stories, show flash sales and ads for coupons as well as create fun and engaging competitions for active users.

Instagram enables businesses to build communities of customers while showcasing their products and services with short videos and pictures. Businesses can re-share a post that was originally shared by a happy customer as well offer incentives and giveaways to users that help to spread the word. Similarly with Vine, businesses can create videos of positive customer experiences, do product placement and "Re-Vine" customers talking positively about their product and services as well as promote discounts and coupons to their followers [15]. The common link between these platforms is that the users are forced to record for a short amount of time in order to convey their message or experience. If we think back to the statistic about 1 minute of video being worth around 1.8 million words, it is easy to see why consumers are using video more and more. It is a multi-sensory experience and in a society with ever decreasing attention spans and an increasing desire for new experiences, it is crucial that we find more efficient ways of disseminating information to save precious time for said experiences [16].

Voxreel aims to take this growing awareness of the power of video and apply it to experience sharing within the hospitality space. By allowing users to record short impressions of local businesses with shots like interior pans showing the size and layout of a place, close ups of the food and drinks served there, any entertainment that might be happening and anything else that captures the general buzz and busyness of a place, Voxreel provides a great alternative to scrolling through reams of anonymous text reviews. There does not have to be a context; it does not have to be a review per se, it can simply represent a view into a place for users to see for themselves.

There are a whole host of reasons however, why people, business owners and their employees may not want to be caught on camera. Aside from that, businesses might be concerned about inaccurate and/or negative exposure of their businesses. According to Yelp, there have been many PR disasters when it comes to photos being shared to various social media sites from both upset employees and customers [17]. The power of video could take this level of sabotage to the next level and could contribute to irreparable reputational damage. These are very real concerns that are impossible to provide 100% fail safe solutions to. Nevertheless, it is integral that the inherent design of Voxreel is such that users are encouraged to create relevant recordings. The following are the main strategies Voxreel intends to use:

1) **Videos are much easier to verify than text.** There is no proper approach of telling if a written review is fake, making it almost impossible to determine whether it should be taken down or not. However, with the right amount of human resources (or automated software solutions), it is relatively simply to assess whether a video is not geographically tagged to the relevant business, does not display relevant content or has a low-quality standard.
2) **Rewards for useful videos.** Videos displaying useful and well shot content will be rewarded with points in the form of likes from other users. The points are gathered up over a period of time and the winners will receive vouchers to spend in any of the listed Voxreel businesses. This will naturally encourage more users to create useful videos.
3) **Ranked videos:** After recording a video, users should be asked to assign a theme to their shot and tag necessary information to it so that it can be searched and easily viewed by other users. Useful videos, the ones with more likes, should then receive the higher rank in the showreel. Again, this should encourage users to provide useful and relevant video recordings.

In conclusion, Voxreel's use of video provides a practical solution to the unreliable, corruptible and unmanageable nature of text reviews. Despite the outlined pitfalls that come with video, it is clear that sharing experiences with video provides a more engaging, more enjoyable and trustworthy platform.

2.3 Competitor Applications

This section is dedicated to elaborate on the useful features and shortcomings of two main mobile reviewing applications - Yelp[2] and Foursquare[3]. The following competitor analysis is mainly focused on their approaches in regards to searching and viewing businesses, alongside adding, viewing and browsing through crowd-sourced reviews. I deliberately do not discuss their social media related features, as they are not within the scope of this project.

2.3.1 Yelp

Overview	Yelp's mobile app is an application that offers user reviews and recommendations of restaurants, shopping, nightlife, entertainment and other services [19]. Although Yelp is known for its convoluted relation with local businesses, it is nonetheless one of the most popular reviewing application with 135 million monthly users and 95 million reviews [19] [21].
Strengths	- It provides great searching features that allows users to instantly find new places by category, keyword, or location. In particular, it has a dedicated page that lists and updates nearby businesses, hence providing a quick and easy way of finding what venues surround someone without the need for any user input. Furthermore, search results can be narrowed down via numerous filtering options. - Users have the ability to add photos to their reviews in order further visualise their experience. Also, each review has a "Useful", "Funny" and/or "Cool" counter attached to them, which can be up voted by any registered user. Based on these counters, users can obtain a shallow understanding of whether or not a review is worthwhile reading. Apart from traditional reviews, users have the opportunity to compose tips, which can receive likes from others, thus providing a way of demonstrating their usefulness. - If a business contains many reviews, they are grouped together based on a common theme (e.g.: "dinner", "reservation", "etc."). Additionally, users can manually search for review content. Both mechanisms can certainly facilitate the decision-making process as users have the ability to find specific information quicker.
Weaknesses	- Each review forces the user to give a five-star rating. As discussed in 2.2.1, ratings are highly biased or fake, which can consequently irreversibly damage a business. - The application offers a feature to record and add videos to a business, nonetheless it is completely hidden away from the user. Moreover, its video capturing process only includes the recording of the video but not the ability to annotate it with useful and more specific data describing its content.

[2] https://itunes.apple.com/us/app/yelp-the-best-local-food-drinks-services-more/id284910350?mt=8

[3] https://itunes.apple.com/us/app/foursquare-city-guide-restaurants-bars-nearby/id306934924?mt=8

2.3.2 Foursquare

Overview	Foursquare is a local search-and-discovery service mobile app that helps users in finding places to go with their friends [22]. With a 45 million user base, Foursquare is certainly one of Voxreel's main competitors [22].
Strengths	- Similar to Yelp, Foursquare provides almost the same user friendly ways of searching and filtering businesses. In contrast to Yelp, it does not provide an always accessible list to nearby venues, meaning each search must be triggered manually by the user. - The app encourages users to write tips (good or bad) instead of traditional lengthy reviews. Tips are limited to 200 characters in length and can include photos or external links to more information [2]. As a result, the content appears to be more concise, speeding up the reading process. Additionally, tips can be set to expire after a certain amount of time. This is especially useful for temporary tips about promotions or events. More interestingly, users can both up and down vote each other's tips, thus discouraging users from writing misleading and offensive comments. - Similar to reviews on Yelp, tips can be searched using predefined filters. Again, this helps users in finding content quicker.
Weaknesses	- Apart from predefined filters, users cannot search for specific user content. In other words, to form a proper decision about a venue users are forced to read through multiple, possible irrelevant tips. - Users are able to rate a business on a "like", "okay" and "dislike" basis. These ratings alongside tips and check-in data feed into a venue's score (between 0.1 and 10) to indicate its general popularity [22]. Given the fact that users pay heavy attention to these ratings as discussed in 2.2.1, lower scores resulting from subjective tips and ratings can certainly negatively affect a business. - The app does not support any video recording/sharing functionality. Users are restricted to make decisions based on a venue's rating, tips and photos.

2.3.3 Summary

Yelp and Foursquare offer solid business search features, from which my application can draw ideas from. Discovering businesses based on the user's location seems to be the predominate use case both applications really focus on when it comes to finding new places. Next, each competitor application primarily supports written reviews in combination with optional photos. Even though, an image does provide some sort of visual insight into the user's experience, it is nonetheless restricted by its expressiveness, as it requires the support of some text to convey its meaning. Moreover, while Yelp follows the traditional reviewing approach, lengthy text together with five-star ratings, Foursquare allows users to either write brief tips (either good or bad) and/or provide ratings separately. Shorter reviews can of course have the benefit of being more compact and quicker to read, however the interpretation of the text and ratings remains subjective.

To narrow down the review space in order to speed up the searching process for specific content, both Yelp and Foursquare follow similar techniques. The use of filter words or manual content searches represent their main strategies. When it comes to video capturing, Yelp vaguely supports it, however completely hides the feature within the application, whereas Foursquare entirely abandons any video related features in the first place. There is no question that both mobile apps primarily rely on the combination of text and/or rating to express users' experiences,

which given the information in 2.2.1, henceforth demonstrates their main shortcoming, from which Voxreel can learn.

2.4 Development Research

2.4.1 Frontend Technologies

Swift over Objective-C: Prior to the development of Voxreel's iOS application, I faced the choice between using Apple's fairly new programming Swift (2014) or the relatively old language Objective-C (1984) [23] [24]. Around September right before the start of the project, Apple launched the latest Swift version 3, which made me skeptical whether Swift would be stable enough to develop a future proof application. However, my research proved me wrong, showcasing that Swift has indeed many advantages over Objective-C, e.g.: faster to execute, easier to read and maintain, safer and less-error prone, unified with memory management, less code required, etc. [25] [26]. More importantly, Apple and IBM have decided to join efforts and invest in Swift together, while Objective-C seems to be left behind and is used as a legacy language [27]. More importantly, Swift is open-source, which will make it "more portable and versatile, enabling its use in projects outside of apps for Apple's platform" [27]. Consequently, learning Swift seemed to be the right choice in terms of productivity and ultimate usefulness.

Keyboard library: A common issue when developing for iOS is the keyboard sliding up and covering text fields or text views, thus preventing users from seeing what they type. While Android solves this issue out of the box, iOS developers need to manually fix this problem themselves. At the beginning of development, I indeed coded a simple solution that would automatically move up text fields and text views when the keyboard would appear. However, with the increasing user interface complexity on certain pages, I decided to make use of the IQKeyboardManager[4] library to save time and effort. IQKeyboardManager is one of the most popular solutions to this prevalent issue in the iOS community, thus it seemed the right choice at the time.

Networking library: The NSURLSession and related classes provide the traditional Swift API for downloading and uploading content via HTTP [20]. When dealing with extensive API communications such as in my project, the boilerplate of writing networking code becomes fairly large. By using networking frameworks such as Alamofire[5] this issue can be greatly reduced. Given its popularity within the iOS community, it appeared to be the best choice for my application.

Image caching library: As part of my project I do implement a custom app cache for storing server related data in order to reduce redundant API calls. Nonetheless, I did not see the purpose in reinventing the wheel when it came to asynchronously downloading and caching images (video thumbnails, category icons, etc.). Caching them myself would only add unnecessary difficulty to the project, instead I selected the Kingfisher[6] library to perform this job for me. Its straightforward use ultimately convinced me in choosing it over more popular image caching libraries such as SDWebImage[7].

Network reachability library: Checking for internet connection availability and receiving notifications when the network reachability changes is crucial for uploading data in the background. Similar to the image caching situation,

[4] https://github.com/hackiftekhar/IQKeyboardManager

[5] https://github.com/Alamofire/Alamofire

[6] https://github.com/onevcat/Kingfisher

[7] https://github.com/rs/SDWebImage

I chose to not implement such mechanisms myself, but rather to reuse an existing codebase, which due to the ease of use ended up being the Reachability.swift[8] library.

2.4.2 Backend Technologies

Backend framework: There are several reasons why I chose the Laravel[9] framework for my API development. To begin with, I had some prior experience with PHP web development, therefore the jump to using a PHP backend framework did not seem far-fetched, thus facilitating me in improving my PHP skillsets in the long run. Of course, I considered several other backend frameworks, most notably Node.js[10] and Django[11]. Although, such frameworks might be considered more modern and possibly more maintainable, the fact that I had to familiarise myself with Swift and iOS development from scratch ultimately made me choose a PHP framework over any other language based frameworks in order to speed up the development process. For the time being Laravel was the most popular PHP framework on GitHub, thus it appeared to be the most appropriate choice for my endeavours.

API data output library: Transforming data before passing it in a RESTful API is crucial and must be done with care. One approach is to simply take the processed and unformatted data from the database or custom computations and pass it back to the API user. This might be appropriate for simple APIs that are not open to the public or not used by mobile applications [28]. For involved APIs such as the Voxreel API, this approach would quickly lead to inconsistent output [28]. It is hence fundamental to use a presentation/transformation library to output complex data. Consequently, I chose the laravel-fractal[12] library for this task, mainly as a result of its simple usage.

Nested set model library: Part of my project involves storing an ordered category tree in an SQL database. The nested set model is the appropriate technique for representing such ordered trees in relational databases that must be queried efficiently (see 4.3.1.2 for further details) [29] [30]. Instead of manually implementing these mechanisms myself, I made use of the Baum[13] library, a Laravel implementation of the nested set pattern. Next to Baum, there is only one other major library I could have deployed, the laravel-nestedset[14] library, which I later abandoned due to its required configuration overhead.

ID Obfuscation library: As later illustrated in 4.3.6.3, my backend implements a SQL database schema using MySQL as its RDMS. It is common to have sequential IDs in SQL tables, resulting from auto incremented, numerical primary keys. Even though they are extremely useful behind the scenes, they tend to give away too much information about the application, allowing adversaries or competitors to estimate table entry counts or perform automated scraping strategies to steal data [31] [32]. Hash IDs are not designed to increase applications' security, but to rather add some layers of obfuscation [31]. In order to therefore hide business or video related IDs, I adopted the laravel-hashids[15] library.

[8] https://github.com/ashleymills/Reachability.swift

[9] https://laravel.com/

[10] https://nodejs.org/en/

[11] https://www.djangoproject.com/

[12] https://github.com/spatie/laravel-fractal

[13] https://github.com/etrepat/baum

[14] https://github.com/lazychaser/laravel-nestedset

[15] https://github.com/vinkla/laravel-hashids

Thumbnail library: Generating thumbnails is indispensable when initially loading external videos onto the iPhone device. This results in a better user experience allowing users to receive a first impression of the video. The library that aided me in the thumbnail generation process was FFmpeg[16], as it appeared to be the simplest solution at the time. Compared to other libraries I use, FFmpeg is a command line tool but can easily be called from within the PHP code.

2.4.3 Sourcing Businesses

One major challenge to solve was to determine how Voxreel would retrieve geographic location information from businesses. It should be obvious by now that sourcing businesses represents the backbone of Voxreel's platform. Without the existence of sourcing businesses, users are simply not able to find venues or associate recorded video content with them. In general, there are two main approaches to this issue:

1) **"Places" API:** All "Places" API alternatives I considered (Google Places API[17], Foursquare API[18], Yelp Fusion API[19]) generally speaking offer the same service; they take a geographic location as input and return nearby places. The Google Places API, with 100 million venues, is by far the most complete worldwide set of businesses, which would certainly allow Voxreel users to discover many places [33]. The main drawback emerges from the fact that Google prohibits applications from caching and storing business related data [34]. Place IDs are the only exception to these regulations, hence enabling applications to store their values for future use [35]. However, as stated in the Google Places API documentation [35], those IDs may become obsolete and change over time, which introduces additional complexity in tracking businesses. The other investigated APIs show similarly restrictive regulations when it comes to caching and storing business related information [36] [37]. Similar to the Google Places API, such regulations would prohibit Voxreel from reusing API data to build its custom public API in the future. Also, the Yelp Fusion API is mainly covering the U.S with only a few international cities, while the Foursquare API hides its business size completely, making both API solutions questionable. Lastly, each "Places" API is limited in its free use. For instance, the Google Places API offers a default of 1,000 free requests per day in its standard plan [38]. This quite low compared to the Fouresquare API providing 5,000 requests per hour [39] or the Yelp Fusion API giving away 25,000 calls per day [40]. Currently this is not an issue as Voxreel has no user base. This might change over time, which consequently could result in paying money for exceeding an API's call limit.
2) **User generated businesses:** This approach would allow users to add businesses manually within the application. There is no question that each created business must be checked for its validity in order to ensure the correctness and consistency of the database. While this option might be involving more work to get Voxreel running, it has the huge advantage of having control over the business data. Furthermore, having a rich database full of business and video related information can be used to make profits via a public API.

Given the restrictive use of existing "Places" APIs in combination with Voxreel's future ambitions of creating its own public API, I decided to follow the second option.

[16] https://ffmpeg.org/

[17] https://developers.google.com/places/ios-api/start

[18] https://developer.foursquare.com/docs/

[19] https://www.yelp.at/developers

2.4.4 Location Autocomplete

Choosing to create my own database, consisting of user generated businesses, introduced the new challenge of writing venue search algorithms. As thoroughly discussed in 4.3.2, my general approach was to base each type of business search on a specified location (e.g.: "United Kingdom", "London", "UCL", "N7 6BB", etc.). For this to function, I had to find a location autocomplete API that would allow users to choose locations suggestions as they type. Users not only prefer autocomplete features as illustrated by [41], but more importantly an autocomplete API allows me to translate each suggestion into a concrete geolocation, making it feasible to find the nearest Voxreel venues in the first place. It is important to note that deploying such an API was necessary, as it would be impossible for my application to provide location suggestions on its own.

The autocomplete service in the Google Places API for iOS[20] was the most scalable and reliable solution to this problem. Although this service is limited by 1000 free requests per day, which can be exceeded by an even small user base, it was still the right tradeoff to make, in the absence of any cheaper, reliable alternatives. Additionally, the Google Places API can restrict suggestions to geocoding results only. Again, this is indispensable as my application cannot conduct venue searches based on suggestions with indeterminate geolocations.

2.4.5 Video Hosting & Streaming

Lastly, I had to decide on how to host and stream videos. Initially I examined YouTube, after all it is the largest and most popular video sharing platform in the world [46]. As it turns out its Data API[21] would provide extensive uploading and streaming capabilities. Another platform with similar functionalities was the Gfycat API[22], a platform focusing rather on user-generated short videos. Both services have the advantage that the video hosting and streaming is entirely taken care of by their platform, subsequently saving me time and future server costs.

Yet, I would have no control over the video content. Once users upload their videos, there is no guarantee they will remain accessible for my application [42] [43]. In addition, other users could download my video content and reuse my video data without my permission. Also, both investigated APIs require users to authenticate themselves either via their Google, Facebook or Twitter account, which would impose additional authentication layers on my application [44] [45].

Consequently, I chose to host and stream videos myself using a dedicated server. While uploading them is pretty straightforward, streaming them can be quite complex depending on which option is chosen:

1) **Progressive downloading** is a method where clients can download video files over HTTP and TCP, allowing media players to start the video playback once part of the file is downloaded, thus giving the impression of streaming [47] [48]. The downloaded content is then stored locally, which makes this option less secure [47] [48]. Key benefits are that it is easy to set up, it requires no special server requirements and that the quality of video is maintained although there could be delays based on the network bandwidth [47].
2) **Streaming** is an alternative approach, in which the video is being delivered over RTMP or RTSP using a streaming server [47] [48]. It is a chunk based delivery mechanism, allowing media players to instantly consume content without any local caching [47]. Furthermore, it can adopt transfer speeds to improve user experience as

[20] https://developers.google.com/places/ios-api/autocomplete

[21] https://developers.google.com/youtube/v3/

[22] https://developers.gfycat.com/api/

it uses the UDP protocol for delivering packets [47]. By downloading content just in time, it also saves bandwidth and users can jump back and forth in the video timeline [47].

3) **Adaptive Streaming** is the "most popular form of video streaming with benefits of quality switching and ease of delivery over HTTP" [48]. Depending on the connection bandwidth, the server automatically adjusts the stream's bitrate by sending lower or higher data-rate contents, hence ensuring a continuous streaming experience [47] [48]. Current adaptive streaming technologies are Adobe's HDS, Apple's HTTP Live Streaming and Microsoft's Smooth Streaming [47].

Given the scope of this project, I went for the progressive downloading option, simply because it requires the least effort and time. For production purposes, I will of course have to switch to either a pure streaming or adaptive streaming technology, since they are clearly more efficient and user-friendly.

3 Requirements & Analysis

3.1 Requirements Derivation

Given the initial problem statement described in 1.1 and its extensive elaboration in 2.2, I firstly derived a set of high-level functionalities that would solve identified challenges:

Identified Problems	Solutions
Written reviews can be misleading, corrupt and highly damaging to a business.	Videos are more genuine and easier to consume, allowing users to quickly form their own opinions about businesses.
The information users are looking for in reviews tend to be hidden within the text, thus forcing users to browse through irrelevant content.	Users need to easily navigate between videos with the possibility of narrowing them down to more specific content.
Written reviews tend be lengthy and tedious to read.	Videos should be kept short, encouraging users to record relevant footage only.
Users can be easily misled by ratings.	Ratings should not be supported or encouraged, shifting the priority to positive user experiences only.
Users need to easily discover and find new places.	Users need to be provided with various searching capabilities, enabling them to find venues by custom search terms, business categories or predefined location bounds.
Users might not be able to find every business.	Users should be able to add new venues in case they do not find them.
Malicious users might write overly offensive reviews.	Users should be able to report on videos, facilitating the Voxreel team in identifying irritating content.

This represented the groundwork for my requirements elicitation process, leading to the described requirements and use cases in the subsequent two chapters.

3.2 Requirements

The requirements list has been broken down into two sections: one for the iOS application the other for the RESTful API components of the system. Each section has a considerable list of functional and non-functional requirements developed from the elicited, high-level solutions and project goals in the requirements derivation process. Similar requirements are grouped together into categories in order to provide the reader with a better overview about the system's key features. Furthermore, each functional or non-functional requirement was prioritised using the MoSCoW prioritisation technique, which uses the following categories to distinguish between the importance amongst requirements: [49]

Category	Description
M – MUST	This is a requirement of highest priority and must be completed.
S – SHOULD	This is a requirement that also has high priority and hence should be implemented.
C – COULD	This is a requirement which could be completed but is not necessary.

W – WON'T	This is a requirement that will not be implemented though it may be considered for future development.

Refer to *Appendix 8.1* for the full requirements document.

3.3 Use Cases

Figure 3.1 shows the use case diagram for the iOS application component of the project.

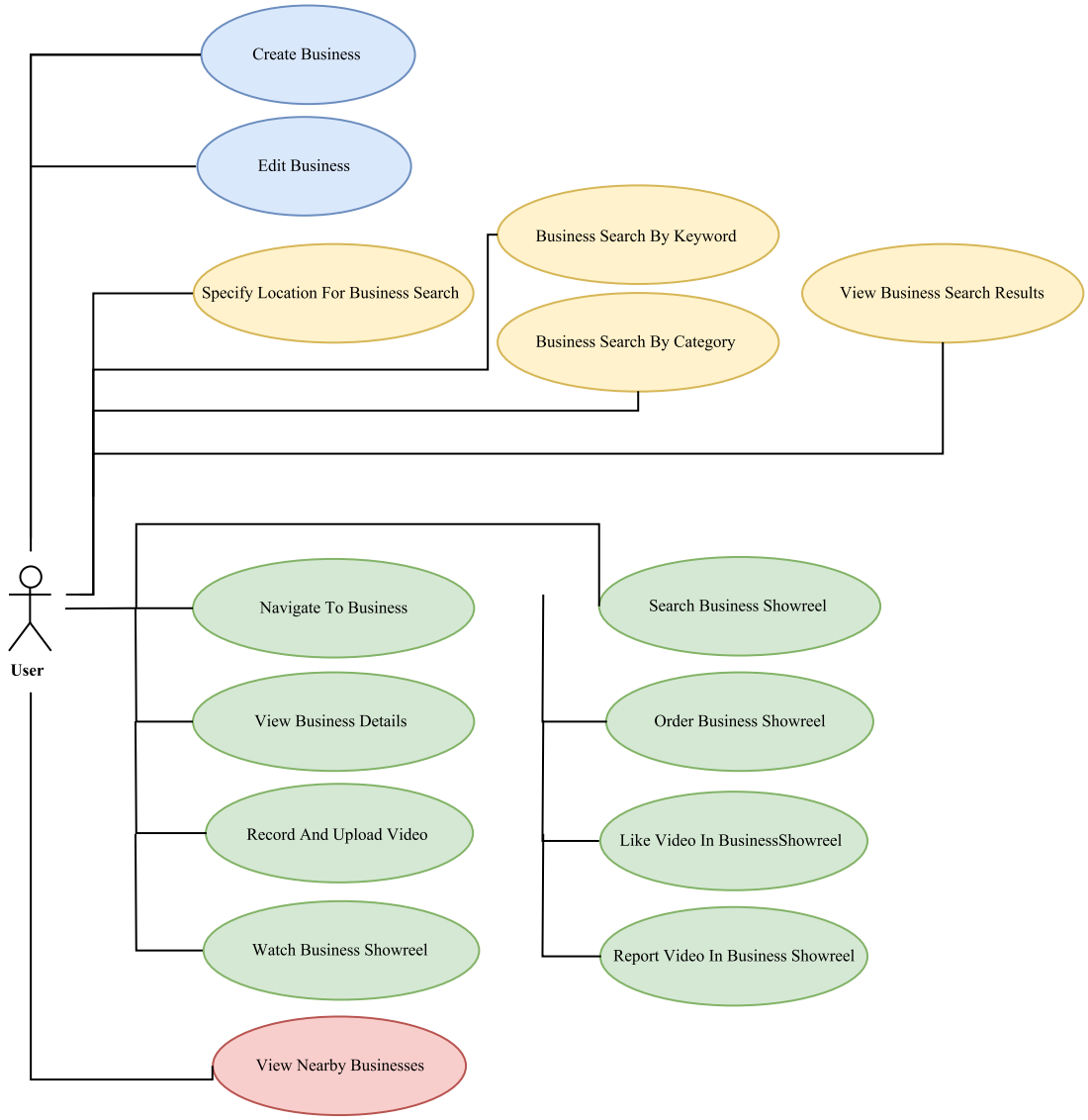

Figure 3.1 - Use Case Diagram

For clarity, I specifically denoted related use cases by the same background colour. Also, note that the users cannot directly interact with the RESTful API, hence there are no separate use cases for that part of the system. However, the listed use cases do mention the RESTful API when the iOS application needs to communicate with the backend. The full use case document can be seen in Appendix 8.2.

3.4 Entity-Relationship (ER) Diagram

In software engineering, ER diagrams are commonly used to represent the information that needs to be stored by an application, demonstrated by entity types and relationships between instances of those entity types [50]. It is hence

a fundamental data model, illustrating the information storage structure that can be later on used to implement my project's relational database [50].

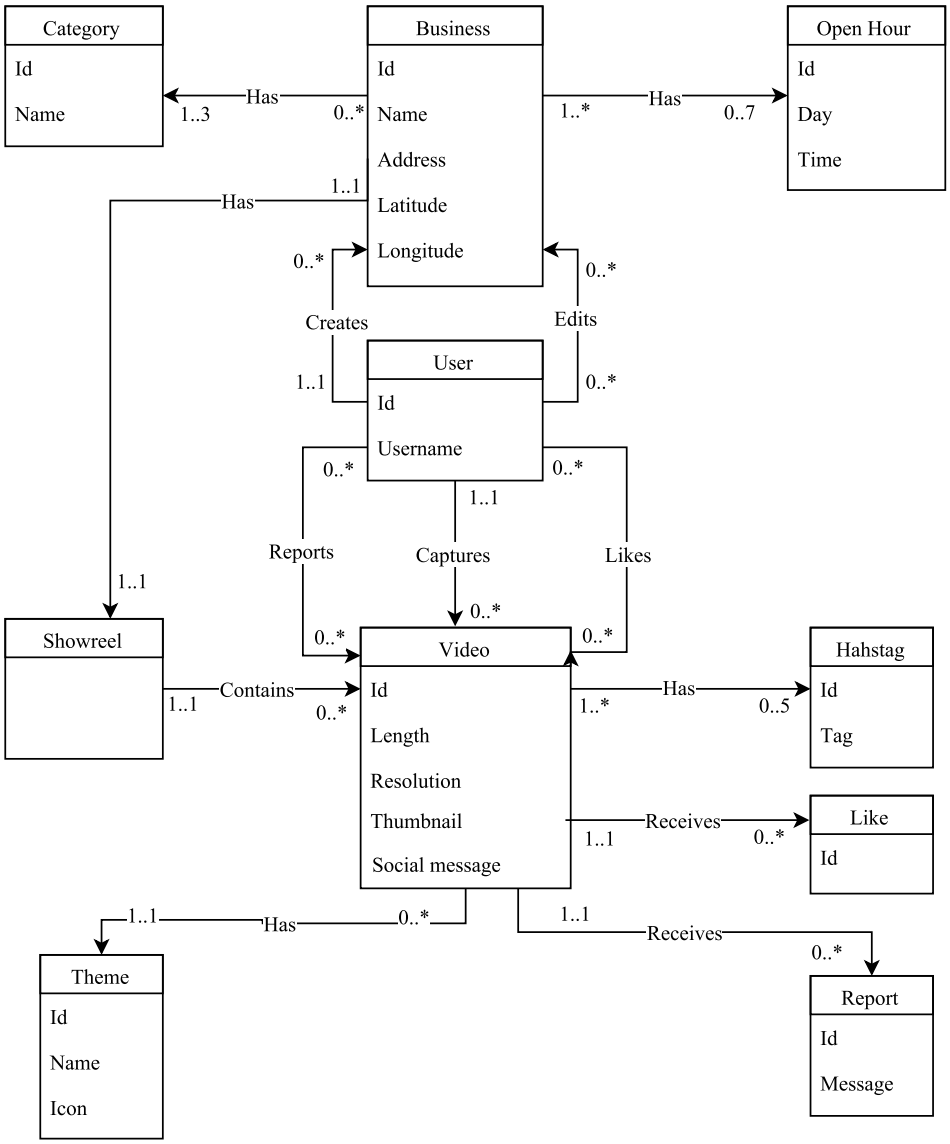

Figure 3.2 - ER Diagram

Figure 3.2 shows the ER diagram resulting from the requirements analysis in the previous two chapters. As demonstrated in the model, a user can create zero or multiple new businesses, each containing basic information that must be provided at its creation. Additionally, a business has between one and three predefined categories. Note that the categories are being reused, hence they do not necessarily belong to just one single business (or might not belong to any business). A business can also have open times associated with it, depending on whether a user has provided them when creating or editing the business. Similar to the business categories, open times can be shared by multiple businesses, as they might have similar working hours (they must belong to at least one business). Furthermore, a business has a showreel consisting of zero or multiple videos.

A user can further record zero or multiple videos for an existing business. Each video has exactly one predefined theme. Again, videos can share the same themes, hence they do not belong to just one single video (might not be used for any video). Apart from that, a user can annotate the video with zero or up to five searchable hashtags.

Lastly, a user can like and report videos in a business's showreels. Consequently, each video can have zero or multiple likes/reports.

4 Design & Implementation

4.1 System Design

In this chapter, I will discuss any design guidelines that were followed by the iOS application and RESTful API. In particular, I will highlight the most notable design patterns/strategies applied.

4.1.1 iOS Application

MVC Design Pattern: The client application was structured using the MVC design pattern. Each application page was assigned to a custom controller responsible for managing the data flow into the corresponding model objects and updating the views on the screen accordingly to any data changes. Normally each page would make use of various self-customized view classes depending on its complexity and design requirements. View classes were programmed generically in order to make them reusable for multiple pages, thus enforcing a general design theme throughout the app. Model types were usually defined as structures if they were used to hold simple data values or preferred to be copied rather than referenced. Otherwise, they would be defined as classes. Also, if the model type inherited from another class, then it had to be defined as a class since structures in Swift do not conform to inheritance relationships.

Decorator Design Pattern: This design pattern allowed me to add behaviour to individual objects without modifying their classes [51]. It represents an alternative approach to traditional subclassing and was often applied in situations where subclassing was not feasible or would have resulted in messy subclass implementations. In Swift, there are two common methodologies in which this pattern can be applied:

(1) *Extensions:* This mechanism aided me in adding new functionalities to existing Apple classes, structures or enumeration types. This turned out to be extremely powerful as any defined extension could be used throughout the entire project.

```
extension String
{
    static func random( _ length: Int = 20 ) -> String
    {
        let base = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"
        var randomString: String = ""

        for _ in 0..<length
        {
            let randomValue = arc4random_uniform( UInt32( base.characters.count ) )
            randomString += "\( base[ base.characters.index(base.startIndex, offsetBy: Int( randomValue )) ] )"
        }

        return randomString
    }
}
```

Figure 4.1.1 - Example of extending Apple's String structure with a static method for generating random strings (was used for creating IDs for objects)

(2) Delegation: This mechanism was extensively used throughout the entire app development, allowing objects to respond to occurrences of particular actions or to retrieve data from external sources without knowing their underlying structure. The pattern is implemented using an interface that represents the delegated responsibilities and a class that conforms to that interface and provides the required functionalities. As a result, a class could delegate some of its responsibilities to another class without the need for creating a strong coupling relationship between them.

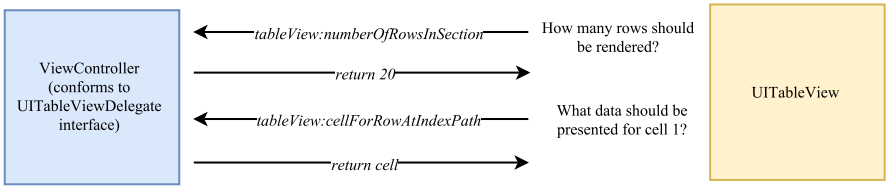

Figure 4.1.2 – Example of how a controller would communicate with a UITableView (Apple's UI class). For demonstration purposes only two interaction scenarios are listed here.

Fail-safe programming via Optionals: In Swift variables that may or may not hold data or a reference to an object are always marked as optional. This is a very powerful feature as it requires the programmer to unwrap the optional variable first to check if it is empty (also referred to "nil") before actually using the variable. As a result, this leads to much safer code and avoids passing around nil variables that could potentially crash the application. Even though there is the overhead of unwrapping the optionals, they enabled me in developing more robust and fail-safe code.

```
// Get selected location
guard let location = self.searchTracker.getLocation( index: index ) else
{
    print( "ERROR: There does not exists a location with the index \(index)." )
    return
}
```

Figure 4.1.3 - This is a code snippet from a method in which I first check if the location for the given index does indeed exist, i.e. is not nil, before I further use it. If the location cannot be unwrapped into the location variable, an error is printed to the console and the function returns immediately without crashing the program.

Service classes: Aside from model, view, and controller classes resulting from the MVC design pattern, several service classes had to be developed to fill the missing gaps in the system design. Depending on their provided functionalities, they were either defined as static or instantiated as singletons. For instance, the "FileManager" or "APIManager" classes are used within multiple parts of the iOS application (see both classes in Appendix 8.7). However, since they do not require data storage, such classes were defined as static in order to prevent any object creation overhead. In contrast, the "AppCache" class is declared as a singleton object since it is responsible for storing nearby businesses which have to be accessible from multiple locations in the source code. It is important to note though that singletons were kept to a bare minimum to prevent the overhead of securely managing concurrent behaviours.

iOS Conventions: As I anticipate to submit the application to the App Store at some point, I additionally focused on following Apple's specified iOS coding conventions [52] and human interface guidelines [53]. With this, I hope to later on accelerate my app submission process as well as ease the navigation through my code in case other iOS developers have to review it.

4.1.2 RESTful API

MVC Design Pattern: Similar to the iOS application, I followed the MVC pattern to structure my backend code. Each API endpoint is represented by a single controller with public methods for performing various HTTP methods (POST, GET, PUT, PATCH or DELETE) on the corresponding URI resource. Models on the other hand represent database tables used to interact with that table. As a result, controllers would query data from the database via the predefined model operators without having to know the underlying query structure. Consequently, there are no dependencies amongst controller and model classes, resulting in a highly decoupled and more flexible backend architecture. Even though there are technically no views involved in an API, all endpoint responses were transformed to user-friendly and standardized format output using transformer classes. As their sole purpose is to format the API outputs, they can be seen as the missing view component of the MVC pattern.

Repository Design Pattern: The use of this pattern enabled the strict separation of data access from business logic on top of the traditional database models. This was crucial for creating thin and compact controller classes, as any complex database queries were nicely hidden away in their corresponding repository classes. The logic defined in the controllers therefore could simply retrieve data objects, i.e. models, using the repository methods without the need to know about their implementation. Consequently, if at any time the backend's database system needs to change, required changes are limited to its database access logic only.

Request Validators: Instead of tediously validating request parameters inside controller methods, I created dedicated request validator classes that extracted out any validation logic. This not only kept the controllers small but more importantly the same validation mechanisms could be reused for endpoints the same request parameters. Furthermore, invalid parameters were immediately communicated back to the API user, avoiding the overhead of propagating the request to the controller. Again, this resulted in a more flexible and scalable backend solution.

Middleware Layers: On top of request validators, I also implemented middleware classes to filter and prepare HTTP requests entering the API endpoints. This way for instance, obfuscated business IDs could be first translated into true business IDs before the request parameters were further passed onto the corresponding request validator. Subsequently, I could avoid duplicating shared middleware mechanisms into multiple request validators, thus reinforcing a better separation between request related API components.

4.1.3 General Conventions

Single responsibility principle: The previously mentioned design patterns in 4.1.1 and 4.1.2 already heavily focused on separating component duties. However, I want to readdress the importance of the single responsibility principle, which was applied on a class and method level, thus avoiding multiple hidden responsibilities. Each class and method was designed to fit its purpose, focusing on weak coupling and strong cohesion to avoid "god elements" and "spaghetti" architectures. This not only resulted this in easier software testing but also prepared the client and server applications to be scalable and flexible for the future development iterations.

4.2 System Architecture

The system architecture can be divided into two main components, the iOS application and the RESTful API running on a webserver. Figure 4.2.1 represents this structure.

The RESTful API is shown at the top, demonstrating the three main controller components (denoted by rectangles with round corners) and their provided services (denoted by rectangles with pointed corners and number labels at the top). Each of those services is operating upon various stored data as shown by the double directed arrow between the API box and the webserver database. To obtain a deeper overview on how each of the services can reached, see the "routes.php" file in Appendix 8.7. Lastly, note that low level implementation details concerning the used request validators, middlewares, transformers or repository services as mentioned in 4.1 are not specified in the diagram.

The iOS application is shown at the bottom, illustrating how the user can transition between the various application pages (denoted by rectangles with round corners). Each page box is additionally marked with the specific controller class name it is controlled by alongside the numbers of consumed API services. Note that the Google API, represented by an ellipse, is the only third party API used, facilitating the location autocompletion process as part of the business search as I will further discuss in 4.3.2. Also, any arrows from a page box to one of the two storage components (denoted by cylinders) or vice versa are used to emphasise on what data, either persistent or cached, they operate upon. Further information about the data storages' specific purposes, is elaborated on in 4.3.1.1 and 4.3.3.3 respectively.

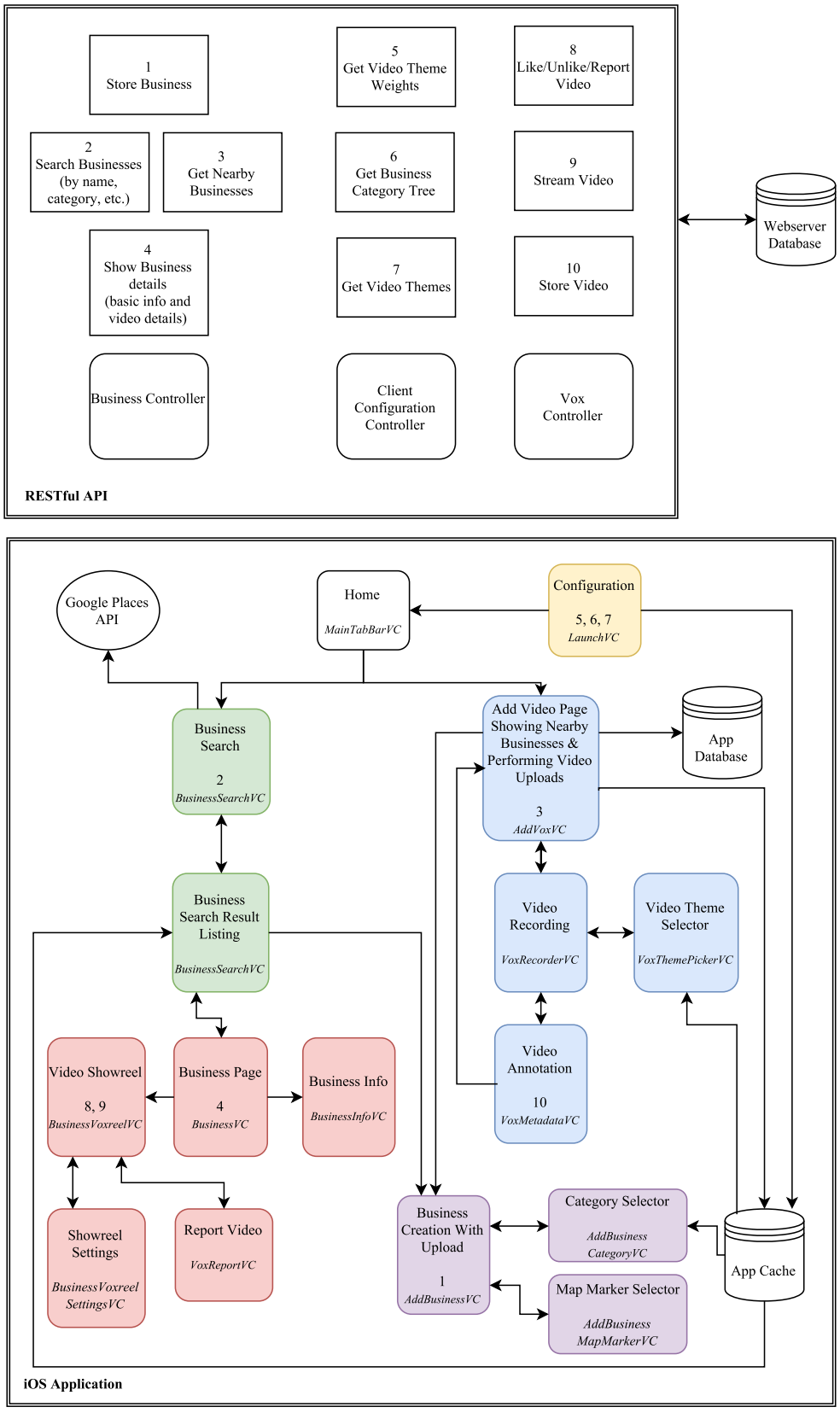

Figure 4.2.1 - System Architecture Overview

I purposely portioned the iOS application components into five distinct groups as illustrated by their background colours. Page boxes with the same background colours share related functionalities and serve together as a primary application feature. Throughout the subsequent subchapters in 4.3, I will discuss each of these main components in greater depth, providing descriptions, database/storage representations and any noteworthy implementation details. Furthermore, the actual webserver database schema will be elaborated on in an additional subsection at the end.

4.3 Application Components

4.3.1 Flexible Configuration

The configuration process is the initial procedure when the iOS application boots up, downloading content via the RESTful API to configure the app and prepare it for the user. As this process involves performing several API calls at the same time, I decided to wrap each of them into separate dispatch groups which run concurrently, thereby avoiding long launching times when running them serially. The dispatch groups work together similar to a traditional synchronisation barrier, meaning all dispatch groups have to finish first until the launching process can reach its end and pass on the execution to the application's home page.

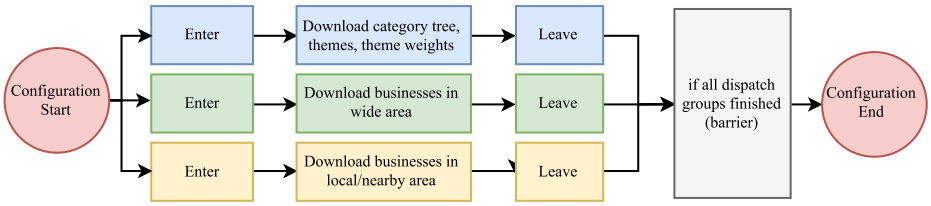

Figure 4.3.1 – Illustration of the three dispatch groups executing concurrently, breaking the barrier once they all finished.

In total, there three dispatch groups concurrently executing; one for downloading the business category tree, video themes and video theme weights, and the other two for downloading businesses in the wide and nearby area. All these procedures are highly flexible from the API's perspective since any content changes will be immediately reflected on the client when launching the app another time, thereby circumventing the overhead of programmatically storing any configuration data. Before going into further details in regards to specific downloaded contents, I will next discuss the app cache used for storing all that data.

4.3.1.1 App Cache

There are three main options for implementing a caching mechanism in iOS. See the table below, contrasting each option with its pros and cons:

Core Data (Apple's own app database solution using SQLite)	Files using pdflist format (Apple's own file storage solution)	NSCache (Apple's own cache class)
+ Can be used as an in-memory store + Allows more complex data modelling with built in querying facilities - Relatively complex to implement (needs learning curve)	+ Easy to use + Low effort - Uses file system I/O operations (low performance) - Not flexible for changes	+ Easy to use + Low effort - Cached data is easily lost, especially when the app goes background

It is important to note that the configuration data is used by multiple application components (possibly simultaneously) and requires fast access without much noticeable delay. Consequently, using "pdflists" was ruled out for its greater access time. The "Core Data" option seemed in theory to be the right choice as it would have allowed the app to store configuration data at the phone without persistently downloading content at the launch of the application. However, this would have required unnecessary effort at the time and added additional complexity to the project, hence the reason why I abandoned it. The "NSCache" option was the option I ended up pursuing. Despite numerous failed attempts to prevent the system from evicting data from my NSCache extended object when the app would enter the background, I finally chose to implement my own cache applying the singleton design pattern. The most difficult part was to make it thread safe, allowing multiple threads to read it simultaneously but enforcing writes to behave in a synchronized manner.

```swift
/// Represents the cache storage of the application. It is a singleton.
final class AppCache
{
    /// One globablly shared AppCache object.
    static let shared = AppCache()

    /// This class is a singleton.
    private init() {}

    /// Stores all nearby businesses.
    private var cachedNearbyBusinesses = ThreadSafeStorage< [ Business ] >()

    /// Stores all businesses around the user (less narrow then nearby businesses).
    private var cachedAroundBusinesses = ThreadSafeStorage< [ Business ] >()

    /// Stores the business category tree.
    private var cachedBusinessCategoryTree = ThreadSafeStorage< BusinessCategoryTree >()

    /// Stores vox themes
    private var cachedVoxThemes = ThreadSafeStorage< [ VoxTheme ] >()

    /// Stores vox theme weights
    private var cachedVoxThemeWeights = ThreadSafeStorage< [ VoxThemeWeight ] >()
```

Figure 4.3.2 - This is a screenshot from the top part of the "AppCache" class. It uses the generic "ThreadSafeStorage" class for storing the various configuration data. See both classes in Appendix 8.7.

4.3.1.2 Category Tree Using The Nested Set Model

The business category tree is a fundamental application component used for selecting categories when uploading a new business or searching businesses by category. More importantly, it is stored on the webserver database using the nested set model (via the "Baum" library as discussed in 2.4.2), a common technique for representing hierarchical data structures such as trees in a relational database [29] [30].

The nested set model is very powerful as its stores the tree nodes in a very specific way. Each node is numbered twice according to a tree traversal, resulting in assigning numbers in the order of visiting. Figure 4.3.3 represents the category tree as a nested set model. Note that the numbers when entering a set are represented as left IDs, whereas the numbers when leaving a set are illustrated as right IDs (see 4.3.6 for the actual database schema).

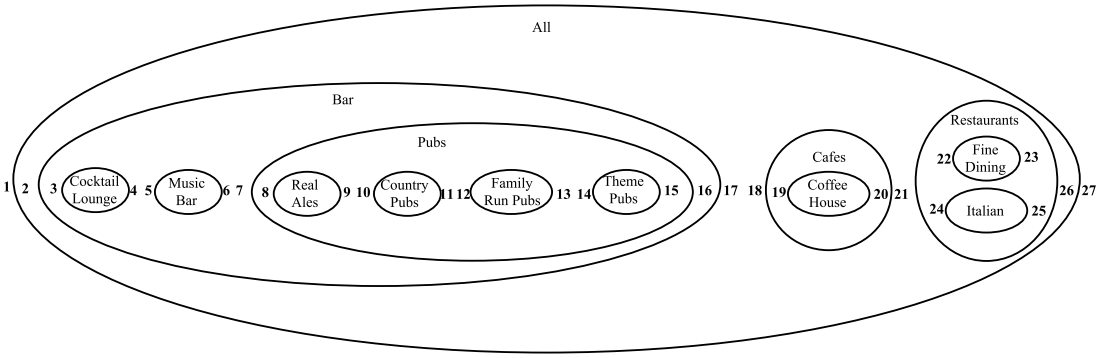

Figure 4.3.3 – This is the current business category tree represented as a nested set model.

By comparing left and right IDs, category relationships can be inferred, thus making it feasible to build the tree on the client side recursively (see "BusinessCategoryTree" class in Appendix 8.7). Consequently, the API only has to send the list of categories, saving server capacities as it does not have to compute the tree for the client. The currently implemented category tree is three levels deep including only basic categories. However, due to the use of the nested set model, it can be adapted to any shape if required, without having to perform any additional changes to the client side code.

4.3.1.3 Video Themes

Video themes are used for picking the appropriate themes when recording new videos as well as filtering businesses' showreels by theme types. Each video theme is represented by its own icon and URL. The latter was required in order to avoid the overhead of storing theme icons locally. They can instead be accessed and displayed via a URL, set up as a symbolic link to the backend's public image storage. This way, icon files do not have to be manually served, thus avoiding any performance penalty which would be otherwise caused by going through the entire API

request and response lifecycle. Hence, the webserver database can be easily extended with new themes, without the need for updating the actual iOS application.

4.3.1.4 Video Theme Weights

Video theme weights are used to map video themes with the business categories that make sense and provide each association with a weight. For instance, a business marked with the "restaurant" and "bar" category should allow users to record "food" and "drinks" themed videos, however not "art" themed ones, since there will not be any mapping from "art to restaurant" or "art to bar". The associated weights in all the mappings are later on used to determine the order in which themes appear on the theme picker page. Again, this approach is highly flexible, as theme weights can be adapted and extended any time via the webserver database.

4.3.1.5 Wide Area & Nearby Businesses

Both wide area and nearby businesses are requested based on the user's latitude and longitude position, however they provide different services. Wide area businesses are used for performing "current location" business searches with the goal to increase the overall system performance by decreasing the server load (see 4.3.2 for further details). Nearby businesses on the other side are used by the add video page, serving as those businesses the user is close enough to record videos for (see 4.3.3 for further details).

I specifically kept both endpoints separate as nearby businesses should be updated more frequently compared to the businesses in the wider area, assuming the user moves around. Currently they are only requested once at launch time. Note however that an appropriate location tracking class ("LocationTracker" class) already exists, which can be used to update cached wide area and nearby businesses as the user position changes (this was not considered of high importance for this project). Lastly, there are a few noteworthy differences when it comes the implementation of the two API endpoints:

Endpoint for querying wide area businesses	Endpoint for querying nearby businesses
Creates a squared bounding box given a user's lat/long position (5km x 5km) and includes all businesses inside that box.	Creates a small bounding circle given a user's lat/long position (0.5 km radius), includes all the business inside that circle, and sorts them in ascending order based on the distance to the user's location.
The ordering is not necessary.	The ordering is necessary as the closest business should be listed first as part of the add video page, thus preventing computation overhead on the client.
Since the distances to the businesses are irrelevant and do not have to be compared, this endpoint performs very quickly for searching several thousand businesses within the box.	Performs quickly for a small radius including a couple hundred businesses. For a large radius (~2km or more) it can be fairly slow due to the distance calculation overhead.

4.3.2 Business Search

It is important to note that the business search page is managed as part of the "BusinessSearchVC" controller and the "BusinessSearchTracker" class (see "BusinessSearchTracker" class in Appendix 8.7). The latter is required in order to maintain the complex states throughout the search. As a result, this minimizes the size of the controller and enforces the single-responsibility principle.

The first step in performing a search is to specify a location. By default, users have the choice to select from a list of predefined options as illustrated in figure 4.3.5, allowing them to quickly pick a location without having to manually specify it. The application automatically disables/hides the "Current Location" option in case users have turned off their location tracking services. Moreover, it periodically tracks the user's location to ensure accurate search results. To save battery power, those updates only occur as long as the application runs in the foreground and the user's location changes noticeably. Figure 4.3.6 demonstrates a user manually entering a location and the autocomplete results that are suggested to him/her. Those are provided via the Google places API, which by default returns the top 5 matching locations including their location bounds (see "GooglePlacesAPI" class in Appendix 8.7).

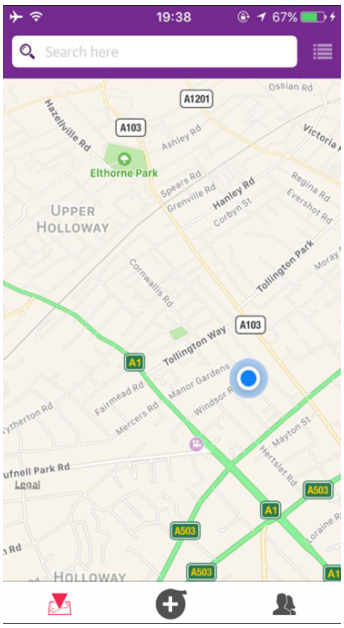

Figure 4.3.4 – Initial business search page

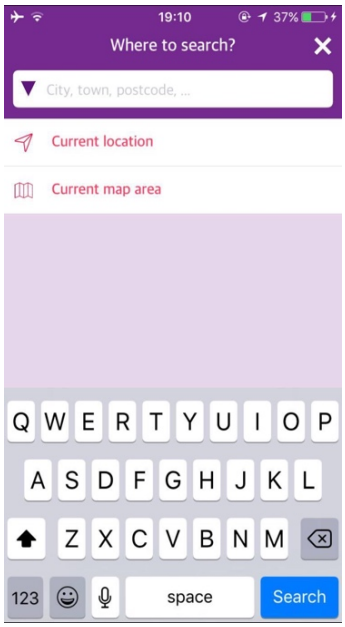

Figure 4.3.5 - Default locations

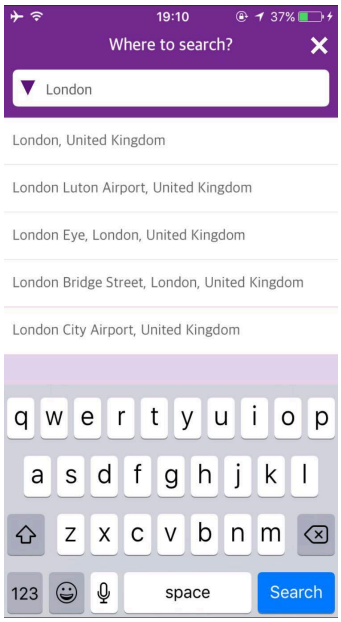

Figure 4.3.6 - Location suggestions provided by the Google Places API

After selecting one of the suggested locations, users are prompted to specify what they are looking for. This can either be categories they are interested in or the names of the businesses they are trying to find. Similarly to the location specification process, users can select from a list of default categories (level 1 categories taken from the cached category tree) to trigger a search by category immediately, thus increasing the user experience (see figure 4.3.7). Otherwise, users can specify their custom search strings. Throughout this entering process, the autocompletion list will adapt accordingly, showing the top matching categories and businesses within the specified location (see figure 4.3.8). At this point, users can either chose to (1) select a business from the autocompletion list to visit its page, (2)

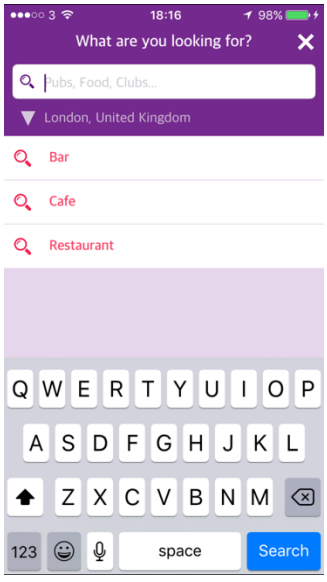

Figure 4.3.7 – Default categories

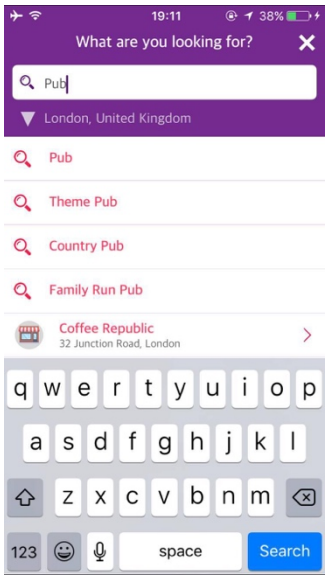

Figure 4.3.8 – Autocompletion list for "Pub"

select the “Search” button and illustrate all found businesses on a map/list view (see figure 4.3.12 and 4.3.13) or (3) select a category from the autocompletion list and perform a business search by category.

Although, it might not be apparent from a user point of view, there are several search algorithms in place, based on the specified location of the search.

4.3.2.1 Search Algorithms

Figure 4.3.9 demonstrates the high-level execution flow of how the various business search types are performed. Note that unnecessary, looping steps have been omitted.

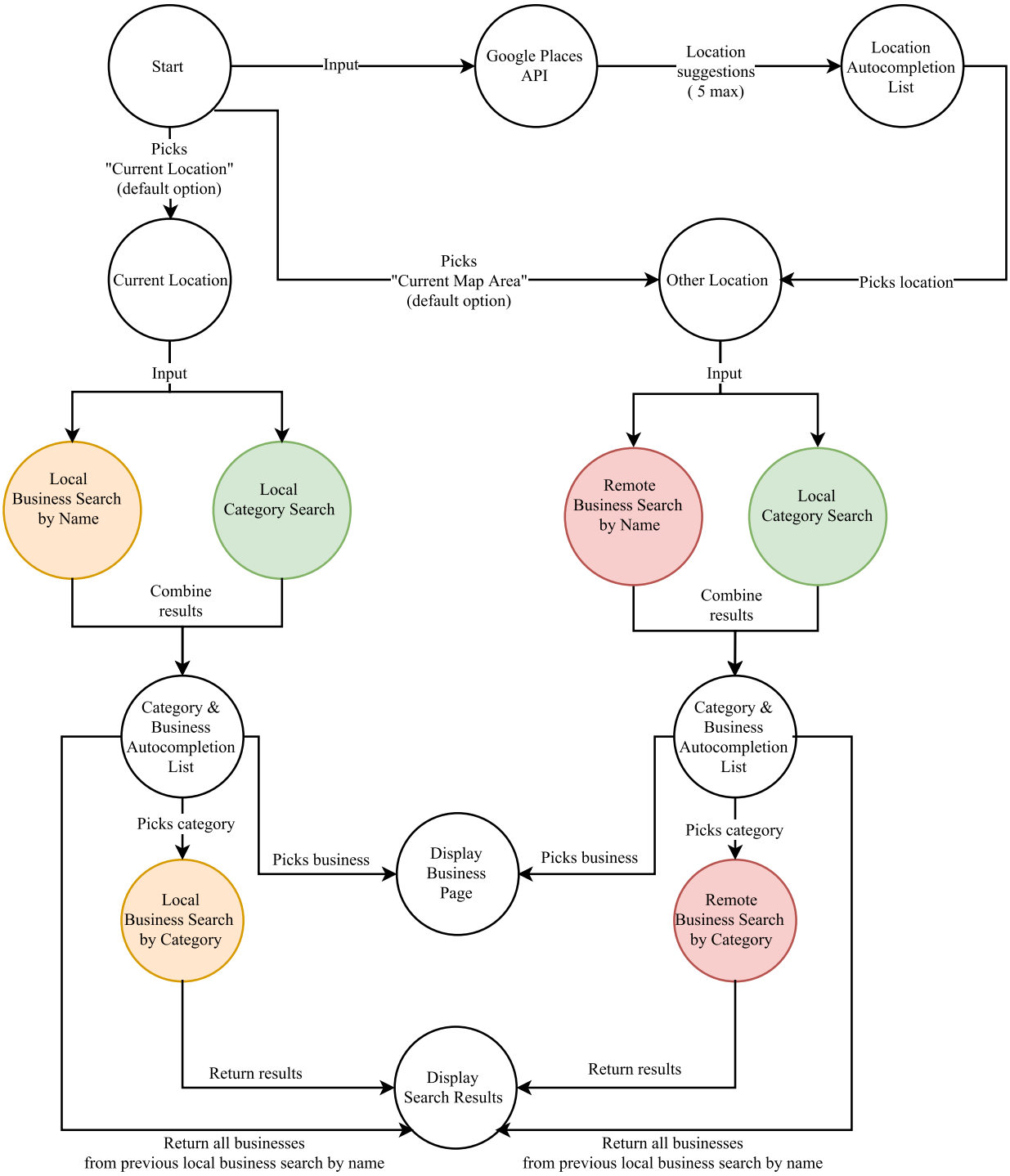

Figure 4.3.9 – High level execution flow of the various searching types.

Local Business Searches (highlighted in orange): Any business searches specified with "Current Location" are performed locally on the client using the cached wide area businesses. Even though this adds the overhead of implementing two additional search algorithms, it decreases the server load, hence saving previous server cost in the future and preventing long waiting times caused by network latencies.

Local business search by name	Local business search by category
It is implemented via a fuzzy string search on the cached wide area businesses.	It is implemented via a category checking algorithm using the cached category tree. Each cached wide area business is checked for whether it has the same specified category or a subcategory of the specified category.
It is performed as the user enters his/her custom search string as part of the autocompletion process. When the user decides to select the "Search" button, the search results are already available for display.	It is performed once the user selects a category from the autocompletion list. Thus, the user has to wait for its completion before any results can be displayed.
Returns the top 40 results (those with the best match)	Returns the first 40 results

Remote Business Searches (highlighted in red): Any business searches other than specified with "Current Location" are performed via the RESTful API (see "BusinessController" and "BusinessRepository" in Appendix 8.7). This is necessary as the specified location might not be covered by the cached wide area businesses. For instance, a user might be situated in central London. As a result, he/she will have all London businesses (i.e. wide area businesses) cached on his/her phone, allowing any "Current Location" searches to be performed locally. Nonetheless, if the user decides to specify "Manchester", any subsequent searches have to be computed via the API as such locations are out of the client's scope. A further optimisation step would be to first check whether a specified location is within the bounds of the cached wide area businesses before performing any remote searches. However, this would have cost me precious development time, therefore I did not follow up this approach.

Remote business search by name	Remote business search by category
Requires a bounding box as its parameter	Requires a bounding box and a category id as its parameters
It is implemented via a fuzzy string search on the businesses within the specified bounding box.	It is implemented via category checking algorithm on the businesses within the specified bounding box. Each relevant business is checked whether it has the same specified category or a subcategory of the specified category.
Returns the top 40 results (those with the best match)	Returns the first 40 results

Local Category Searches (highlighted in green): Regardless of whether a local or remote business search by name is performed when a user enters his/her custom search string, the application additionally performs a fuzzy string search on the cached category tree, whose category results are reflected within the autocompletion list together with the business search results. This is crucial as users might be looking for searchable categories.

4.3.2.2 Fuzzy String Search

As previously mentioned, the fuzzy search is a fundamental component for category as well as local and remote business name searches. The fuzzy search algorithms on both the client and the backend side are logically identical,

however differently implemented as a result of their languages, i.e. Swift vs PHP. Each developed fuzzy search is based on the Levenshtein distance algorithm, which informally calculates the minimum number of character replacements (insertions or deletions) required to transform one string to another, also referred to as Levenshtein or edit distance. If two strings are equal the Levenshtein distance is 0, a higher value indicates that the strings are different.

As illustrated in figure 4.3.10, with each user input, edit distances would be calculated separately for the set of relevant businesses and set of categories, each ordered from low to high (only for inputs with two characters or more in order to avoid long computations). During this process, a sensible threshold distance would be specified to filter out irrelevant search results.

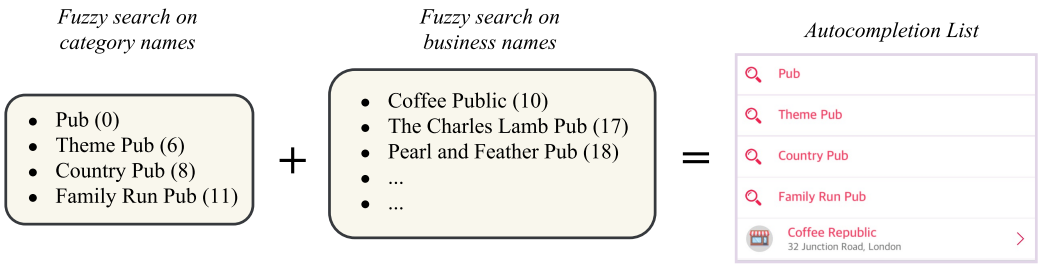

Figure 4.3.10 – This illustrates how the autocompletion list is formed given the user input "Pub" with "London, United Kingdom" as the specified location. Also note that the number next to each list item represents its edit distance and the results from the business search by name (fuzzy search on business names) are kept in memory in case the user choses to illustrate them on a map/list view in the following step.

4.3.2.3 Debounced Search

To avoid unnecessary network traffic and search computations, each search request performed during the autocompletion process is debounced by 300 milliseconds. This enforces that a search request is not to be called again until a certain amount of time has passed. Consequently, if a user enters an input really fast, only the last search request is executed while the others invalidate each other.

To achieve debouncing, search requests have to share a timer, which is updated before each request goes into its waiting mode. Once a search request has been successfully delayed and its execution can presume, it will only be executed if the shared timer plus the delay time (300 milliseconds) are less than or equal to the current time, guaranteeing that the timer has not been invalidated in the meantime. Note though that each search request is guaranteed to become active shortly after 300 milliseconds due to the fact that the timer check is being performed on the main thread, which has the highest scheduling priority and is usually used for rapid UI updates. After a request has passed this check, it is finally executed on a high priority background thread, to keep the main thread free for any UI interactions. Figure 4.3.11 demonstrates an example of a debouncing behaviour.

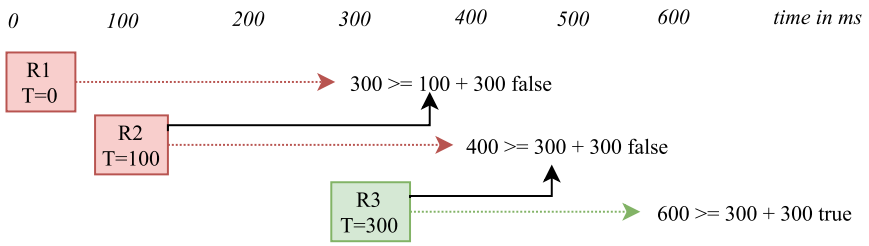

Figure 4.3.11 – Three search requests R1, R2 and R3 are triggered, each updating the shared timer T. Since R2 invalidates R1 and R3 invalidates R2, only R3 is being executed in a background thread. Refer to Appendix 8.7 for the actual debouncing implementation.

4.3.2.4 Out Of Order HTTP Responses

When performing API search requests during the autocompletion process, it can be the case that their responses arrive out of order. Without any additional mechanisms, responses from previous search requests might therefore be able to override responses from newer search requests, thus incorrectly updating the autocompletion list. This usually occurs due to changing search performances on the server side. To overcome this obstacle, search requests are associated with increasing IDs. Hence, when the API responds with the results for a particular search request, only if its ID is greater than the IDs already seen it will be accepted.

4.3.2.5 Enhanced Usability Features

To increase the user experience, the iOS application offers both a map (figure 4.3.12) and list view (figure 4.3.13) for showing search results, allowing the user to alternate between them.

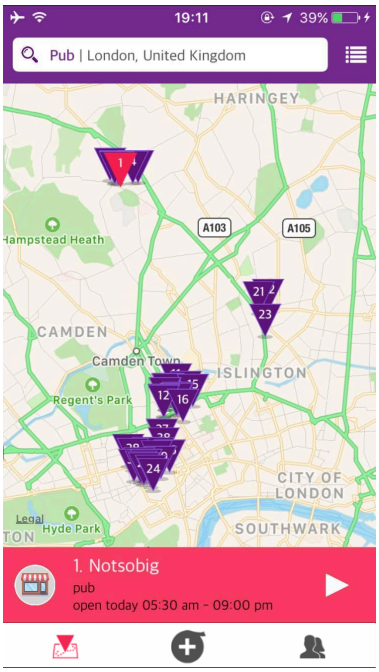

Figure 4.3.12 - Search results in map view. A user can swipe left and right between the businesses (within the bottom red box) or select them via their map markers.

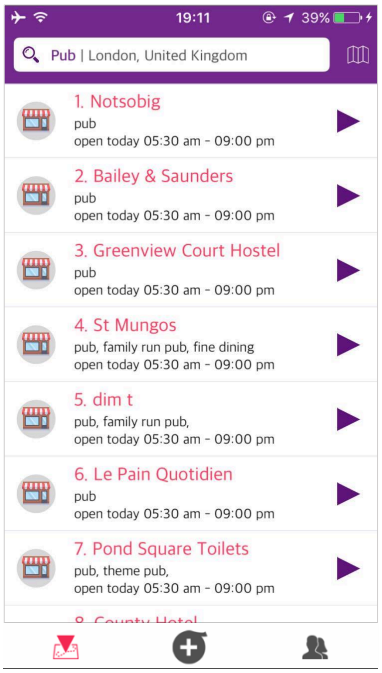

Figure 4.3.13 - Search results in list view. A user can scroll up and down the list.

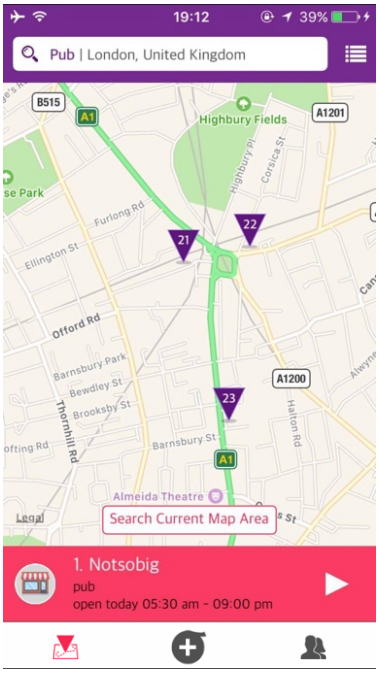

Figure 4.3.14 – A "Search Current Map Area" button appears once the user changes the map view.

Furthermore, it is oftentimes necessary to change the location bounds of the current search in order to narrow/widen the search results. To avoid the overhead of manually specifying a more/less specific location, a user can adapt his/her map view and query his/her resulting map area with the previously used search criteria (figure 4.3.14). Note though that this will change the prior specified search location to "Current Map Area".

4.3.3 Video Capturing

As already mentioned in 4.3.1, based on the user's location, nearby businesses are cached as part of the configuration process. They represent those venues users are allowed to capture videos for, thus restricting malicious ones from producing disruptive content to any random business they can find within the application. Figure 4.3.15 demonstrates an example of an add video page. To increase usability in scenarios where there might be many businesses situated around a user, the application provides a feature of further narrowing down the business list via the same fuzzy search algorithm mentioned in 4.3.2.2.

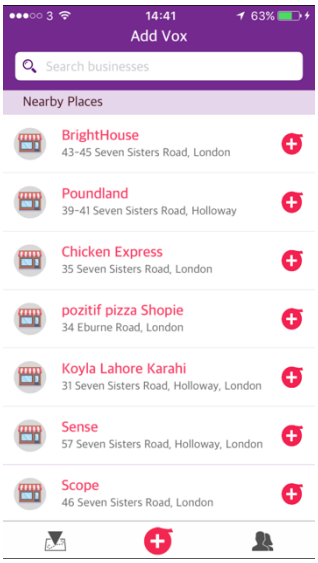

Figure 4.3.15 - Add video page

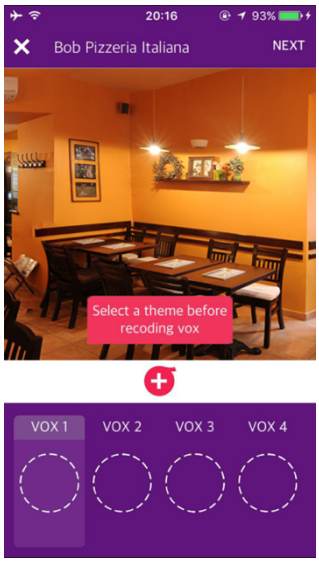

Figure 4.3.16 - Initial video recorder

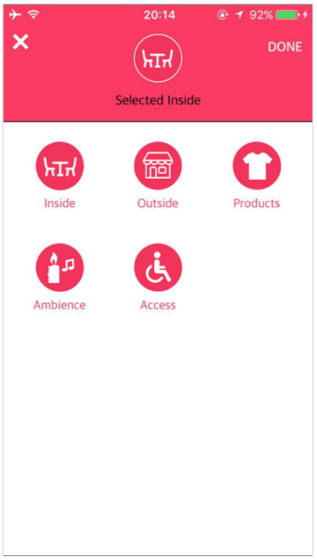

Figure 4.3.17 - Theme selector

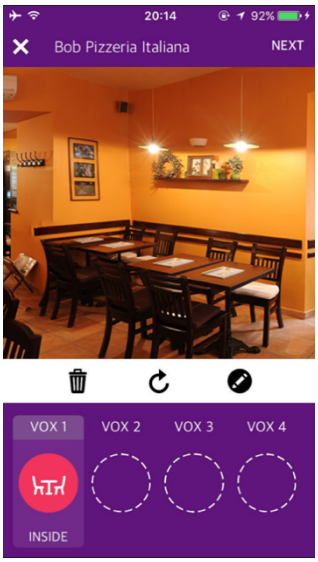

Figure 4.3.18 - Recorded video preview

After a venue is selected from the nearby businesses listing, the video recorder will present itself (figure 4.3.16). Initially, only one video could be captured as part of each recording session. However, to provide a better user experience this number was adapted to four videos since a greater number would have overcomplicated the UI design and hindered the uploading process. Moreover, videos can only be recorded via the back facing camera in order to avoid "selfie" centred videos, hence maintaining the focus on the various aspects of the business instead, not the user. Also, shy users are more eager to record since they do not have to face the front facing camera at all.

Once a user starts recording by typically holding down a thumb on the video square, he/she is given 10 seconds to record. This encourages users to capture concise experiences only as well as keeps the resulting video file relatively small, enabling a fast upload later on (up to 2MB). Each recorded video has to be associated with a theme, either prior or after the recording (figure 4.3.17). This is fundamental to the application's existence as users would not be able to search a business's showreel by themes, a feature described in 4.3.4. Note that users can playback recorded videos, edit their themes and delete or rerecord them as illustrated by figure 4.3.18.

After the recording process, users are prompted to annotate their videos (figure 4.3.19). It firstly offers users to compose a short message, allowing others to get an even deeper impression of the video. Secondly, a user can add custom hashtags to increase the searchability of that video as part of its corresponding showreel. Lastly, users can provide pricing information depending on whether it makes sense for the specified theme of the video. For instance, price labels would only apply to themes such as "food" or "drinks", yet not "inside" or "outside" themes.

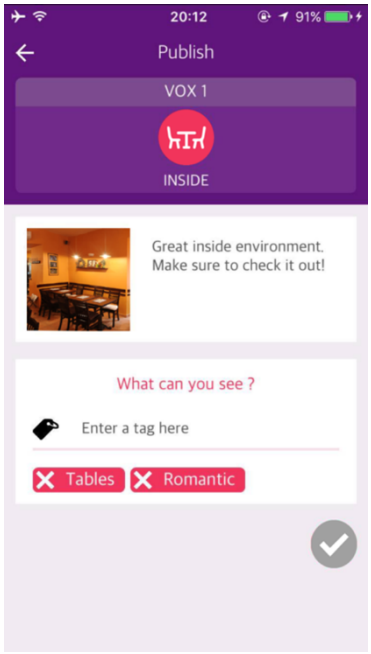

Figure 4.3.19 – Annotation before marking video as ready

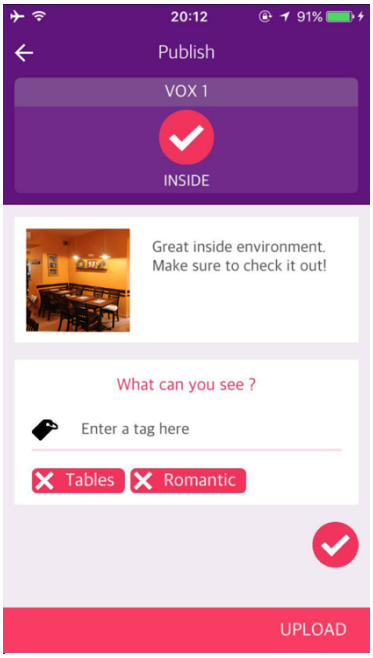

Figure 4.3.20 – Annotation after marking video as ready

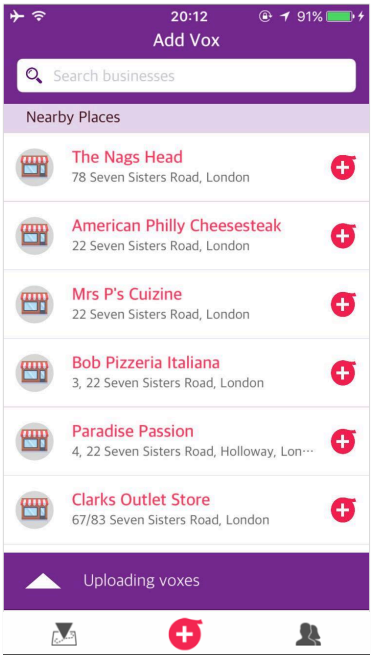

Figure 4.3.21 – Uploading process

To initiate an upload, users have to specifically mark videos as ready before an upload can be triggered (compare differences in figure 4.3.19 and 4.3.20). This way, accidental uploads caused by incorrectly using the UI can be prevented considerably. Once videos are submitted for uploading, the user is navigated back to the add video page, which will keep the user informed regarding the upload status (figure 4.3.21).

To sum up, the video capturing process is divided up into three stages: (1) recording, (2) annotating, and (3) uploading; each holding different challenges as will be elaborated on next.

4.3.3.1 Recording

Before implementing any recording utilities, I had to decide whether videos would be captured in full screen or squared mode. The first approach would have utilised the entire phone space, hence making it tricky to develop adequate UI components around it. More importantly, phones have distinct screen sizes thus leading to different video dimensions. Consequently, users' recorded videos might not properly fit on other users' phone screens, resulting in black bars on the top and bottom when playing them. To avoid any of these issues, the second approach seemed to be the more appropriate choice, even though it added the overhead of implementing video cropping functionality.

Many third-party applications, use Apple's build-in video recording utility, which due to the squared video format cannot be applied here. Therefore, a custom video recorder had to be developed, managing both the video recording, cropping as well as turning on and off the camera's flash light.

Videos were chosen to run with a traditional 30 frames per second alongside a 720p video quality setting. As a result, videos would run smoothly enough but at the same time not consume too much memory storage. After each recording session, the video recorder performs a cropping operation on all video frames, executing in a background thread in order to keep the main thread free for any user initiated UI updates. Any specific implementation details regarding this cropping functionality can be found inside the video recorder's "cropVideo()" method in Appendix 8.7.

A cropping invocation normally takes up to 2 seconds or less, before a user can replay the video. It is important to note, that during the cropping process a new video file is exported, preserving the same video quality, though the old

video file has to be deleted right after. All final videos are maintained as mp4 files inside the application's dedicated phone storage, not in the publicly accessible photo library as many iPhone users would be familiar with, enabling the video recorder to preserve full control over any of its recorded videos. Moreover, at this stage the application's internal database is not yet informed about any newly recorded video since users might decide to rerecord, thereby deleting the just created video file. Figure 4.3.22 demonstrates the threading mechanism applied during the recording process.

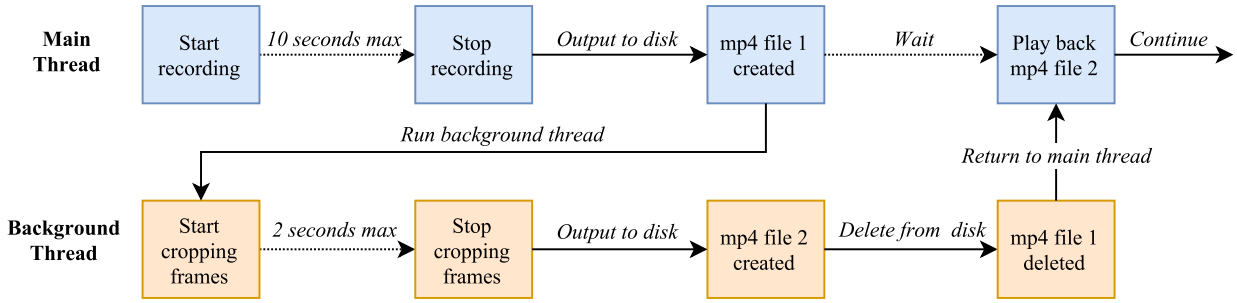

Figure 4.3.22 –High level overview of the recording process

4.3.3.2 Annotating

The annotation stage is supported via the "VoxMetadataVC" controller. Its main complexity lies in creating dynamic user interface components depending on the how many videos the user has recorded and what themes they belong to. Figure 4.3.23 for instance, demonstrates the effects on the annotation page in case a user has exhausted the maximum recording limit of four videos. The controller has to adequately layout the top menu items (purple box) as well as dynamically create four scroll views, each designated to the corresponding recorded video. In addition to that, the correct annotation boxes have to be rendered within the scroll views. To achieve this, each annotation box is realized in its own view class, thus enforcing reusability. Furthermore, users might not be satisfied with their recorded videos and thereby go back the video recording step to rerecord them. In order to prevent the loss of any video related annotations in between those two steps, both the "VoxMetadataVC" and "VoxRecorderVC" controllers have to communicate state to each other, applying the delegation design pattern as described in 4.1.1. The same principle applies, when the "VoxMetadataVC" permits an upload and delegates the video uploading responsibility to the "AddVoxVC" controller managing the original add video page (figure 4.3.21).

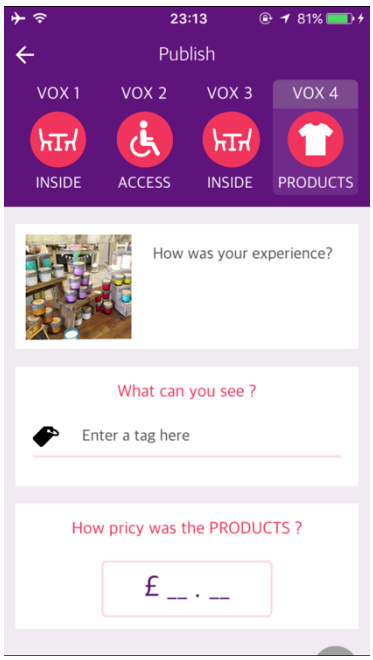

Figure 4.3.23 – Annotation page with four videos

4.3.3.3 Uploading

Once a user initiates a video upload, the control is passed onto the "AddVoxVC" controller. The controller then communicates the upload to its "VoxUploader" object property, which is responsible for handling the actual uploading mechanism (see "VoxUploader" class in Appendix 8.7). It is important to note that users can carry on using the application from that point onwards while the uploading activities occur in the background. Although, this adds to the overhead of notifying users about the uploading status, the application's usability is certainly improved as users can record further videos or perform other tasks without having to wait.

In essence, any incoming uploading requests are served in separate background threads running concurrently to accelerate multiple executing uploads. Most importantly, before initiating any uploading procedure, each recorded video with its annotation related data is stored inside an internal database using Apple's Core Data framework. This way, recorded videos are guaranteed to not be lost in case the user shuts down the application or accidental crashes occur, thus adding robustness to the system. Core Data uses a SQLite database as its persistent storage. However, it hides away any SQL implementation details, hence allowing developers to store, edit, and query objects similar to an object-relational database (ORD). Figure 4.3.24 demonstrates the data model representation serving as the underlying database schema. Any recorded video is stored as two separate Core Data objects, one holding video related details, the other capturing any annotation related data. Both objects keep references to each other in order to maintain their relationship. They are deliberately kept distinct as not every recorded video might be associated with some additional metadata, since users are not obligated to provide any further details during the video annotation process.

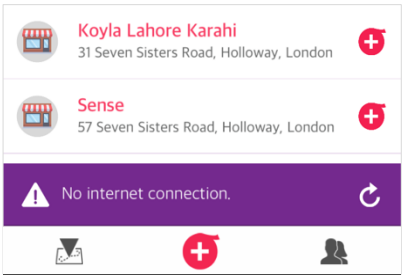

Figure 4.3.24 – Error message in case of a failed upload. Note that only part of the screenshot is shown here.

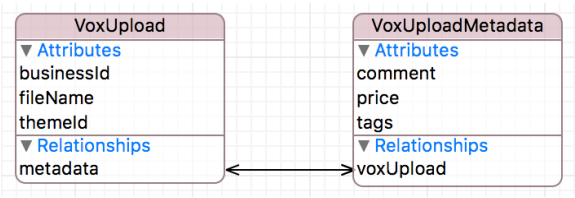

Figure 4.3.23 – Internal database schema

After successfully storing the videos of an upload request, the "VoxUploader" object will concurrently transfer each video file including its metadata using the RESTful API. With each successful upload, the corresponding video will be deleted from the disk alongside any stored information in the internal database. On the server side, each uploaded video is assigned a UID, which is stored together with the video's metadata inside the webserver database. Additionally, a thumbnail image is generated, which acts as a placeholder image when the iOS application has to buffer the corresponding video as part of the streaming process. Both the video and its thumbnail are backlogged on private disk space inside the "storage/voxes/video" and "storage/voxes/thumbnails" directories. At this point, it should be stated that each uploaded video is currently stored as part of a default user account (username is "seeder"), as user authentication was not within the scope of this project.

Since the uploading process is such a fundamental part of the application, I deliberately focused on increasing its robustness. More specifically, I wanted to address scenarios where uploads would fail due to missing network connections caused by network disruptions or phones simply losing their internet connections. To avoid any of these obstacles, the "VoxUploader" stops executing, maintains the data of any failed uploads and reports back occurred errors to the "AddVoxVC" controller. Again, this relies on the delegation design pattern since the responsibility of how to handle failed uploads is managed by the controller, not the uploader class. The same mechanism applies in other scenarios where the uploader object notifies the controller when it starts and finishes uploading, hence delegating the task of informing users about the uploading status to the controller.

Once the controller receives an upload error caused by a missing internet connection, it is immediately reported back to the user, allowing him/her to possibly act upon the situation (figure 4.3.24). During the downtime, the controller will listen for any "Reachability" events, which includes information about the network status. Those events are

triggered by a global “Reachability” object created during the launch of the application. It provides reachability notifications throughout the lifetime of the application, allowing any classes to register observers to receive network updates. This illustrates a great example of how the observer design pattern aided the “AddVoxVC” controller to obtain up to date network notifications. In the event of regaining the internet connection, the controller will notify its uploader object to presume with the remaining uploads. Refer to figure 4.3.25 to gain a conceptual overview of how the uploading process functions:

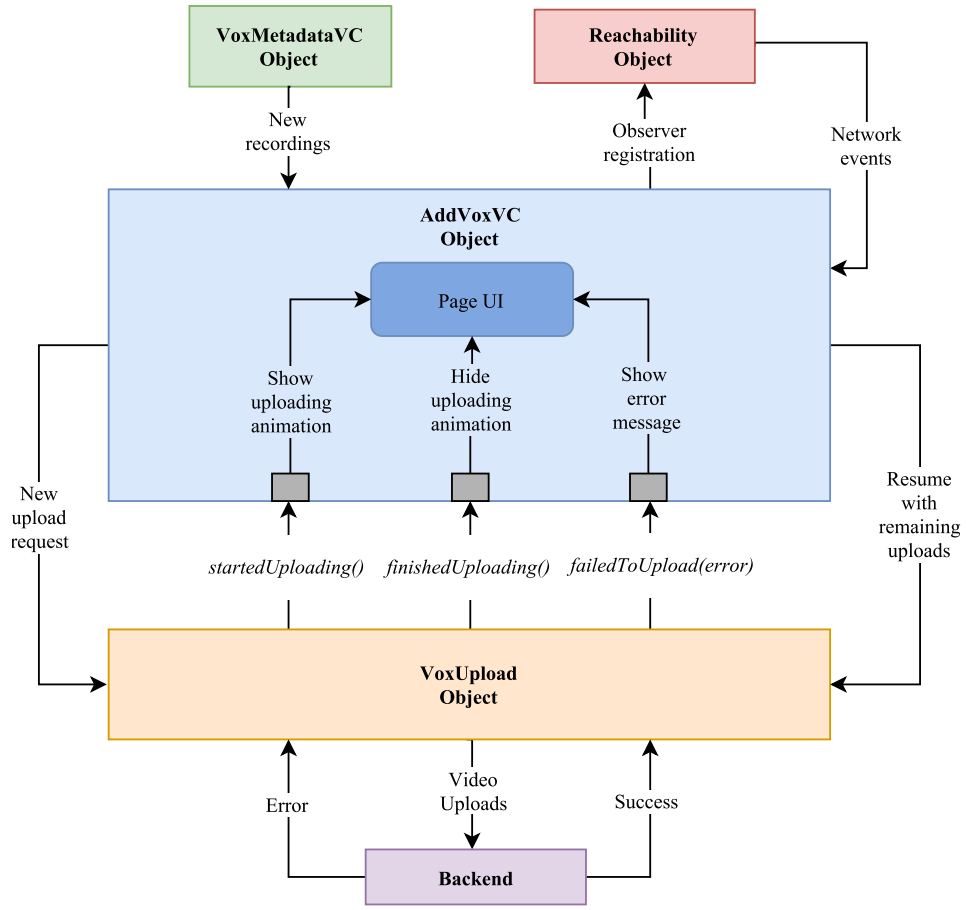

Figure 4.3.25 – High level overview of the uploading process

4.3.4 Video Showreel (Voxreel)

In 4.3.2 it was discussed how users can search for businesses and navigate to their pages. In this chapter, I specifically elaborate on the main functionalities they provide.

Each business page is managed by the “BusinessVC” controller which in return has two additional controllers embedded, one for coordinating the business info page (“BusinessInfoVC”) the other for handling the business showreel page (“BusinessVoxreelVC”). Note that only the latter will be discussed in this chapter.

Once users navigate to view to a showreel page, its most recently recorded video starts playing. In case there are not any, an appropriate message will be displayed explaining the situation. Users have the ability to swipe through the showreel and can hence actively browse through any available videos without having to watch those that do not seem interesting to them. As demonstrated by figure 4.3.26, any currently active video takes up the entire screen space and plays in a loop while listing its annotated information underneath. Users have the ability to like/unlike each video by tapping the heart button or even report it by selecting the warning button. The latter will present a video report page (figure 4.3.27), enabling the user to draft a complaint before sending it to the RESTful API where it will be stored and made accessible to the Voxreel team. With this approach, the vetting process for offensive and not applicable

content can thereby be accelerated and supported. More importantly, users can filter and order a showreel by selecting the white box button above the currently playing video. The application will present a settings page (figure 4.3.28) where users can select from a number of sorting and filtering options that they would like to apply to the showreel. It should clear that only those themes appear for which videos exist.

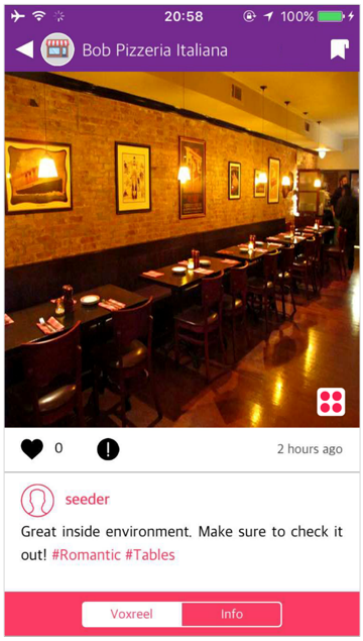

Figure 4.3.26 – Illustrates the showreel for a business named "Bob Pizzeria Italiana"

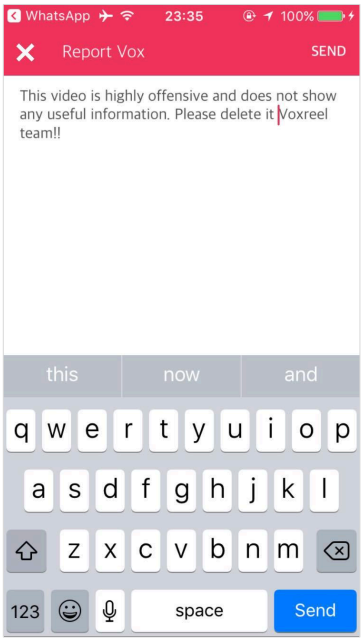

Figure 4.3.27 – Demonstrates the video report page

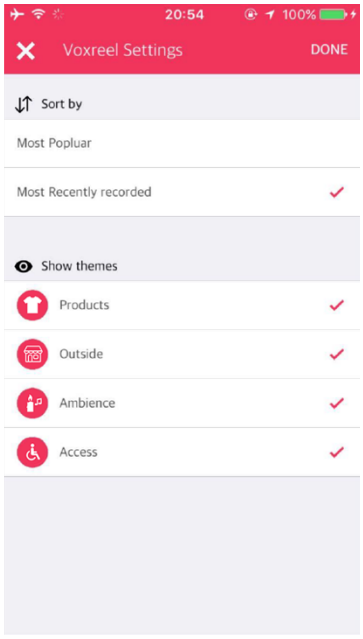

Figure 4.3.28 - Shows a business's showreel settings

4.3.4.1 Video Streaming

When loading a business page, its complete showreel with all its corresponding video details is downloaded from the server. Only the videos' metadata is fetched onto the iOS application, not the actual video files. This way, unnecessary file downloads are prevented, hence resulting in less memory allocation and short loading times. Refer to figure 4.3.29 to see the JSON format of a fetched video:


```
{
  "id": "qYgvZeEbo6boVjL0RyNk",
  "created": "2017-03-01 17:04:36",
  "voxTheme": 15,
  "thumb_url": "https://localhost/voxthumbs/qYgvZeEbo6boVjL0RyNk.jpg",
  "stream_url": "https://localhost/api/voxes/qYgvZeEbo6boVjL0RyNk/watch",
  "total_likes": 0,
  "has_liked": false,
  "has_reported": false,
  "voxTags": [],
  "author": {
    "userId": "qYgvZeEbo6boVjL0RyNk",
    "name": "seeder",
    "username": null,
    "photo": "https://localhost/profilepics/seeder.png"
  }
}
```


Figure 4.3.29 – Illustrates the JSON format of a downloaded video. Note a showreel is actually returned as a list of the videos.

After downloading a showreel, the "BusinessVoxreelVC" has all the information available to stream and display any available video in it. Typically during the initial video buffering, a thumbnail would be displayed to provide users with a first video insight (see "stream_url" in figure 4.3.29). Similarly to the video themes, thumbnails are stored in a publicly available server repository in order to avoid going through the full API request and response lifecycle. Consequently, a thumbnail can be displayed almost immediately assuming there is an internet connection. Moreover, downloaded thumbnails are cached using the "Kingfisher" image caching library as discussed in 2.4.1, thereby averting repetitive downloads of the same image.

Each streaming process is managed by a "VoxPlayer" object, capable of playing video resources referenced by a given URL. Thus, by passing the "stream_url" of the provided video JSON, a progressive download session with the RESTful API can be initiated. The progressive download mechanism is implemented on the backend side as part of the "VideoStream" class, achieved in four distinct steps:

1. *open():* Opens the video file
2. *setHeader():* Sets the proper HTTP headers to serve the video
3. *stream():* Streams the predefined amount of data progressively
4. *end():* Closes the video file

During the streaming process, the client can start playing a video once part of the file is downloaded, thereby generating an impression of streaming.

4.3.4.2 Reusability & Prefetching

The showreel page UI is implemented via two scrolling types. There is a main horizontal scroll enabling the left and right swiping between the various videos. Each of its video items is represented by a vertical scroll, allowing users to scroll up and down to view all available information for that video. The vertical scrolls were necessary as different iPhone types might not be able to fit the entire video information inside their screens.

Since showreels might contain many videos, it was crucial to create the horizontal scroll items dynamically, right before they are required by a user, thereby saving precious memory resources. Additionally, items were reused to save processing time instead of constantly allocating and deallocating them. Due to this reusability mechanism, the "VoxPlayer" instances associated with their corresponding horizontal items, could be reused as well. As a result, they are able to pre-fetch video files from the server, which results in less buffer times when videos eventually become active and start playing. Moreover, previously running video players hold on to their streamed video content in case users swipe back to re-watch their videos. Figure 4.3.30 demonstrates this applied prefetching approach in combination with the various scrolling types involved:

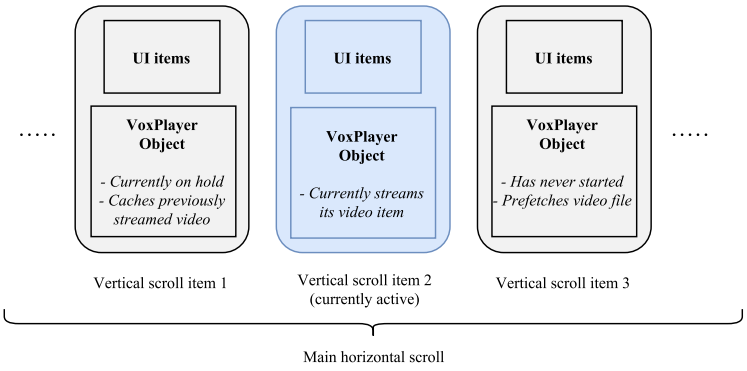

Figure 4.3.30 – High level overview of the scrolling mechanisms applied to a business showreel.

4.3.4.3 Local Filtering & Ordering

In order to prevent the "BusinessVoxreelVC" from performing too many responsibilities, an additional "Voxreel" class was implemented to support the controller in managing a showreel's current state and content (see "Voxreel" class in Appendix 8.7). More importantly, it performs the various filtering and ordering methods, allowing users to customize a business showreel based on their own preferences. Consequently, showreel manipulations can be performed locally and very quickly, escaping the overhead of querying the RESTful API.

4.3.5 Sourcing Businesses

Up to this point, it was assumed that users would be able to find any business either through the business search page or add video page. To come full circle with the system architecture, a business sourcing feature had to be developed, allowing users to contribute new businesses in case they do not exist on the webserver database. As already thoroughly discussed in 2.4.3, user generated businesses were the preferred approach over a "Places" API.

As illustrated in figure 4.3.31, users are able to create new venues via "add business" boxes that appear on both the business search or add video page at the bottom of their business listings.

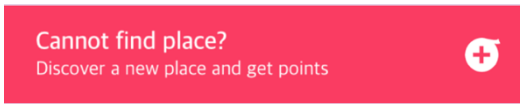

Figure 4.3.31 – Add business box

By tabbing one of these boxes, the application will present the "AddBusinessVC" controller (figure 4.3.32). Users are then prompted to specify the name, between one or three categories as well as the location of the new business. Categories are selected via the "AddBusinessCategoryVC" controller, granting users to navigate up and down the cached business category tree to find appropriate categories (figure 4.3.33). The business location is established through the "AddBusinessMapMarkerVC" controller where users are asked to move around the map and specify an exact location using a marker (figure 4.3.34). Users are forced to adequately zoom into the map, to guarantee an accurate location, otherwise the app will not set the marker. Once all required business details are specified, users can initiate an upload via the RESTful API, ultimately storing the new business in the webserver database. Each freshly stored business is searchable but marked as unapproved until a Voxreel team member manually verifies its correctness.

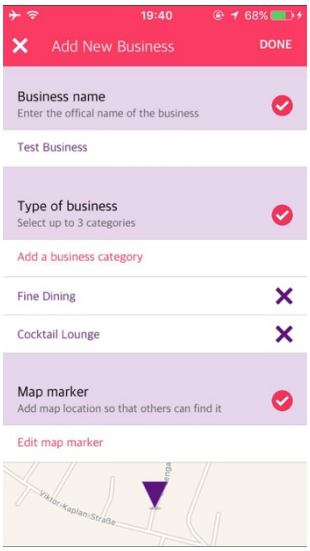

Figure 4.3.32 - New business

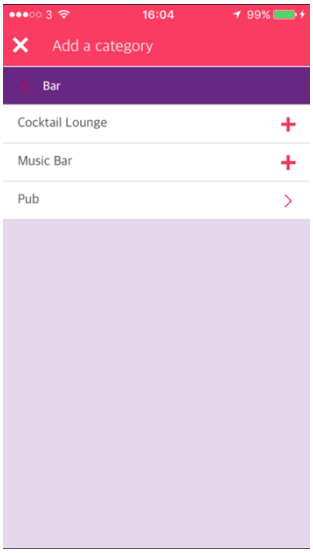

Figure 4.3.33 – Category

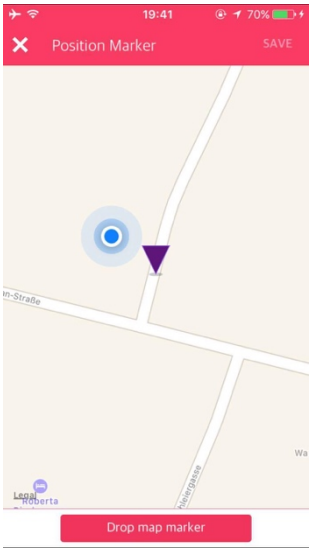

Figure 4.3.34 – Setting business location

4.3.6 Webserver Database

4.3.6.1 Structure

The database schema is implemented via a relational database using MariaDB as its database management system. It was designed using the normalisation technique, which observes the functional dependencies between fields within the tables in order to control redundancy and hence eliminate update anomalies. Consequently, the data integrity of the tables could be optimised. The resulting dependencies are as follows:

- All tables are in First Normal Form (1NF) since each row/column intersection only contains one single attribute value.
- All tables with a single-attribute primary key are automatically in Second Normal Form (2NF). The tables with a composite key ("business_categories", "vox_likes" table, "vox_theme_weights") also conform to 2NF as they do not include any partial dependencies.
- All tables are in Third Normal Form (3NF) since none of them contain transitive dependencies.

A complete list of the tables implemented is specified underneath. Note that each table attribute is denoted with its corresponding data type using the following abbreviations: I (INT), TI (TINYINT), SI (SMALLINT), V (VARCHAR), DE (DECIMAL), DO (DOUBLE), TS (TIMESTAMP), T (TIME), TE (TEXT).

"businesses" table

businessId (PK)	businesIdHashed	name	lng	lat	address	approved	created_at	created_at	createdBy
I	V	V	DE	DE	V	TI	TS	TS	I

"categories" table

categoryId (PK)	partent_id	ltf	rgt	depth	name	single_name	abv_name	created_at	updated_at
I	I	I	I	I	V	V	V	TS	TS

"business_categories" table

businessId (PK, FK)	categoryId (PK, FK)
I	I

"open_hours" table

businessId (PK, FK)	day	openTime	closeTime	created_at	updated_at
I	I	T	T	TS	TS

"voxes" table

voxId (PK)	voxIdHashed	userId (FK)	businessId (FK)	voxThemeId (FK)	description	price	created_at	updated_at
I	V	I	I	I	TE	DO	TS	TS

"vox_tags" table

voxId (PK, FK)	tag
I	V

"vox_likes" table

voxId (PK, FK)	userId (PK, FK)	created_at	updated_at
I	I	TS	TS

"vox_reports" table

voxReportId (PK)	voxId (FK)	userId (FK)	message	created_at	updated_at
I	I	I	TE	TS	TS

"vox_themes" table

voxThemeId (PK)	themeName	pricing	icon_filename	created_at	updated_at
I	V	TI	V	TS	TS

"vox_theme_weights" table

categoryId (PK, FK)	voxThemeId (PK, FK)	weight
I	I	SI

“users” table

userId (PK)	name	username	email	password	photo_filename	created_at	updated_at
I	V	V	V	V	V	TS	TS

4.3.6.2 Implementation

Each table was implemented using its custom migration class as demonstrated by figure 4.3.35. A migration typically has an *up()* method which specifies on how to create the database table, and a *down()* method instructing on how to delete the corresponding table when resetting the database. Migrations could then be run to either build or reset their tables via appropriate terminal instructions.

```
public function up()
{
    Schema::create('voxes', function (Blueprint $table) {
        $table->increments('voxId');
        $table->string('voxIdHashed')->default('');
        $table->integer('userId')->unsigned();
        $table->integer('businessId')->unsigned();
        $table->integer('voxThemeId')->unsigned();
        $table->text('description')->nullable();
        $table->double('price',8, 2)->nullable();
        $table->timestamps();


        $table->foreign('userId')->references('userId')->on('users')->onUpdate('cascade');
        $table->foreign('businessId')->references('businessId')->on('businesses')->onUpdate('cascade');
        $table->foreign('voxThemeId')->references('voxThemeId')->on('vox_themes')->onUpdate('cascade');
    });
}
```

Figure 4.3.35 – Illustrates the up() method of the “voxes” table migration

Most importantly, the table structures are specified via source code, allowing me and the remaining Voxreel development team to easily adapt and share the database schema. Additionally, tables could be created very quickly, thus supporting the development flow without having to use an external database building tool such as “MySQL Workbench”.

4.3.6.3 Obfuscation

In the “business” and “voxes” table there exists “businesIdHashed” and “voxIdHashed” attributes respectively. They represent the hashed IDs derived from their corresponding primary key IDs, used to obfuscate the actual business or vox ID whenever a business or vox would be returned as part of an API response. On the backend side, this added the implementation overhead of converting hashed IDs to their actual IDs or vice versa whenever API requests would arrive or be responded back to the client. Nonetheless, there are several arguments for choosing an obfuscation mechanism as already discussed in 2.4.2, most importantly to make it more challenging for adversaries or competitors to estimate table sizes or steal data via automated scraping strategies.

The reason why hashed IDs are stored in the database and not computed on the fly is to improve the API’s responsiveness. Through performance experiments, it was determined that the conversion from a hashed ID to an actual ID or vice versa takes on average 1ms. Consequently, when returning potentially thousands of businesses back to the client, such as for the wide area search during the client’s configuration stage, performance would be drastically impacted. For instance, responding back a list of 5000 businesses would take an extra 5 seconds to compute. To avoid these issues, hashed IDs are stored in the database even though they add redundancy. However, since they are all depending on primary key IDs, they do not affect the normalisation form of their tables.

5 Testing

In this chapter, I will elaborate on the various testing strategies applied to the iOS application and the RESTful API, followed by an overall test summary.

5.1 iOS Application

5.1.1 Unit Testing

Unit testing has been applied to the main driving helper and service classes of the application to ensure they maintain their intended behaviour while other components of the system adapt or new parts are being added (figure 5.1.1). I specifically focused on testing those classes that have a significant impact on the application such as the "Voxreel", "VoxPlayer", or "BusinessSearchTracker". These classes are least likely to change, making them easy to overlook and break with changes to other areas. Additionally, I unit tested the various components of global service classes, including the "FileManager" or "GooglePlacesAPI", thus guaranteeing that their provided functionalities can be safely used throughout the application. For each of the corresponding classes, there exists separate test case classes. Therefore, unit tests could be run independently from each other. More importantly, an extensive testing case was developed for the "APIManager" class, which is responsible for coordinating entire network communication with the RESTful API. Consequently, errors introduced by changes in the API were detected swiftly.

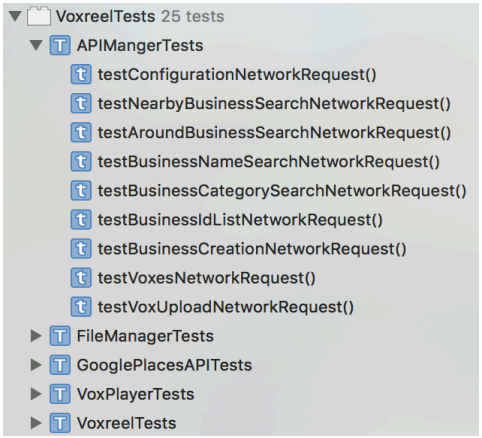

Figure 5.1.1 – Shows the unit tests as well as the underlying tests for the "APIManagerTests" case

It is important to note that additional measures had to be taken in order to test the asynchronous methods of the "APIManager" and "GooglePlacesAPI" since their corresponding tests have to wait for the asynchronous operations to complete. Waiting times were manually specified for each unit test taking into account testing with the local and remote server (dedicated server rented from UK Dedicated Servers Limited[23]). This not only allowed testing for their expected behaviour but also to ensure performance thresholds are met, such as network requests responding within a certain amount of time. Since asynchronous tests usually execute slower, they should be kept separate from the faster unit tests. Again, this influenced my decision of developing individual test cases for each tested class.

5.1.2 User Interface Testing

The UI is a crucial part of the iOS application, which heavily focuses on generating a great user experience. All application pages involved a substantial amount of custom animations and self-designed views. Subsequently, in order to guarantee that the UI behaves as intended, it was necessary to consider the various user interactions with the application by applying an UI testing methodology.

Since Xcode offers a very powerful built-in UI testing framework, I chose it as my preferred UI testing infrastructure. Each test case was attributed to a specific application feature as their corresponding tests are similar in their use cases. Figure 5.1.2 illustrates this by listing the five main use case groups alongside the underlying tests for

[23] http://www.ukservers.com

"BusinessCreationUITests". Each UI test was written as an acceptance test, specifying user interactions through the testing API as illustrated by figure 5.1.3. The API would then be responsible for synthesising the appropriate queries and events to the application's view objects, thus allowing me to primarily focus on the specific use cases. Additionally, the UI objects' properties and state would be compared against the expected state, again by using the

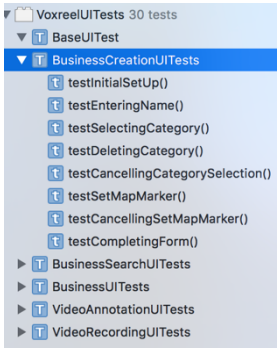

Figure 5.1.2 – Overview of UI test cases

```
func testEnteringName()
{
    // When I arrive at the bussiness creation page
    let tablesQuery = self.app.tables
    let textField = tablesQuery.textFields[ self.addNamePlaceholder ]

    // AND no business name has been specified yet
    XCTAssertEqual( textField.value as! String, "" )

    // WHEN I enter a new business name
    let userInput = "Costa Coffee"
    textField.tap()
    textField.typeText( userInput )

    // AND confirm it
    self.app.buttons[ "Done" ].tap()

    // THEN the name textfield is filled out
    XCTAssertEqual( tablesQuery.textFields[ userInput ].value as! String, userInput )
}
```

Figure 5.1.3 – "testEnteringName()" UI test (part of "BusinessCreationUITests" case) showing its acceptance testing nature

UI testing framework.

Even though, UI tests take a considerable amount of time to run, they proved to be of high importance to ensure use cases remain stable when new UI updates were implemented.

5.1.3 Device Testing

At the end of the development, the iOS application has been tested on the following physical iPhone devices running iOS 10.2 in order to guarantee they conform to the specified UI interactions:

- iPhone 5 (4 inch display)
- iPhone 6s (4.7 inch display)
- iPhone 6 Plus (5.5 inch display)

The device testing approach was very fundamental as each of the chosen devices offered different screen sizes. Therefore, the UI could be tested very effectively as visual abnormalities could be demonstrated clearly. Additionally, UI tests were executed on each of them to detect any defect behaviours in the application's main uses cases. However, the testing results indicated that there were no issues with the user interface experience on any of the devices.

5.2 RESTful API

5.2.1 Manual Testing Via Postman

During the API development, it was often necessary to quickly and easily test the new endpoint functionalities without having to write integration tests straight away. This manual testing was achieved through the popular Postman [24] application. Firstly, it allowed the creation of requests against the various API endpoints to rapidly verify if they adequately report errors such as missing or incorrect parameters alongside their corresponding HTTP codes. Secondly, API responses could be checked whether they output the expected results in the appropriate format and how long it would take to process each request. Ultimately, this lead to the discovery of the performance impact of dynamically computing businesses' and voxes' hashed IDs as mentioned in 4.3.6.3.

[24] https://www.getpostman.com/

By choosing this testing strategy, the API's correctness could be guaranteed to a great extent without having to validate the API's functionalities via HTTP calls from the iOS application's "APIManager" class. Consequently, both the iOS application and the API could be developed separately from each other, which resulted in faster development overall.

5.2.2 Integration Testing

After manually testing an API endpoint, several integration tests were written in order to guarantee its correctness throughout the remaining development. Each endpoint would be tested using the following testing criteria (see figure 5.2.1 for an example):

1. A test to check that the endpoint is even working
2. A test to check the endpoint's returned results are correct and properly formatted using various mock inputs
3. A test to check the endpoint is responding correctly for missing or invalid request parameters (only applies to endpoints where request parameters have to be provided)

Tests would traditionally contain PHPUnit assertions for verifying expected conditions. For endpoints manipulating the database state, additional database testing would be applied, checking that data exists or does not exist in the database matching a given set of criteria. Furthermore, the database would be reset for each test in order to prevent data from a previous test from interfering with tests that follow afterwards. During the testing stage, it was often required to fill the database with predefined test data sets. This was achieved by implementing factory classes that generate mock data models that could then be inserted into the database. Instead of manually specifying the attributes of the mock models, they were filled with randomly generated inputs using Faker[25], a PHP library. In addition to that, a random video file would be used as a stub for the integration tests involving the video upload.

```
/**
 * Test nearby search is working
 *
 * @return void
 */
public function testNearbySearch(){...}

/**
 * Test nearby search results
 *
 * @return void
 */
public function testNearbySearchResults(){...}

/**
 * Test if nearby search is responding correctly when parameters are missing
 *
 * @return void
 */
public function testNearbySearchMissingParams(){...}
```

Figure 5.2.1 – Tests for nearby search endpoint as part of the "BusinessSearchTest" case. Note their implementation is hidden behind the "{...}" notation.

5.2.3 Database Seeders

As already discussed in 4.3, many components of the iOS application rely on data provided via the RESTful API. To support the manual testing of those features, the webserver database had to be seeded with data. Therefore, seeder classes were developed on the backend side that could then be executed when performing database migration runs. One very crucial class was the "BusinessSeeder" class, responsible for filling the database with over 5000 thousand real businesses around the UK (mostly London), scraped via the Google Places API.

It is important to note that some seeders were used to generate non-testing database records as well. This includes those liable for filling the database with the business category tree, video themes and video theme weights.

5.3 Test Summary

A wide set of testing methodologies have been applied during the development of the system. For the iOS application 30 UI tests were written covering the main use cases for each page including 25 unit tests for verifying the main

[25] https://github.com/fzaninotto/Faker

driving classes. Both automated testing methods were used throughout the application development to detect functional and UI related errors that may have otherwise not been found through manual testing. Nonetheless, the manual device testing still offered reinsurance that the application behaves the same for variously sized iPhone devices.

During the development of the RESTful API, 20 integrations tests were written, which have been consistently executed to verify its behaviour. Since all its endpoints have been tested for multiple request use cases, it can be expected that the API functions accordingly to its specification and returns correct results. The additional manual testing approach via Postman was crucial for rapidly testing new or changed endpoints before automated tests were written for them or existing ones adapted.

To sum up, the resulting automated tests can be re-run for future development to ensure the system continues to function correctly throughout its lifetime.

6 Conclusions & Evaluation

6.1 Summary Of Achievements

In the 1.3, several primary goals were set before the start of the project. The initial goals focused on conducting a background research on the effectiveness of videos including a critical reflection on today's reviewing culture as well as producing a competitor analysis on current reviewing applications. Each of those goals were thoroughly discussed and hence met in 2.2 and 2.3 respectively. More importantly, the remaining goals addressed the implementation of an iOS application and RESTful API that would support the following key features:

a. Users can record and share short videos, annotating them with useful metadata
b. Users can search for businesses using various searching algorithms
c. Users can stream and browse through venues' user generated video reels
d. Users can search video reels for specific content

All of the above features have been fully implemented and backed up by a functioning iOS application and RESTful API. Additionally, each feature has been elaborated on in great detail in various subchapters as part of 4.3. Given the above evidence, I can conclude that each of the desired goals were indeed achieved.

6.2 Evaluation Of Project

6.2.1 Stated Challenges & Delivered Solutions

As detailed in 1.1, there exists several problems in our current reviewing culture, which the Voxreel system was developed to solve.

The first identified problem was caused by the subjective and negative nature of written reviews including their star ratings, resulting in ultimately destroying a business's reputation without any chance for recovery. My application solves that issue by abandoning any star ratings and written reviews completely. Instead, it focuses on allowing users to generate and share short videos that they can further annotate with useful information, thus encouraging people to rather focus on positive experiences than negative ones. Furthermore, a reporting feature was implemented enabling users to report offensive and inappropriate video content, thereby supporting the Voxreel team in detecting unpleasant video materials. Even though inappropriate experiences can be expressed and shared, they can be ultimately reported and do not add to the overall image of the business as there are no ratings in place.

The second challenge results from fake and made up written reviews, thereby posing a considerable cause for concern regarding the exception gap between consumer reviews and actuality. My application aims to diminish this problem by heavily focusing on the medium of video rather than text, which makes it a lot more difficult to fake content. Also since there are no ratings in place and users can report offensive content, there is not much to gain from fake videos improving or hurting a business's reputation. Consequently, by allowing users to stream videos rather than reading reviews, the expectation gap could be further closed.

The last problem emerges from text providing little actual evidence and insights as it depends on the subjective experience of the users as well as how they express themselves. My application avoids those issues since users streaming a business's showreel can decide for themselves on whether or not what a venue offers appeals to them. Additionally, users have the ability to filter the showreel for video themes they are interested in the most, hence avoiding watching video content they do not wish to view.

Ultimately, the fact that current competitor applications as in Yelp and Foursquare mostly abandon the video medium and heavily focus on written reviews as illustrated in 2.3, while the Voxreel system supports the exact opposite, solidifies that my application is fit for purpose.

6.2.2 Implementation & Design

The iOS application was designed on top of the MVC design pattern, resulting in a clear separation between controller, model and view classes. Thus, changes to the UI, data flow or model logic can be easily achieved in the future and usually requires adapting one class only. Also, through the delegation design pattern classes could communicate with each other without causing strong coupling relationships. Again, this resulted in loosely coupled application components, which can hugely benefit future development. Furthermore, by following a fail-safe programming style and extracting out global functionalities to service classes, a great overall software quality could be achieved.

From an implementation point of view, the configuration process (see 4.3.1) provides an elegant way of dynamically configuring the application using the RESTful API, avoiding the manual storage of configuration data. However, that data is currently cached instead of persistently stored as the latter would have added unnecessary complexity to the project. Ultimately, this mechanism has to be adapted to Core Data, in order to store configuration data in a database, thereby avoiding redundant configuration downloads during the application launch. For the business search (see 4.3.2), several optimisation techniques were implemented to avert overloading the server and increase searching speeds. As stated in 4.3.2.1, one missing optimisation is to firstly check whether a specified location is within the bounds of the cached wide area businesses prior to performing any API searches. The video recording feature (see 4.3.3) also follows an adequate implementation practice, conducting video cropping operations and video uploads in background threads, thus maintaining a responsive UI. For the latter video data is stored inside an in-app database using Core Data before initiating an upload. Therefore, the application overcomes the risk of losing crucial video recordings in case there is no internet connection or the application shuts down. Lastly, the business voxreel feature (see 4.3.4) was implemented on top of reusable view and video player components to minimize memory allocation, addressing the dangers of having too many showreel videos. Additionally, video streams are prefetched to reduce loading times when they become active. More importantly, the implemented “VoxPlayer” class, responsible for coordinating video streams, is immune to any changes in the streaming technique applied by the backend, thereby offering a robust solution.

From a technology point of view, it was the right decision to pick Swift over Objective-C, simply due to the fact that Swift is more cutting-edge, provides a better user-friendly language syntax and will ultimately eliminate Objective-C in the near future [27]. Moreover, the chosen libraries, such as the network or image caching libraries, provided the required functionalities without causing any implementation issues and were hence also fit for purpose.

Similarly to the iOS application, the RESTful API was developed using the MVC design pattern as well, hence offering the same implementation benefits. More importantly, the repository design pattern resulted in the strict separation of data access from business logic, thereby avoiding the mixture of SQL and data flow code. Furthermore, request validator classes were used to validate incoming API requests and middleware layer classes aided in preparing requests before entering a controller, whereas API responses were formatted via transformer classes. All of these strategies allowed the implementation of thin controller classes and overall weakly coupled backend components.

Implementation wise, all business search queries operate very quickly as tested with a seeded database of over 5000 businesses. Any other endpoints, such as retrieving configuration data, creating a new business, querying a business's showreel videos or liking/unlinking a video, all perform swiftly and operate upon common sense request parameters for client usability. However, one lacking point is the currently implemented video streaming functionality as it relies on progressive downloading. As already elaborated on in 2.4.5, a progressive download methodology can be fairly slow if there is a poor internet connection since it does not consider the currently availabe bandwidth. The ultimate solution is to apply an adaptive streaming technique that can dynamically adjust the stream's bitrate. One technology solution would be Apple's HTTP Live Streaming [47].

The chosen backend technologies including the thumbnail or nested set model libraries each serviced the API implementation accordingly. However, there is still no guarantee that Laravel, the chosen backend framework, will be maintainable enough for the future. Although, it is the most popular PHP framework on GitHub, it is nonetheless based on a fairly old language. Possibly, Django or Node.js could have offered a more future proof and scalable solution since they are based on more modern languages: Python and JavaScript respectively.

6.2.3 User Interface

A considerable amount of time was spent on developing a dynamic and user-friendly interface. To evaluate the iOS application's resulting UI, Don Norman's six design principles can be applied [54]:

1. Visibility: All the information displayed is clearly formatted to the user in a meaningful way using distinct colours, text sizes/styles and icons. Global functions such as the business search or nearby business listings can be accessed anytime as part of the application's bottom navigation bar.

2. Feedback: The application provides numerous feedback features to inform the users about the effects of their actions. For instance, buttons and switches are animated accordingly, always illustrating to the user whether or not they are pressed, active or inactive. Moreover, loading animations appear for longer lasting computations as in the video cropping, the video uploading, the business searching or the initial configuration when launching the application. Also, before users commit to any delete operations (e.g. delete or redo video), the application asks for an additional user consent.

3. Affordance: UI views were designed in such a way that their attributes provide clues to their functions and how they should be used. During the business search or business creation process for instance, clues such as the white cross button in the left corner indicates that user can always dismiss the search or business creation. Another example would be that the placeholder texts in any search bar or text field indicates what users are expected to enter.

4. Mapping: All application pages offer clear mappings between their controls and their effects. For instance, the upload button to finally upload someone's recorded and annotated videos is clearly marked with an upload label in capital letters, thus guaranteeing that users will understand the effects of pressing that button. The same design mechanism applies to any other button initiating some crucial process. Additionally, appropriate button icons are used to avoid further ambiguities about their expected functionalities.

5. Consistency: The application offers similar operations and uses similar elements for achieving similar tasks. For instance, any dismiss operation is initiated with a cross button or any add operation with a plus button. Similarly, any business search type (search by name or by category) behaves the same from the user's point of view regardless

if performed locally or via the RESTful API. Also, the application's colour theme (red, purple and white) and icons remain consisted throughout all pages. Subsequently, this makes it easy for users to learn and use the system.

6. Constraints: Constraints are applied to prevent users to prematurely trigger main actions. For example, during the business creation process users have to provide a name, at least one business category and the exact location of the business before they can press the submit button to upload the new business. The same mechanism applies to the video recording process. Users can only move onto the annotation page if they have recorded at least one video.

6.3 Future Work

A number of upcoming developments are listed below:

- The "AppCache" class has to be refactored using Core Data to persistently store configuration data.
- The "VideoStream" class on the backend side has to be reimplemented to apply an adaptive streaming technique. This will also require enhancing the video upload procedure to transcode each file into various chunks of bitstream sizes which can then be dynamically served to the iOS application depending on the current bandwidth requirements. Most likely, a video codec software will have to be applied here to achieve the best results.
- The business search has to be further optimised by firstly checking if specified locations are within the bounds of the locally stored businesses. Consequently, the server load can be reduced even more.
- The business info page has to be fully displayed as currently only the business voxreel page is implemented. This should be fairly straight forward as the business details are all available as part of the controller, although they are currently omitted from the UI. Furthermore, the underlying functionality for the bookmark button on a business page has to be provided as well.
- The configuration process has to automatically rerun when users change their locations notably, so that nearby and wide area businesses are always up to date. The "LocationTracker" class has already been fully implemented to serve this purpose.
- A profile page has to be created as illustrated by 6.3.1, listing a user's recorded videos and businesses. The associated controller already exists as part of the submitted iOS code base. This will require an authentication mechanism implemented using the Facebook SDK[26], thus allowing users to login in with their Facebook credentials. Note that this is very fundamental as currently each uploaded business and video is associated with a dummy user.

Figure 6.3.1 – Missing profile feature

6.4 Conclusions

Firstly, I was able acquire new programming skills in Swift and gain experience with iOS development, something that was completely new to me prior to this project. Secondly, my knowledge in developing RESTful APIs using PHP could be greatly improved as well, thereby aiding me when learning other frameworks such as Django or Node.js. I anticipate using both my newly learned frontend and backend skills for my future career as a software engineer.

Since I did not previously work on a project of this scale on my own, this experience also taught me valuable lessons in software engineering in regards to code documentation and code management in general. More importantly, I

[26] https://developers.facebook.com/docs/ios/

learned the essence of keeping new applications to a bare minimum, thus striving for a lean product, instead of over engineering solutions that hinder the development process and do not add value to the application. On a side note, I was able to improve my understanding of the current reviewing culture by investigating the negative impacts of written reviews and ratings and how the use of videos can overcome such issues.

To sum up, I am very pleased with the outcome of this project and I anticipate launching the application onto the App Store once the remaining list of future developments have been fully integrated.

7 Bibliography

[1] Khan, H (2016), "How Online Reviews Impact Local SEO and Why They Matter to Your Bottom Line", https://www.shopify.com/retail/119916611-how-online-reviews-impact-local-seo-and-why-they-matter-to-your-bottom-line

[2] Rudolph, S (2015), "The Impact of Online Reviews on Customers' Buying decisions[Infographic]", http://www.business2community.com/infographics/impact-online-reviews-customers buying-decisions-infographic-01280945#PWDildZdkO0tK2rw.97

[3] Charlton, G (2015), "Ecommerce consumer reviews: why you need them and how to use them", https://econsultancy.com/blog/9366-ecommerce-consumer-reviews-why-you-need-them-and-how-to-use-them/

[4] BrightLocal (2016), "Local Consumer Review survey", https://www.brightlocal.com/learn/local-consumer-review-survey/

[5] Shrestha, K (2016), "[infographic] 50 Stats You Need to Know About Online Reviews", https://www.vendasta.com/blog/50-stats-you-need-to-know-about-online-reviews

[6] Muchnik, L., Aral, S. and Taylor, S.J., 2013. Social influence bias: A randomized experiment. Science, 341(6146), pp.647-651.

[7] Thibaut (2017), "Video sharing vs text content sharing: 9 reasons for an exponential contrast", http://video-university.87seconds.com/video-sharing-vs-text-content-sharing-9-reasons-for-an-exponential-contrast/

[8] West, D., Grant, T., Gerush, M. and D'silva, D., 2010. Agile development: Mainstream adoption has changed agility. Forrester Research, 2(1), p.41.

[9] Aral, S. (2013), "The Problem With Online Ratings", http://sloanreview.mit.edu/article/the-problem-with-online-ratings-2/

[10] Bonelli, S (2016), "70% will leave a review for a business when asked", http://searchengineland.com/70-consumers-will-leave-review-business-asked-262802

[11] CMA (2015), "CMA acts to maintain trust in online reviews and endorsement", https://www.gov.uk/government/news/cma-acts-to-maintain-trust-in-online-reviews-and-endorsements

[12] Wikipedia (2017), "Marshall McLuhan", https://en.wikipedia.org/wiki/Marshall_McLuhan

[13] Marketwired (2017), "A Minute of Video Is Worth 1.8 Million Words, According to Forrester Research", http://www.marketwired.com/press-release/a-minute-of-video-is-worth-18-million-words-according-to-forrester-research-1900666.htm

[14] Margalit, L (2015), "Video vs Text: The Brain Perspective", https://www.psychologytoday.com/blog/behind-online-behavior/201505/video-vs-text-the-brain-perspective

[15] Frasco, S (2017), "Why Instagram, Snapchat & Vine Should Be Included In Your Social Media Marketing Strategy", https://www.convertwithcontent.com/instagram-snapchat-vine-included-social-media-marketing-strategy/

[16] Riecke-Gonzales, A (2015), "Marketing to Millennials: The Attention Deficit Generation", http://www.optimizemybrand.com/2015/11/08/marketing-to-millennials-attention-deficit/

[17] Yelp Sucks (2017), http://yelp-sucks.com/

[18] Witczak, B. (2017), "Carthage vs. CocoaPods vs. Git submodules", https://medium.com/real-life-programming/carthage-vs-cocoapods-vs-git-submodules-9dc341ec6710

[19] Wikipedia (2017), "Yelp", https://en.wikipedia.org/wiki/Yelp

[20] Apple Developer Guide (2016), "Using NSURLSession", https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/URLLoadingSystem/Articles/UsingNSURLSession.html

[21] Clark, P (2013), "Yelp's Newest Weapon Against Fake Reviews: Lawsuits", https://www.bloomberg.com/news/articles/2013-09-09/yelps-newest-weapon-against-fake-reviews-lawsuits

[22] Wikipedia (2017), "Foursquare", https://en.wikipedia.org/wiki/Foursquare

[23] Wikipedia (2016), "Swift (programming language)", https://en.wikipedia.org/wiki/Swift_(programming_language)

[24] Wikipedia (1984), "Objective-C", https://en.wikipedia.org/wiki/Objective-C

[25] Kuz, M. (2016), "7 Advantages of Using Swift Over Objective-C", https://mlsdev.com/blog/51-7-advantages-of-using-swift-over-objective-c

[26] Solt, P (2015), "Swift vs Objective-C: 10 reasons the future favors Swift", http://www.infoworld.com/article/2920333/mobile-development/swift-vs-objective-c-10-reasons-the-future-favors-swift.html

[27] Cunningham, A. (2015), "Craig Federighi talks open source Swift and what's coming in version 3.0", https://arstechnica.com/apple/2015/12/craig-federighi-talks-open-source-swift-and-whats-coming-in-version-3-0/

[28] Fractal (2017), "Introduction", http://fractal.thephpleague.com/

[29] Gorelik, A., Chawla, S., Syed, A., Burda, L., Yee, M. and Grantimahapatruni, S., Chawla Sachinder S., Syed Awez I. and Yee Mon F., 2001. *Nested relational data model*. U.S. Patent Application 09/782,186.

[30] Wikipedia (2017), "Nested set model", https://en.wikipedia.org/wiki/Nested_set_model

[31] StubbornJava (2017), "Obfuscating and Shortening Sequential ids with HashIds", https://www.stubbornjava.com/posts/obfuscating-and-shortening-sequential-ids-with-hashids

[32] Sturgeon, P (2015), "Auto-Incrementing IDs: Giving your Data Away", https://philsturgeon.uk/http/2015/09/03/auto-incrementing-to-destruction/

[33] Google Places API (2017), "Places API for iOS", https://developers.google.com/places/ios-api/

[34] Google Maps APIs (2017), "10.5 Intellectual Property Restrictions", https://developers.google.com/maps/terms#section_10_5

[35] Google Places API (2017), "Documentation – Place IDs", https://developers.google.com/places/place-id#save-id

[36] Foursqaure (2014), "Venues Service", https://developer.foursquare.com/overview/venues#rules

[37] Yelp Fusion (2016), "Terms of Use", https://www.yelp.at/developers/api_terms

[38] Google Maps APIs (2017), "Pricing details", https://developers.google.com/maps/pricing-and-plans/#details

[39] Foursquare For Developers (2017), "Rate limits", https://developer.foursquare.com/overview/ratelimits

[40] Yelp Fusion (2017), "FAQ – General questions", https://www.yelp.at/developers/faq

[41] Ward, D., Hahn, J. and Feist, K., 2012. Autocomplete as a research tool: a study on providing search suggestions. *Information Technology and Libraries (Online)*, *31*(4), p.6.

[42] Cfycat (2017), "Gfycat Terms of Service – Intellectual Property Rights", https://gfycat.com/terms

[43] YouTube (2017), "YouTube API Services Terms of Service", https://developers.google.com/youtube/terms/api-services-terms-of-service

[44] YouTube (2017), "Implementing OAuth 2.0 Authorization", https://developers.google.com/youtube/v3/guides/authentication

[45] Gfycat (2017), "Authentication", https://developers.gfycat.com/api/#authentication

[46] Wikipedia (2017), "List of video hosting services", https://en.wikipedia.org/wiki/List_of_video_hosting_services

[47] Narang, N. (2015), "#6 Concept Series: What is the difference between Progressive Download, RTMP Streaming and Adaptive Streaming", http://www.mediaentertainmentinfo.com/2015/04/6-concept-series-what-is-the-difference-between-progressive-download-rtmp-streaming-and-adaptive-streaming.html/

[48] Ozer, J. (2011), "Streaming Vs. Progressive Download Vs. Adaptive Streaming", http://www.onlinevideo.net/2011/05/streaming-vs-progressive-download-vs-adaptive-streaming/

[49] Waters, K., 2009. Prioritization using moscow. *Agile Planning*, *12*.

[50] Wikipedia (2017), "Entity-relationship model", https://en.wikipedia.org/wiki/Entity%E2%80%93relationship_model

[51] Wikipedia (2017), "Decorator Pattern", https://en.wikipedia.org/w/index.php?title=Decorator_pattern&action=history

[52] Apple Developer (2017), "The Swift Programming Language (3.1)", https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/

[53] Apple Developer (2017), “iOS Human Interface Guidelines”, https://developer.apple.com/ios/human-interface-guidelines/overview/design-principles/

[54] Norman, D.A., 1983, December. Design principles for human-computer interfaces. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems* (pp. 1-10). ACM.

8 Appendices

8.1 Requirements Document

8.1.1 iOS Application

Functional Requirements

ID	Requirement	Priority
Category: Business creation		
REQ1.00	The iOS application shall have reusable page for creating businesses.	M
REQ1.01	The iOS application shall require a name, between 1 and 3 business categories and a geographic location when creating a new business.	M
REQ1.02	The iOS application shall allow the user to provide open hours and address details when creating a new business.	C
REQ1.03	The iOS application shall provide the user with a business category tree to select from.	S
REQ1.04	The iOS application shall allow the user to select the business location by dropping a map marker on a map.	M
REQ1.05	The iOS application shall force the user to specify a very detailed business location.	S
REQ1.06	The iOS application shall allow users to find created businesses.	M
Category: Nearby businesses		
REQ1.00	The iOS application shall always provide the user with nearby businesses around him/her.	M
REQ.1.01	The iOS application shall periodically download and cache nearby businesses around the user.	M
REQ.1.02	The iOS application shall allow the user to search the list of nearby businesses.	C
Category: Business search > specifying search criteria		
REQ3.00	The iOS application requires a location for each search.	M
REQ3.01	The iOS application shall always reuse the location from the last search.	S
REQ3.02	The iOS application shall allow the user to specify and alter the search location.	M
REQ3.03	The iOS application shall allow the user to select his/her current location or the current map view as a location for a search.	M
REQ3.04	The iOS application shall allow the user to type out a custom location description and select an autocomplete suggestion as a location for a search (provided by the Google Places API's place	M
REQ3.05	The iOS application shall allow the user to search by a custom keyword.	M
REQ3.06	The iOS application shall allow the user to search by a predefined business category.	M
REQ3.07	The iOS application shall provide an autocompletion list of business categories and business names when the user enters a custom keyword.	S
REQ3.08	The iOS application shall allow the user to select a business from the autocompletion list and navigate to the business's page (without triggering a search).	S
REQ3.09	The iOS application shall allow the user to select a business category from the autocompletion list and trigger a search by category immediately.	S
REQ3.10	The iOS application shall perform searches locally via the cached businesses in case the user triggers a search based on his/her current location.	C

Category: Business search > displaying results		
REQ4.00	The iOS application shall always display a search bar with the lastly triggered search criteria.	M
REQ4.01	The iOS application shall display each search result with the business's name, categories and open hours.	M
REQ4.02	The iOS application shall always display the results from the last search.	M
REQ4.03	The iOS application shall allow the user to display the search results either on a list view or map view.	S
REQ4.04	The iOS application shall allow the user to navigate to a business's page when selecting it from the search results.	M
REQ4.05	The iOS application shall always order the search results by nearest location.	S
REQ4.06	The iOS application shall allow the user to create a new business if he/she cannot find a particular business amongst the search results.	S
REQ4.07	The iOS application shall display a warning in case no businesses could be found.	S
REQ4.08	The iOS application shall allow the user to filter the search results further.	C
Category: Video recording, annotating and uploading		
REQ5.00	The iOS application shall allow the user to record short videos only (minimum 5 sec, maximum 10 sec).	M
REQ5.01	The iOS application shall record each video in a square format.	C
REQ5.02	The iOS application shall force the user to select a theme from a predefined theme collection for each recorded video.	M
REQ5.03	The iOS application shall allow the user to add up to 5 hashtags to each recorded video.	M
REQ5.04	The iOS application shall allow the user to write a short social message to further describe the recorded video (150 characters max).	M
REQ5.05	The iOS application shall allow the user to specify pricing information for the recorded video (if applicable to the theme).	C
REQ5.06	The iOS application shall allow the user upload a finished video in the background, keeping it locally stored in an internal database in case there is currently no internet connection.	C
REQ5.07	The iOS application shall allow the user to record, annotate and upload up to 4 videos at the same time.	C
Category: Business page		
REQ6.00	The iOS application shall display basic information (name, location on map, number of videos, categories, and if applicable open hours and address) on each business page.	S
REQ6.01	The iOS application shall allow the user to edit and extend a business's basic information.	C
REQ6.02	The iOS application shall allow the user to bookmark a business.	W
REQ6.03	The iOS application shall allow the user to record a video for the business's showreel.	M

Category: Business page > video showreel		
REQ7.00	The iOS application shall display a showreel of videos.	M
REQ7.01	The iOS application shall always list all available videos in a showreel by default.	M
REQ7.02	The iOS application shall allow the users to swipe through the video showreel.	M
REQ7.03	The iOS application shall always play the current video of the showreel while also showing its metadata information.	M
REQ7.04	The iOS application shall allow the user to filter the showreel by themes.	M
REQ7.05	The iOS application shall allow the user to filter the showreel by hashtags.	C
REQ7.06	The iOS application shall allow the user to order the showreel.	M
REQ7.07	The iOS application shall allow the user to like or report each video in the showreel.	M
Category: User Authentication		
REQ8.00	The iOS application shall allow the user to sign in with the Facebook API.	W
Category: User Profile		
REQ9.00	The iOS application shall display the user's generated videos and businesses.	W
REQ9.01	The iOS application shall display the user's bookmarked businesses.	W
REQ9.02	The iOS application shall display the user's total received points from adding/editing businesses and uploading videos.	W
Category: Leaderboard		
REQ10.00	The iOS application shall display a leaderboard showing the user's points ranking compared to his/her Facebook friends and the voucher price that can be won this month.	W

Non-Functional Requirements

ID	Requirement	Priority
Category: Implementation		
NREQ1.00	The iOS application shall be implemented in Swift 3.	M
Category: User Interface		
NREQ2.00	The iOS application shall work for any iPhone device running iOS10 apart from iPhone 4 and iPhone 4S.	S
NREQ2.01	The iOS application shall conform to Apple's design guidelines for future deployment on the App Store.	S
NREQ2.02	The iOS application shall look professional and include animations to enhance user experience.	S
NREQ2.03	The iOS application shall be consistent with Voxreel's colour scheme (purple, red, white).	S
NREQ2.04	The iOS application shall report to the user if there is a delay in performing tasks.	S
Category: Performance		
NREQ3.00	The iOS application shall take no longer than 5 seconds to load during launching.	S

NREQ3.01	The iOS application shall take no longer than 5 seconds to load when navigating through the app.	S
Category: Usability		
NREQ4.00	The iOS application shall allow users to search for business regardless whether or not the location tracker is working.	S
NREQ4.01	The iOS application shall allow the user to stream videos continuously without many buffering delays.	C
NREQ4.02	The iOS application shall not become unresponsive in case of lengthy processes.	S
NREQ4.03	The iOS application should be intuitive to use for users of any skill level.	S
NREQ4.04	The iOS application shall gracefully handle a loss of internet connection or no internet connection at all.	C
Category: API use		
NREQ5.00	The iOS application shall use the Google Places API for its location autocompletion feature.	M
NREQ5.01	The iOS application shall use Voxreel's RESTful API for any other backend related communication.	M

8.1.2 RESTful API

Functional Requirements

ID	Requirement	Priority
Category: Configuration		
REQ1.00	The RESTful API shall have an endpoint for configuring the iOS application with nearby businesses, a category tree and video themes.	M
Category: Business creation		
REQ2.00	The RESTful API shall have an endpoint for creating new a new business based on a name, between 1 and 3 categories and a location.	M
REQ2.01	The RESTful API shall mark each created business initially as unchecked.	M
Category: Business search		
REQ3.00	The RESTful API shall have multiple endpoints for the various business searching capabilities of the iOS application.	M
Category: Business view		
REQ4.00	The RESTful API shall have an endpoint for retrieving a business's basic information.	M
Category: Video storage and streaming		
REQ5.00	The RESTful API shall have an endpoint for uploading a video and its annotations.	M
REQ5.01	The RESTful API shall have an endpoint for retrieving a business's video metadata.	M
REQ5.02	The RESTful API shall have an endpoint for progressively downloading a video.	M
REQ5.03	The RESTful API shall have an endpoint for liking and reporting a video.	M
Category: User Authentication		
REQ6.00	The RESTful API shall have an endpoint for authentication the user with the Facebook API.	W

REQ6.01	The RESTful API shall have an endpoint for requesting and refreshing authentication tokens.	W

Non-Functional Requirements

ID	Requirement	Priority
Category: Implementation		
NREQ1.00	The RESTful API shall be implemented in Laravel (v 5.3).	M
NREQ1.01	The RESTful API shall store all permanent data in a MySQL database.	M
NREQ1.02	The RESTful API shall obfuscate business and video IDs.	
Category: Performance		
NREQ2.00	The RESTful API shall only return the bare minimum data for each endpoint in order to decrease query times.	M
Category: Integrity		
NREQ3.00	The RESTful API shall mark newly created businesses that have not been checked for their correctness as ‘unchecked’.	M
Category: Usability		
NREQ5.00	The RESTful API shall output its API data in a structured and concise format.	M
NREQ5.01	The RESTful API shall output an appropriate HTTP status code as part of each API response.	M
Category: Security		
NREQ6.00	The RESTful API shall use JWT tokens for authenticating user requests.	W
NREQ6.01	The RESTful API shall set a rate limit to mitigate DOS attacks.	C

8.2 Use Case Document

Use Case	Create Business
ID	UC.01
Description	A user creates a new business.
Primary Actors	User
Secondary Actors	None
Preconditions	The user might have triggered a business search before and did not find the desired business or the user might not have found the desired businesses amongst the list of nearby businesses.
Main Flow	1) The user specifies the details of the new business. 2) The user clicks the 'send' button. 3) The iOS application validates the user input. 4) The iOS application submits the new business to the RESTful API. 5) The RESTful API stores the new business in the database. 6) The iOS application displays a successful notification.
Post Conditions	The user has successfully created a business that can now be search and edited by others.
Alternative Flow	None

Use Case	Edit Business
ID	UC.02
Description	A user edits an existing business.
Primary Actors	User
Secondary Actors	None
Preconditions	The user has navigated to a business page.
Main Flow	1) The user clicks the 'edit' button. 2) The iOS application displays a new page with all the editable business information. 3) The user edits (improves or adds) the business details. 4) The user clicks the 'send' button. 5) The iOS application validates the user input. 6) The iOS application submits the updated business to the RESTful API. 7) The RESTful API updates the stored business accordingly.
Post Conditions	The user has successfully updated a business's information.
Alternative Flow	None

Use Case	Specify Location For Business Search
ID	UC.03
Description	A user specifies a location for a business search.
Primary Actors	User
Secondary Actors	None
Preconditions	The user has navigated to a business search page.
Main Flow	1) The user selects the search bar. 2) The iOS application displays a list of default locations. 3) The user selects a default location (in this case skip step 4 and 5) or enters something in the search bar. 4) The iOS application displays an autocompletion list of location suggestions offered by the Google Places API. 5) The user selects a location from the autocompletion list. 6) The iOS application remembers the location, clears the input and asks the user to specify a keyword or a business category.
Post Conditions	The user has specified a location on which the next business search should be based on.
Alternative Flow	None

Use Case	Business Search By Keyword
ID	UC.04
Description	A user triggers a business search after specifying a custom keyword.
Primary Actors	User
Secondary Actors	None
Preconditions	The user has already specified a location for the business search.
Main Flow	1) The user enters something in the search bar. 2) The iOS application displays an autocompletion list of businesses and business categories.

	3) The user clicks the 'search' button. 4) The iOS application performs the search via the RESTful API. 5) The iOS application displays a loading animation. 6) The iOS application displays the search results.
Post Conditions	The user can see businesses whose names match the specified keyword and are nearby the specified location.
Alternative Flow	None

Use Case	Business Search By Category
ID	UC.05
Description	A user triggers a business search after selecting a business category.
Primary Actors	User
Secondary Actors	None
Preconditions	The user has already specified a location for the business search.
Main Flow	1) The user enters something in the search bar. 2) The iOS application displays an autocompletion list of businesses and business categories. 3) The user selects a business category from the autocompletion list. 4) The iOS application performs the search via the RESTful API. 5) The iOS application displays a loading animation. 6) The iOS application displays the search results.
Post Conditions	The user can see businesses that match the specified category and are nearby the specified location.
Alternative Flow	None

Use Case	Navigate To Business
ID	UC.06
Description	A user directly navigates to a business without performing a search.
Primary Actors	User
Secondary Actors	None
Preconditions	The user has already specified a location for the business search.
Main Flow	1) The user enters something in the search bar. 2) The iOS application displays an autocompletion list of businesses and business categories. 3) The user selects a business. 4) The iOS application fetches the relevant business data via the RESTful API. 5) The iOS application displays a business page.
Post Conditions	None
Alternative Flow	See use case UC.09

Use Case	View Business Search Results
ID	UC.07
Description	A user views the business search results on a map view, followed by a list view.
Primary Actors	User
Secondary Actors	None
Preconditions	The user has triggered a business search.
Main Flow	1) The iOS application displays each business with a marker on the map. 2) The user selects a business marker. 3) The iOS application displays popup view above the marker, showing the business's general information (name, categories and today's open hours) 4) The user clicks on the 'list' button. 5) The iOS application displays each business in a list view. 6) The user scrolls up and down to see the business's general details. *Note: The main flow could also start with the list view and then move on to the map view.*
Post Conditions	None
Alternative Flow	None

Use Case	Navigate To Business
ID	UC.08
Description	A user navigates to a business by selecting a business search result.
Primary Actors	User

Secondary Actors	None
Preconditions	The user has triggered a business search and the iOS application displays the results in either on a map view or in list view.
Main Flow	1) The user clicks the 'show more' button on a business search result. 2) The iOS application fetches the relevant business data via the RESTful API. 3) The iOS application displays a business page.
Post Conditions	None
Alternative Flow	See use case UC.07

Use Case	View Business Details
ID	UC.09
Description	A user views a business's details
Primary Actors	User
Secondary Actors	None
Preconditions	The user has navigated to a business.
Main Flow	1) The iOS application displays the business's basic information. 2) The user can scroll up and down in the content list.
Post Conditions	None
Alternative Flow	None

Use Case	Watch Business Showreel
ID	UC.10
Description	A user watches and browses through a business's video showreel.
Primary Actors	User
Secondary Actors	None
Preconditions	The user has navigated to a business.
Main Flow	1) The user clicks on the 'showreel' button. 2) The iOS application fetches video metadata from the RESTful API. 3) The iOS application shows the business's showreel page. 4) The iOS application streams the first video of showreel via the RESTful API and displays all its relevant metadata information underneath the video. 5) The user swipes right or left to see other videos in the showreel. 6) The iOS application streams the relevant videos accordingly.
Post Conditions	None
Alternative Flow	None

Use Case	Search Business Showreel
ID	UC.11
Description	A user filters a business's video showreel by themes or hashtags.
Primary Actors	User
Secondary Actors	None
Preconditions	The user has navigated to a business's showreel.
Main Flow	1) The user clicks on the 'setting's button. 2) The iOS application shows a list of theme and hashtag filters that are relevant the business's showreel. 3) The user selects only the themes and hashtags of the videos that he/she is not interested in seeing. 4) The user clicks the 'apply' button 5) The iOS application adjusts the showreel accordingly to the filters. 6) The iOS application plays the first video of the showreel.
Post Conditions	The business's video showreel has been customised to the user's needs.
Alternative Flow	None

Use Case	Order Business Showreel
ID	UC.12
Description	A user orders the videos in the business's showreel.
Primary Actors	User
Secondary Actors	None
Preconditions	The user has navigated to a business's showreel.

Main Flow	1) The user clicks on the 'setting's button. 1) The iOS application shows a list of ordering options. 2) The user selects a new ordering option. 3) The user clicks the 'apply' button. 4) The iOS application orders the videos in the showreel accordingly. 5) The iOS application plays the first video of the showreel.
Post Conditions	The business's video showreel has been customised to the user's needs.
Alternative Flow	None

Use Case	Like Video From Business Showreel
ID	UC.13
Description	A user likes a video from a business's showreel.
Primary Actors	User
Secondary Actors	None
Preconditions	The user has navigated to a business's showreel.
Main Flow	1) The user clicks on the 'like' button underneath the currently playing video. 2) The iOS application submits the like to the RESTful API. 3) The RESTful API stores the like and update's the video's like counter.
Post Conditions	A video's like counter has been incremented.
Alternative Flow	None

Use Case	Report Video In Business Showreel
ID	UC.14
Description	A user reports a video from a business's showreel.
Primary Actors	User
Secondary Actors	None
Preconditions	The user has navigated to a business's showreel.
Main Flow	1) The user clicks on the 'report' button underneath the currently playing video. 2) The iOS application opens up a text box. 3) The user explains the reason for reporting the video. 4) The user clicks the 'send' button. 5) The iOS application validates the user's input. 6) The iOS application submits the report to the RESTful API. 7) The RESTful API stores the report.
Post Conditions	A new video report is available for the Voxreel team to read.
Alternative Flow	None

Use Case	Record And Upload Video
ID	UC.15
Description	A user records a short video, annotates it with useful metadata, and then uploads it.
Primary Actors	User
Secondary Actors	None
Preconditions	The user has navigated to a business or to the list of nearby businesses.
Main Flow	1) The user clicks the 'add video' button. 2) The iOS application shows a video recorder. 3) The user records a video by holding down a finger on the screen. 4) The user selects a theme for the video. 5) The user provides metadata information about the video (hashtags, message and pricing information if applicable). 6) The user clicks the 'upload' button. 7) The iOS application submits the video and its metadata information to the RESTful API. 8) The RESTful API stores the video and its metadata information.
Post Conditions	A new video is now available for others to watch.
Alternative Flow	None

Use Case	View Nearby Businesses
ID	UC.16
Description	A user views the nearby businesses around him/her.
Primary Actors	User

Secondary Actors	None
Preconditions	None
Main Flow	1) The user navigates to the list of nearby businesses. 2) The iOS application periodically updates the nearby business list via the RESTful API. 3) The user scrolls up and down to see the business's general information.
Post Conditions	None
Alternative Flow	None

8.3 System Manual

8.3.1 Accessing Code

The final project code is stored on a private BitBucket repository, holding the code for both the iOS application and RESTful API. In the interest of gaining access to the repository, email andreas.l.rauter@gmail.com to be granted permission.

The link to repository can be found on https://bitbucket.org/SickAustrian/individual-project-andreaslukasrauter.

8.3.2 Setting Up Code & Environment

RESTful API

The RESTful API is based on the Laravel framework (version 5.3), which has a few system requirements. Since all of these requirements are satisfied by the Laravel Homestead virtual machine, I recommend you to install and use Homestead as your local Laravel development environment. The official installation guide can found on https://laravel.com/docs/5.3/homestead. Make sure you go through the chapters "Introduction" and "Installation & Setup" to properly configure the RESTful API inside the Homestead environment. Note that you can find the API codebase inside the "RESTful API" directory of the project repository.

1) At this point you should have already installed Homestead and configured it with the RESTful API. Additionally, you should have access to the API URL (e.g. "http://restful-api.app/api").

2) Run the terminal command ***composer update*** inside the "RESTful API" directory to install any composer dependencies required for the API. Note that this requires you to have Composer installed on your machine.

3) Download and install ngrok from https://ngrok.com/. This tool is necessary to allow the iOS application (running on the iPhone) to communicate with the RESTful API (running on local server).

4) Create an HTTP tunnel to the API URL by running the following terminal command:
ngrok http --host-header=restful-api.app restful-api.app:80
Remember the HTTPS forwarding link (e.g.: "https://749f3b04.ngrok.io") that is provided to you as this is the tunneled URL that the iOS application can use to communicate with the API (see red rectangle in figure 8.3.1)

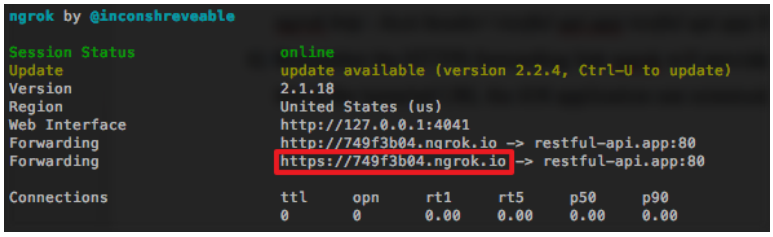

Figure 8.3.1 - Tunneled HTTPS URL

5) Open the ".env" file inside the "RESTful API" directory and replace "TUNNELED-API-URL" in line 34 with the actual tunneled API URL from the previous step (e.g. "https://749f3b04.ngrok.io/").

6) Navigate to your Homestead directory and ssh into your vagrant box using ***vagrant ssh***. If your virtual machine is not booted up yet, run the ***vagrant up*** first. Once you connected to the virtual machine, navigate to the "RESTful API" repository. Run ***php artisan migrate --seed*** to finally create all database tables and seed the database with mock businesses.

iOS application

To perform the following steps, it is assumed you have an Apple computer device with Xcode IDE installed as well as an iPhone device and an Apple developer account to run the application.

1) Open Xcode and press the "Open another project" button, resulting in a file selection pop up to appear. Navigate to the downloaded project repository, select the "iOS application" directory and press "Open". This will open up the iOS application as a project inside Xcode.

2) Download and install CocoaPods from https://guides.cocoapods.org/using/getting-started.html. CocoaPods is the dependency manager I used to import the required libraries discussed in 2.4.1.

3) Open the terminal, navigate to the "iOS application" directory and then run ***pod install.*** This will install all required dependencies for the iOS application to run.

4) Open up Xcode again and sign the iOS application with your Apple developer account. Refer to https://developer.apple.com/support/code-signing/ for more information.

5) Select the "AppConstant.swift" file inside the Xcode IDE and replace "TUNNELED-API-URL" in line 64 by the actual tunneled API URL.

6) Connect your phone with your computer.

7) Inside the Xcode IDE, choose your device and click the run button to launch the application on your phone (top left corner).

8.4 User Manual

This section highlights the key tasks of the resulting iOS application.

8.4.1 Start Application

Figure 8.4.1 shows the application's launching screen when it boots up, whereas figure 8.4.2 demonstrates the application's home screen (i.e. business search screen) after the configuration process has been completed. You can see your current position as a blue animated circle on the map assuming you have allowed the application to track your location. For clarity, the items in the navigation bar illustrate the following pages (left to right): (1) business search page, (2) add video page, and (3) profile page.

Note: The navigation bar item not implemented is highlighted by a blue circle (profile page).

Figure 8.4.1 Launch screen

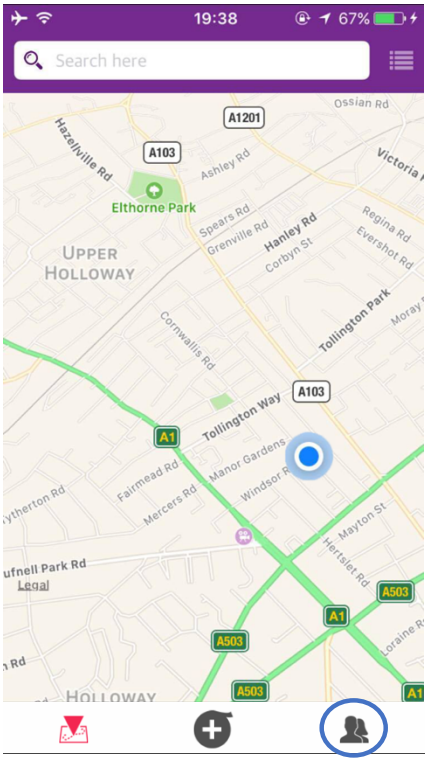

Figure 8.4.2 - Home Screen

Steps:

1) Select the Voxreel application icon on your iPhone's home screen.

2) The application will boot up and immediately show an animating launching screen while it performs its configuration process.

3) When the app launches for the first time, it will ask you to grant location tracking permissions. Make sure you do allow the application to track your location, otherwise you cannot later on search venues by your current location nor does the application identify nearby businesses around you.

8.4.2 Search Businesses

Specify Location

Figures 8.4.3 to 8.4.6 demonstrate the search view that appears whenever you specify new details for a business search. If you have never triggered a search before, you are initially asked to specify the location for the search (see figure 8.4.3). Otherwise, the application will reuse the specified location form the last business search.

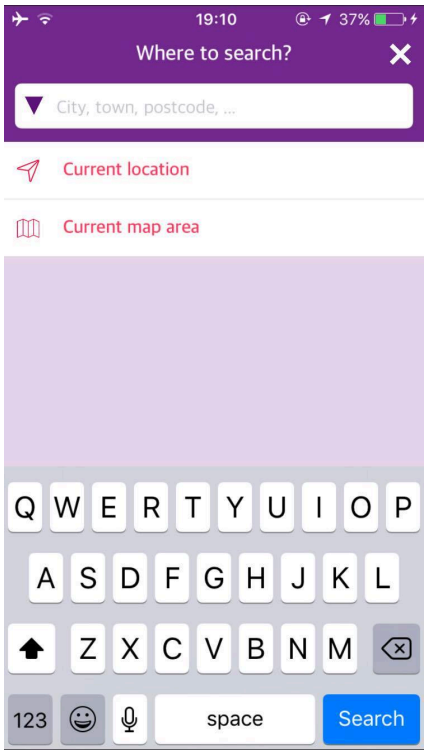

Figure 8.4.3 - Initial location specification

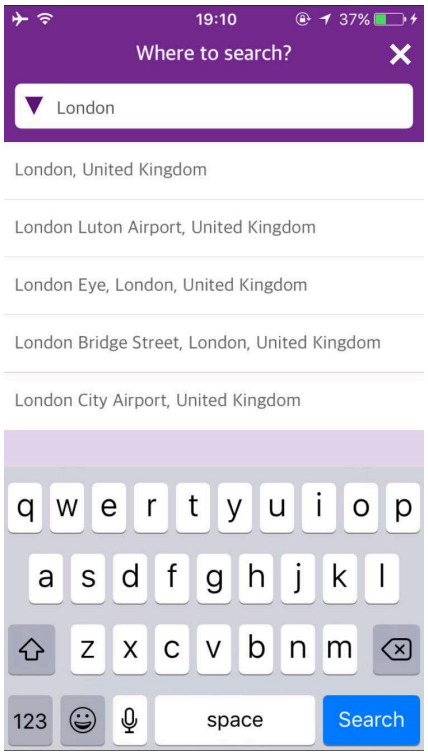

Figure 8.4.4 - Location autocompletion

Steps:

1) Tab the search bar on the business search page.

2) A new screen appears for you to specify the location for your next search (figure 8.4.3). By default, the application provides you with two options to choose from.

3) While you enter a location description, an autocompletion list of location suggestions is shown. Figure 8.4.4 illustrates this for the user input “London”.

4) Once you have found the location you are looking for, tap its corresponding list item.

Specify keyword or category

Figure 8.4.5 and 8.4.6 show the search view after you have specified a location, “London, United Kingdom” in this example.

Note: You can edit the location by selecting the location label, consequently navigating you back to the previous step.

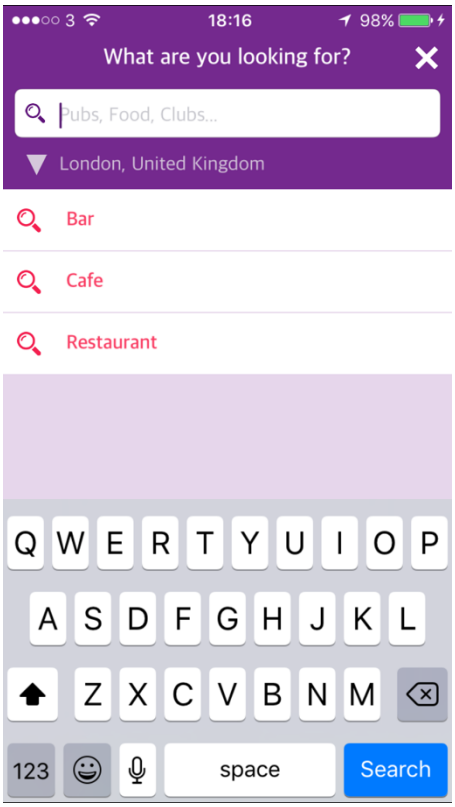

Figure 8.4.5 – Initial keyword or category specification

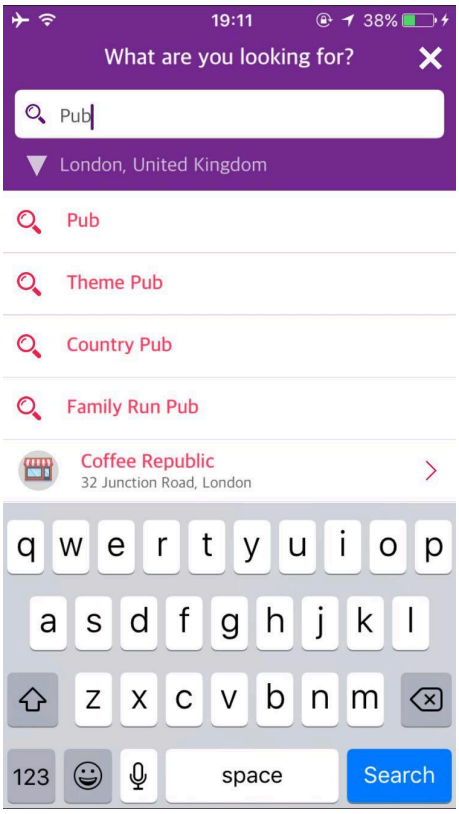

Figure 8.4.6 - Business and category autocompletion

Steps:

5) Initially you are presented with default categories, i.e. level 1 categories form the business category tree (figure 8.4.5).

6) While you enter your input in the search bar, an autocompletion list comprising business and category suggestions is shown (figure 8.4.6).

7) Select any item from the autocompletion list. If you tab a category, the application will perform a business search by category. The results will then be presented on either a map or list view within the business search page (see 8.4.7). If you decide to select a business item instead, the application will navigate you straight to a business page (see 8.4.7). Alternatively, you can also tap the ‘Search’ button and the iOS application will then either perform a search by name or category (only if your input matches a category name or no input is specified at all).

8.4.3 Browse Businesses

Once you have triggered a business search, the search results are illustrated on either a map view (figure 8.4.7 and 8.4.8) or as part of a list view (figure 8.4.9), depending on which option is currently enabled.

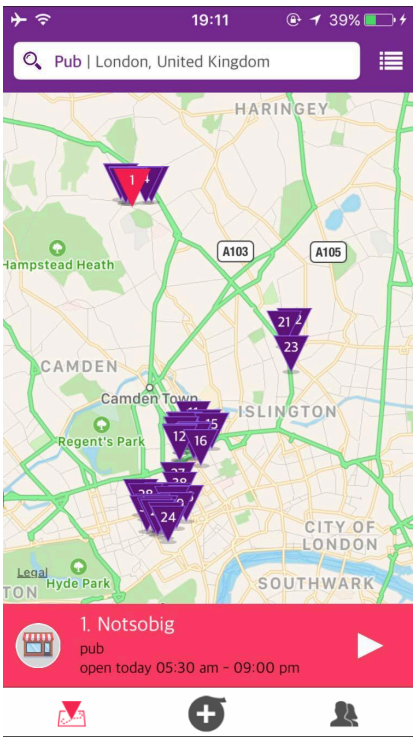

Figure 8.4.7- Search results in map view

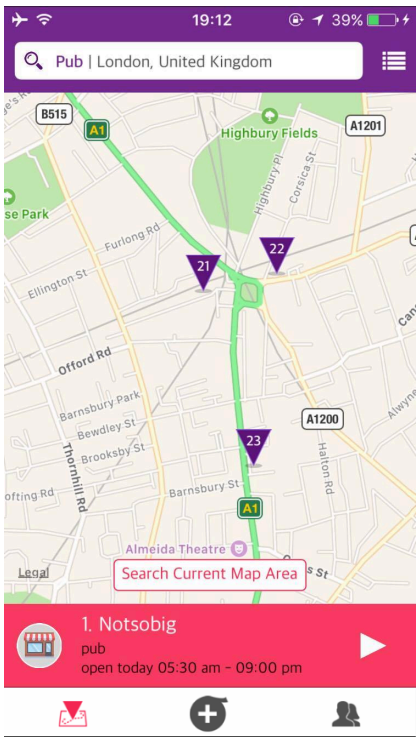

Figure 8.4.8 - Search current map view

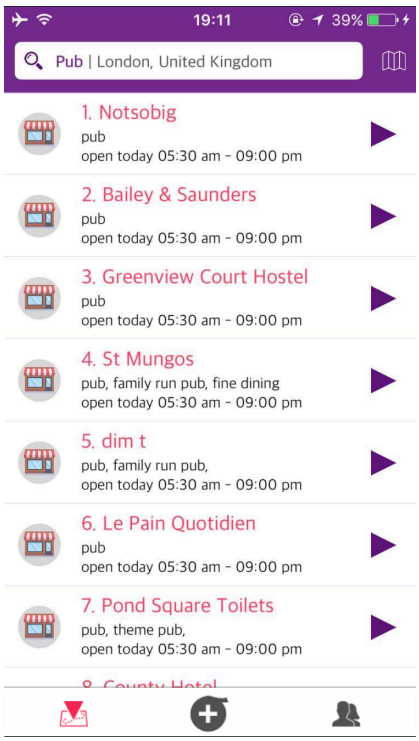

Figure 8.4.9 - Search results in list view

Highlights:

- Map view:
  - To browse through businesses, you can click between business map markers or swipe left and right inside the red box.
  - Use your hand gestures to navigate around and zoom in and out of the map view in order to gain a better understanding of the businesses' locations.
  - Once you begin navigating around the map view, a button appears above the red box, allowing you to search for more businesses inside the current map view (figure 8.4.8).
- List view:
  - Scroll up and down the list to obtain a quick overview of all the businesses that were found.
  - At the bottom of the list, there is button that allows you to create a new venue assuming you could not find the business you were looking for.
- Click the button right next to the search bar if you want to switch between map and list view.
- To further explore a business, you can select the triangle inside a business item. The application will then navigate you to the corresponding business page (see 8.4.7).
- By tabbing the search bar, you can modify your current search specifications and perform another business search.

8.4.4 Nearby Businesses

Figure 8.4.10 and 8.4.11 demonstrate the add video page, listing the nearby businesses around you. Those are the businesses you are currently allowed to record a video for, thus preventing you from uploading random video content to any venues.

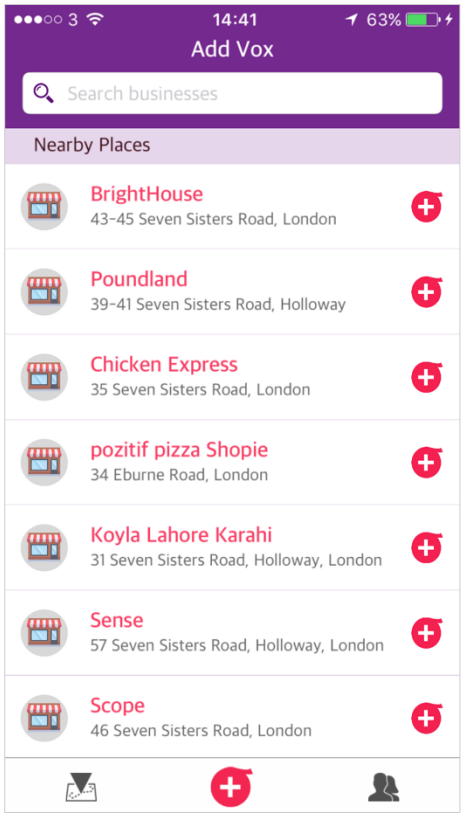

Figure 8.4.10 - Add video page

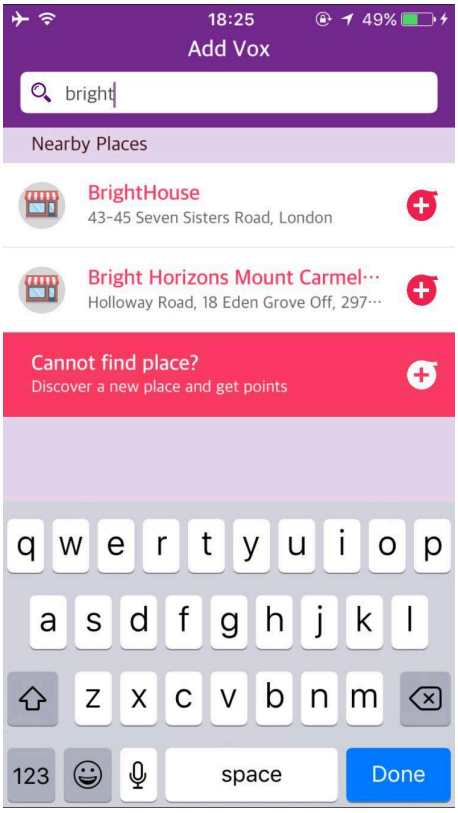

Figure 8.4.11 - Search nearby businesses

Highlights:

- To capture a video for a business, select any of the cross buttons. The application's video recorder will then appear (see 8.4.6).
- You can tab the search bar and enter the name of the business you are looking for.
- If you are not able to find a nearby business, select the plus button inside the red box to quickly create the missing venue (figure 8.4.11). Refer to 8.4.5 for further information.

8.4.5 Create Businesses

Figure 8.4.12 to 8.4.14 demonstrate the process of adding a new business.

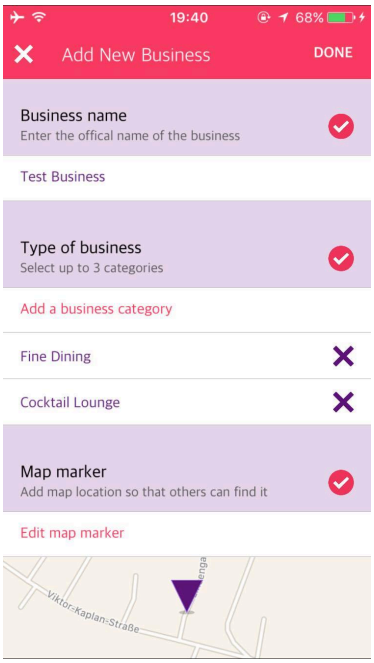

Figure 8.4.12 - New business page

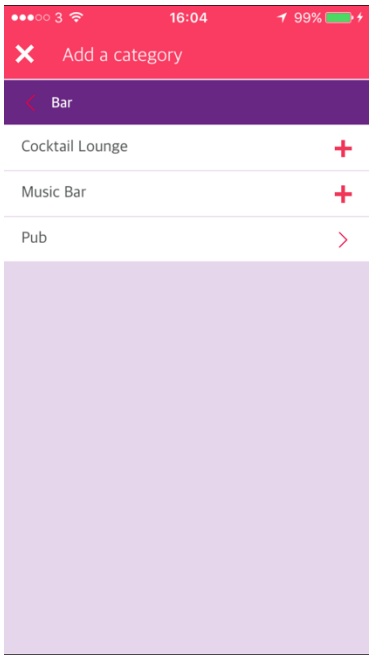

Figure 8.4.13 – Category selection

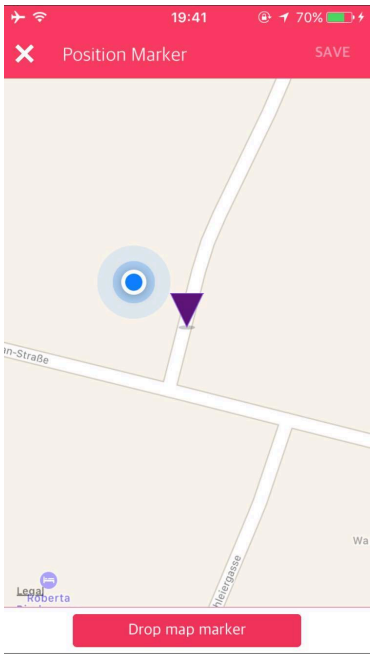

Figure 8.4.14 – Setting business location

Steps:

1) Enter the name of the business.

2) Select between 1 and 3 categories using the "Add a business category" button. Each time, a new page appears, allowing you to pick a category from a category tree as demonstrated in figure 8.4.13. Afterwards, feel free to remove any category by clicking the purple cross button of a category item.

3) Specify the location of the new business using the "Add map marker" button. Again, the application takes you to a separate page where you have to specify the location of the new venue (figure 8.4.14). Navigate around the map to properly position the marker and make sure to zoom in adequately in order establish an accurate location. Once you found the desired marker position, click the "Drop map marker" button and save the location by selecting the "SAVE" button. If at any time, you wish to edit the location marker, click the "Edit map marker" button and repeat the same process.

4) Once you filled out all necessary input fields select the "DONE" button to upload the new business.

8.4.6 Capture Videos

Recording video

Figure 8.4.15 to 8.4.17 demonstrate the video recording process, the first step in capturing a video.

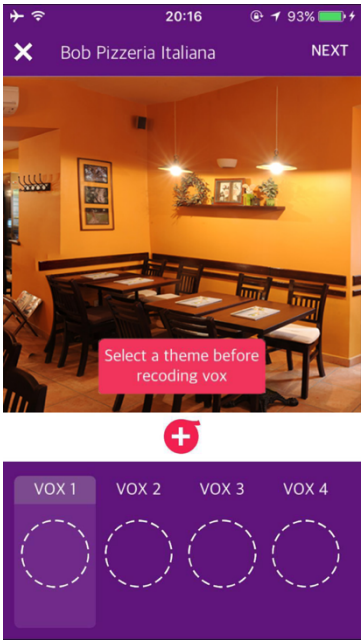

Figure 8.4.15 Initial video recorder

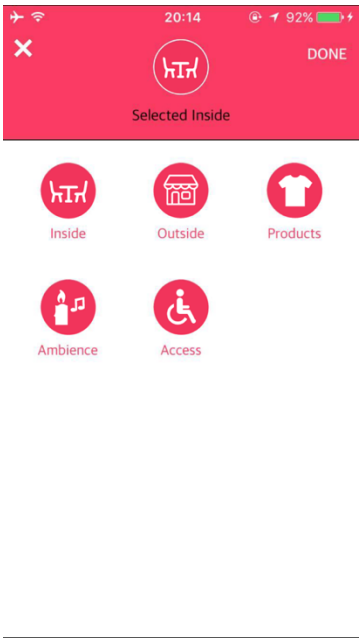

Figure 8.4.16 - Theme selector

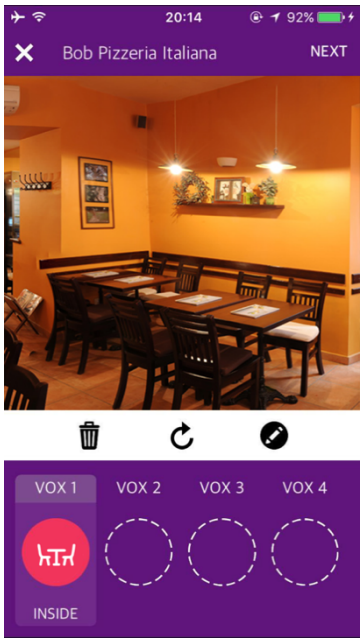

Figure 8.4.17 - Recorded video preview

Steps

1) To record a new video, you first have to select a theme as demonstrated in figure 8.4.15. To do so, click the cross button and the theme selector page will appear (figure 8.4.16).

2) Select a theme and then press the 'DONE' button to confirm.

3) Hold down your thumb inside the video square to record a video. A red progress bar will appear above the white menu, illustrating the currently recorded length of the video (maximum 10 seconds). The video recorder will notify you in case the recorded video ended up being too short.

4) Once you recorded the video, the video will be quickly processed and cut down to a square format. Afterwards, the video will begin playing in a loop as indicated in 8.4.17. It is important to note that, you can perform the following operations on a recorded video using the white menu box: delete the video by clicking on the trash icon, redo the video by selecting the reply icon or change the video theme by tabbing the edit icon.

5) Repeat step 1 to 4 if you wish to record more videos (maximum 4 videos at the time).

6) Press the "NEXT" button to start annotating your video(s).

Note: Currently there are only a few available themes, hence the limited amount choices listed in figure 8.4.16. Furthermore, you can record a video and then specify the theme right after. To do so, skip step 1 and perform step 3 prior to step 2.

Annotate and upload video

Each video you recorded undergoes an annotation process (figure 8.4.18 and 8.4.19), followed by uploading it to the server (figure 8.4.20).

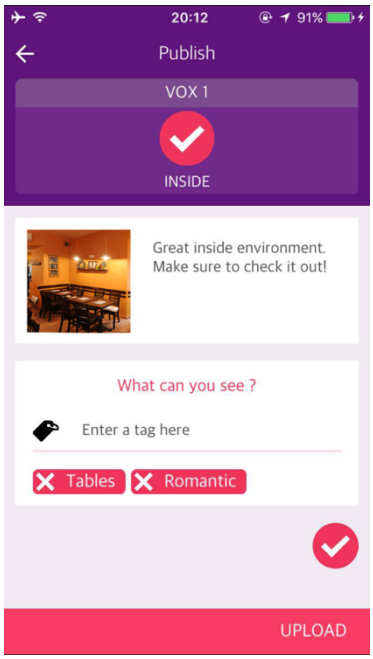

Figure 8.4.18 – Annotation before marking video as ready

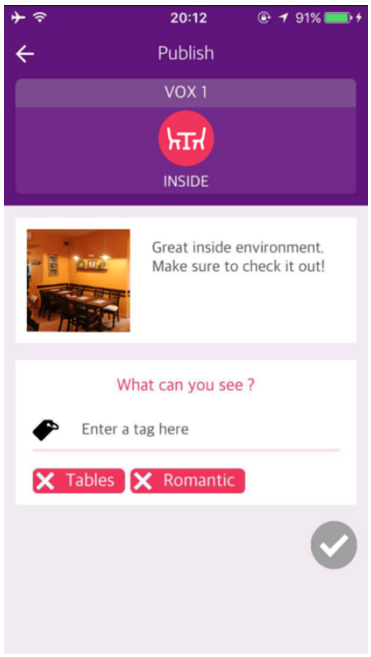

Figure 8.4.19 – Annotation after marking video as ready

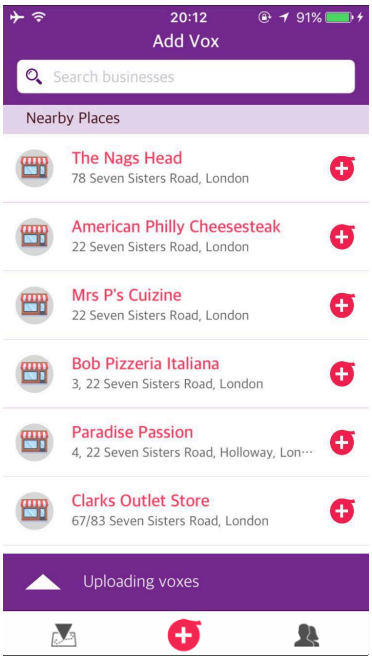

Figure 8.4.20 – Uploading process

Steps

7) You can specify a short message for each video (maximum 100 characters). This message will then be visible when other users watch your video as part of the business showreel (see 8.4.7).

8) You can further enter up to 5 hashtags. This will make the video more visible when searching the business showreel for hashtags (not implemented yet).

9) You can also annotate the video with a price label. This step depends on the video theme though. For instance, an inside theme does not support a price label whereas a food theme does. In case of a food theme, the application would ask you to select the amount of money you spent on food. Again, figure 8.4.18 and 8.4.19 demonstrate the annotation process for an inside themed video, thus the application would not ask you to provide any pricing details.

10) After you finish annotating the video, press the checkmark icon and the "UPLOAD" button appears. Figure 8.4.18 and 8.4.19 clearly demonstrate the annotation screen before and after marking the video as ready.

11) Finally select the "UPLOAD" button to initiate the upload. The application will navigate you back to the add record page and indicate the uploading process via a purple box popping up from underneath the navigation menu. You are not required to wait for the data transfer to finish as the application will upload your video in the background. Once the upload has succeeded, the purple box will disappear. In case of a failure, a report message is being displayed inside the purple box with a button that allows you to reinitiate the upload.

Note: Step 7 to 9 are optional. You are not forced to annotate the video if you do not prefer to. Nonetheless, annotating the video increases its value, therefore making it more insightful.

8.4.7 Explore Businesses

Figure 8.4.21 shows a traditional venue page you would see after navigating to a particular business.

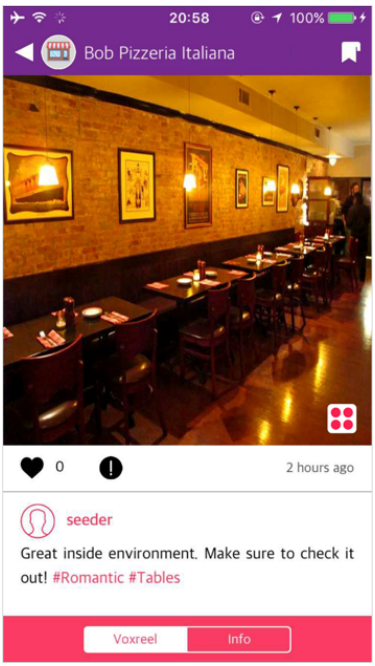

Figure 8.4.21 – Illustrates the showreel for a business named "Bob Pizzeria Italiana"

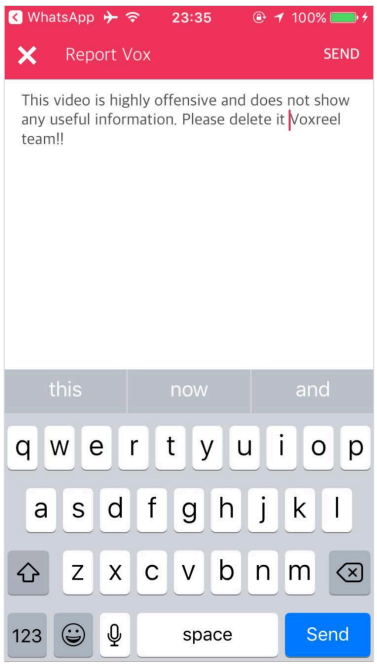

Figure 8.4.22 – Demonstrates the video report page

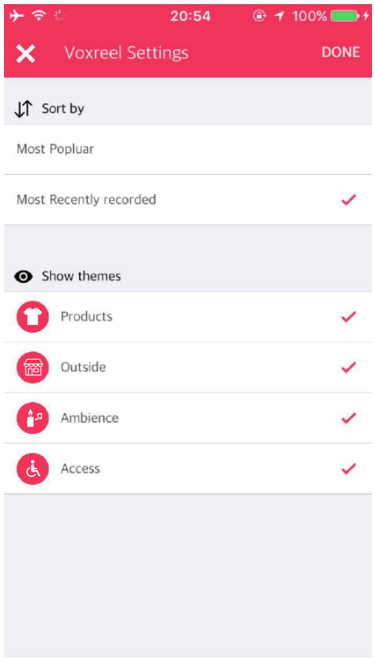

Figure 8.4.23 - Shows a business's showreel settings

Highlights

- You can watch the business showreel by selecting the "Voxreel" tab (figure 8.4.21).
- You can swipe left and right to browse through the showreel. The active video immediately plays in a loop while listing its metadata and annotated information underneath.
- You can like/unlike each video by tabbing the heart icon.
- You can report each video by selecting the warning icon. A new page will appear, allowing you to compose and upload a short message to the Voxreel team (figure 8.4.22).
- You can filter and order the business showreel by selecting the white box icon above the currently playing video. The application will present you with a settings page where you can select a number of sorting or filtering options you would like to apply to the business showreel (figure 8.4.23).

Note: Currently the application does not allow you to bookmark a business or view the business's basic information (location, open hours, etc.). Due to my limited amount of time, I decided to implement those feature as part of my future work.