INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA

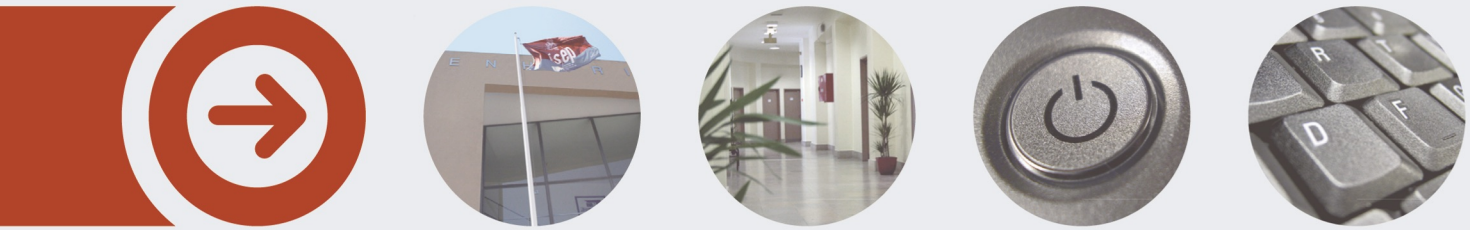



iOS Technologies & Frameworks

NINO SANTOS VITALE
Outubro de 2016

POLITÉCNICO
DO PORTO

INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA



iOS Technologies & Frameworks

NINO SANTOS VITALE
Outubro de 2016

POLITÉCNICO
DO PORTO

iOS Technologies & Frameworks

Nino Santos Vitale

A dissertation submitted in partial fulfillment of the requirements for the degree of Master of Science, Specialization Area of Computer Systems

Supervisor: Dr. Paulo Baltarejo Sousa

Porto, October 23, 2016

Abstract

Apple's mobile platform — iOS — currently generates the largest amount of revenue out of all mobile app stores. The majority of iDevices run the latest major iOS version (iOS 10) due to Apple users' tendency to update their devices. Consequently, iOS developers are pressured into keeping their apps up to date.

Advantages to updating apps consist of new features and adapting apps to the platform's hardware and software evolution. However, this does not always happen. There are apps, some popular (with many users), which either receive slow updates, or not at all. The main consequence of developers not updating to the latest tendencies (i.e. user interface or API changes) is the degradation of their apps' user experience. This subpar user experience leads to a decrease in the number of installs (and sales) and a search for alternatives that have been updated to support the latest firmware iteration fully.

We identified a common pattern amongst ten apps which have subpar reviews on the App Store: excessive battery consumption and lack of user onboarding were just a few of the issues. Above all, almost all those apps belong to the top 1% of apps (which generate 94% of the App Store's revenue), so the lack of focus on the user experience is unfortunate considering their massive user bases.

We listed the available resources for those wanting to develop or improve iOS apps. Given these requisites, we studied the possibility of developing a mobile app that adopted good engineering practices and, above all, focused on delivering an excellent user experience in a given timeframe of six months.

The app's idea consisted of a wish list management app called Snapwish that allows the user to take photos of objects they want, create wish lists, and share them with family and friends. The app allows for offline usage, with data syncing automatically (in real-time) without user intervention when the app's Internet connection is present.

We tested Snapwish thoroughly to measure the quality of its implementation. Profiling helped assert that core metrics like CPU and memory usage, network data requests and energy consumption were within acceptable values while unit and user interface tests served to validate our code functionally. Furthermore, our team of five beta testers provided valuable feedback and suggestions.

Ultimately, the six-month timeframe proved to be insufficient in regards to a release on the App Store, as Snapwish remains in the latter beta stages at the time of writing. This delay is mostly attributed to a lengthy testing process. Thus, we plan on releasing it in the first trimester of 2017.

Keywords: iOS apps, iOS development, technologies, frameworks, App Store, wish lists.

Resumo

Hoje em dia, a plataforma móvel da Apple — iOS — é a que tem maior revenue em aplicações móveis. A maior parte dos dispositivos móveis iOS corre a versão mais atual (iOS 10), devido à tendência dos seus utilizadores em atualizar o sistema operativo com frequência. Consequentemente, os desenvolvedores da plataforma são pressionados para manterem as suas apps atualizadas.

Algumas das vantagens das atualizações consiste em adicionar novas funcionalidades e adaptar as apps à evolução do hardware e do software da plataforma. Contudo, isto nem sempre se verifica. Existem muitas apps, algumas "populares" (com muitas instalações) cuja atualização demora ou não acontece. A principal consequência da não atualização das apps às tendências atuais, quer em termos de interação, quer em termos de mecanismos de proteção de dados, consumo de bateria e outros, é a degradação da experiência de quem as utiliza, consequentemente, a diminuição do número de instalações (e vendas) e a crescente procura de alternativas que tenham estes princípios em conta.

Foi identificado um padrão comum em dez aplicações cujas classificações na App Store são medíocres: um consumo exagerado de bateria e falta de *user onboarding* foram apenas alguns dos problemas. Acima de tudo, quase todas pertencem ao 1% de aplicações que geram 94% das receitas da App Store. A falta de foco na experiência do utilizador é infeliz considerando as enormes bases de utilizadores dessas aplicações.

Foram listados os recursos disponíveis para quem pretende desenvolver ou melhorar uma aplicação iOS. Dadas essas premissas, foi estudada a possibilidade de desenvolver uma aplicação móvel que adote boas práticas de engenharia e, acima de tudo, foque na experiência do utilizador, num período de seis meses.

A ideia para a aplicação consistiu num gestor de listas de desejos designada Snapwish que permite tirar fotos de objetos que o utilizador deseja, criar listas, e partilhá-las com amigos e familiares. Além disso, a app permite o uso offline e os dados são sincronizados em tempo real sem intervenção do utilizador quando a app dispõe de uma conexão à Internet.

A nossa aplicação foi testada extensivamente para medir o nível de qualidade da sua implementação. O *profiling* ajudou em constatar que métricas fundamentais como o consumo de CPU e memória, pedidos de dados de rede e de consumo de energia (bateria) estavam dentro dos parâmetros aceitáveis. Além disso, uma equipa de cinco *beta-testers* contribuiu com comentários e sugestões de grande valor.

Em última análise, o prazo de seis meses revelou-se insuficiente em relação ao lançamento da app na App Store. O Snapwish permanece numa fase *beta* avançada (no momento da escrita desta tese). Este atraso é principalmente atribuído a um extenso processo de testes. Assim, pretendemos lançar a aplicação no primeiro trimestre de 2017.

Palavras-chave: Aplicações iOS, desenvolvimento iOS, technologias, *frameworks*, App Store, listas de desejos.

Contents

List of Figures

List of Tables

List of Source Code

List of Acronyms

ABI	Application Binary Interface
AHP	Analytic Hierarchy Process
API	Application Programming Interface
ARPU	App Revenue Per User
ATS	App Transport Security
BaaS	Backend as a Service
BAR	Background App Refresh
BDD	behavior-driven development
CI	consistency index
CLI	command-line interface
CPU	Central Processing Unit
CR	consistency ratio
CVC	collection view controller
DSC	Dollar Shave Club
EFF	Electronic Frontier Foundation
EU	European Union
GB	Gigabyte
GCD	Grand Central Dispatch
GPU	Graphics Processing Unit
GUI	graphical user interface
HIG	Human Interface Guidelines
HTTP	HyperText Transport Protocol
HTTPS	HyperText Transport Protocol Secure
IAP	in-app purchases
IDE	Integrated Developer Environment
JPEG	Joint Photographic Experts Group
JSON	JavaScript Object Notation
KVO	Key-Value Observing
MAU	Million Active Users
MB	megabyte
MVC	Model-view-controller

NoSQL	"non SQL"
PNG	Portable Network Graphics
RAM	Random Access Memory
regex	regular expression
REST	Representational State Transfer
SDK	Software Development Kit
SQL	Structured Query Language
ToS	Terms of Service
TVC	table view controller
UC	use case
UI	User Interface
UID	unique ID
URL	Uniform Resource Locator
UUID	universally unique identifier
UX	user experience
VC	view controller
VC	value for the customer
WWDC	Worldwide Developers Conference

Chapter 1

Introduction

This chapter presents a brief context and problem description to help understanding of the topic and the issue at hand. It also covers how the problem will be tackled (i.e., methodology): questions that guide the research, value analysis of a solution, and proposed goals. These are the cornerstones for a viable working solution.

1.1 Context

The release of the iPhone (Apple's line of smartphones) in 2007 spurred a revolution in the computer industry. Multi-Touch (multi-finger gestures) and the 3.5" screen it shipped with were key in creating new user experiences, allowing the iPhone to stand out from other smartphones.

When third-party developers got on board with the release of the App Store (2008), Apple's mobile business skyrocketed. This growth eventually led to the announcement of the iPad (2010), a new product category aimed at both entertainment and productivity. The increased screen size allowed developers to present more content at once.

Apple followed up the success of the iPad and iPhone line with the iPad mini (2012), larger-sized iPhones (2014) and the iPad Pro (2015).

Nowadays, these devices are more capable than ever and come with a variety of hardware and software features that many users integrate into their daily lives seamlessly, such as Touch ID (fingerprint authentication), Apple Pay (contactless payments) and 3D Touch (pressure-sensitive displays).

1.2 Problem

Over half of all Apple mobile devices (54%) are running iOS 10 as of October 2016 [1]. Users' tendency to update iDevices (iPhone, iPad and iPod touch) puts pressure on developers to consistently update their apps[1] and maintain a satisfactory — if not great — level of commitment, ranging from technical support and fixing bugs to introducing new features that users desire. For users, there are many advantages to updating apps: bug fixes, performance improvements, and new features.

[1] In the context of this document, the term "app" will be used to refer to a mobile application.

There should be a reasonable cadence between app releases, and this is particularly the case when it coincides with the release of a major version of iOS (Apple's mobile operating system). Unfortunately, many top charted apps receive belated or minor updates after these major releases. It can lead to user dissatisfaction that is voiced with negative App Store (Apple's store for mobile applications) reviews and other criticism, which in turn reduces the retained user count and increases the search for alternatives that satisfy users' needs.

It is fundamental for popular apps to retain their user base. Losing users is much easier than the process it took gaining them, and regaining is almost impossible once they find a suitable alternative. Big-name apps (typically from large corporations) need to focus on the user experience over profits and margins because, ultimately, keeping a satisfied user base is the key to more success.

1.3 Approach

We need to explain how developers can benefit by updating their apps to support the latest technologies and hardware features, as well as showcasing how new developers can learn and build apps for iOS.

To do this, we break the issue down into two parts. The first analyzes a series of apps that are popular offenders to determine a consistent list of problems: excessive battery consumption, failure to adhere to Apple's interface guidelines, lack of updates, and lack of updates just to mention some. The second part showcases the development of an iOS app from scratch to attest the possibility it can adhere to guidelines in a limited timeframe (six months).

The self-proposed idea was to create a **wish list management app** because it provides an opportunity to create value in a small market where few solutions already exist.

1.3.1 Research questions

Apple provides plentiful resources for developing iOS apps, in addition to their annual best practices videos at the Worldwide Developers Conference (WWDC) [2].

One might then question why is it that popular apps are the ones most neglected. The large user bases these apps possess should warrant developers' utmost attention, so there must be other factors that lead to subpar updates and bad user experiences and reviews. This leads to the main question this dissertation tries to answer:

> **Main question**: *Is it feasible to build (and maintain) an iOS app in a reasonable timeframe that follows good engineering practices and focuses on the user experience?*

Note that "reasonable timeframe" depends on the size of the development team and the number of apps to develop/maintain. This dissertation will focus on a single app with just one developer in a six-month timeframe from the original idea to its App Store submission.

Issues with team scaling are beyond the scope of this dissertation due to their complexity. It is up to the project manager to define team goals and objectives for each project and make sure they are viable, yet fairly ambitious.

The primary question above inevitably leads to others which are directly or indirectly related. We call them subquestions (SQ)[2], and they are the following:

SQ-1: *Which factors do developers become most complacent about when creating/-maintaining apps?*

SQ-1.1: *Which apps are the main offenders?*

SQ-2: *Which are the best engineering practices for developing iOS apps?*

SQ-3: *Is Swift (programming language) mature, or should we continue developing in Objective-C?*

SQ-4: *What resources (frameworks/technologies/tutorials/documentation) exist to aid iOS development?*

SQ-4.1: *Will we use a backend provider, or build our own?*

SQ-4.1.1: *How many iOS-centered backend providers are there?*

SQ-5: *How do we comply with Apple's interface guidelines?*

SQ-5.2: *Which design patterns are most prevalent in iOS development?*

SQ-5.1: *How do we build a responsive and appealing user interface?*

SQ-6: *How do we test our code?*

SQ-6.1: *How do we test the user interface?*

SQ-6.2: *How can we use beta-testers to receive valuable feedback?*

SQ-7: *How do we submit an app to the App Store?*

SQ-7.1: *Are there costs involved?*

SQ-7.2: *How can we increase exposure of our app?*

These subquestions are answered throughout this dissertation. Chapter 3 answers the first four (SQ1–SQ4), while Chapter 4 deals with SQ5. SQ6 and SQ7 are covered in depth in Chapter 6. The conclusion will serve as a basis to answer the primary question.

1.4 Value analysis

The creation of a value proposition[3] starts with defining an idea for the app. We decided to create a wish list management app that allows users to take photos of items they desire, create wish lists, and share them with friends.

We then defined the longitudinal perspective on value for the customer (i.e., benefits and sacrifices in each of the four stages), various types of negotiation scenarios that can occur, and the Business Model Canvas which supports our business model.

[2]While these questions try to be as generic as possible to apply to most use cases, some of them will be specific to our wish list management app.

[3]Covered in-depth in Chapter 2.

Lastly, the Analytic Hierarchy Process (AHP) technique is applied to help users choose between a variety of sample apps according to a series of criteria (e.g., price, reviews, User Interface (UI) design).

1.5 Objectives

We want to understand why larger companies have issues with maintaining and updating their apps, and how we can propose a way to fix this (or, at least, mitigate it as much as possible).

Furthermore, we will showcase that despite all the issues, a single iOS developer can still create value for this platform by making use of available tools and resources[4].

1.6 Motivation

The **main question** focuses on whether the user experience needs to be at the heart of the development process for any app. Focusing on the end user instead of just an app's profitability means developers (and designers) can put out apps which concentrate more on what the user wants than what the creators believe they want.

The idea of this dissertation stemmed from users' growing dissatisfaction with popular apps that either received tardy updates (e.g., Whatsapp), or not at all. Quality is imperative for Apple, and their care shows users that the platform can be trusted. Between the two million apps on the App Store [3], one can assume there are thousands which not only lack compatibility updates, but have been entirely abandoned by their creators, yet continue to be available for download.

In addition to providing answers to the research questions listed above, we wanted to prove that it is still possible to find value in a seemingly saturated market. While App Store monetization is becoming increasingly tougher for independent developers and small companies, curious and willing individuals can still explore a niche filled with subpar or lesser known apps and improve upon what already exists [4][5]. Furthermore, with the App Store market continuously shifting, a new app can fill an unexplored void at the time of its release (e.g., Instagram, Snapchat, Pokémon Go).

With our wish list management app, we aimed to do the following: find a niche, attempt to find value, and analyze the market. As this proved viable, we followed up by developing an app that complies with iOS's design guidelines, improves upon the feature set of the best-analyzed app, and can be considered a "good citizen" (i.e., no excessive battery consumption, advertisements, or data mining).

1.7 Structure

The rest of this dissertation is structured as follows:

[4]We give preference to open-source or freeware utilities to reduce costs.

- **Chapter 2** focuses on providing background information, a detailed exposure of the problem, and the proposed solution.

  It also includes answers to pertinent engineering questions, value analysis of our proposed app, and the state of the art, detailing the current hardware; technologies and frameworks for app development; an overview of the App Store, and an examination of three existing wish list management apps.

- **Chapter 3** discusses Apple's Human Interface Guidelines (HIG). It also analyzes ten popular apps that have been criticized by the media to find out common issues (according to a set of predefined metrics), details the most popular third-party frameworks and utilities, and presents detailed information about Swift and its maturity status.

- **Chapter 4** presents our app's design and requirements. The functional and non-functional requirements are defined, followed by the architectural model of the application (client-server). We also include the app's database model (and object graph), and detail the main design pattern used for iOS development: Model-view-controller (MVC).

- **Chapter 5** details how we store data, the workings of our chosen backend provider, and the implementation of use cases in our app from a logical standpoint.

- **Chapter 6** explains the various testing mechanisms in iOS development, from functional unit tests to performance (profiling) and user interface testing. We also describe how the beta-testing process will work and give insight on plans for deployment and marketing.

- **Chapter 7** closes out this dissertation with a review of completed requirements, limitations and future work remarks, and a final appreciation statement.

1.8 Summary

This introductory chapter covered a brief exposure to the context and issue at hand, as well as our two-part approach to solving it. To complement this approach, we defined a primary question and a series of derived subquestions that help guide the research and development processes. To follow up, we presented the objectives and motivation for this project. Lastly, we covered the document structure with a brief description of the contents of each chapter.

Chapter 2

Context, Value Analysis & State of the Art

This chapter opens with a thorough examination of the context and problem at hand. We present a proposed solution and place it under scrutiny by answering a series of engineering questions. Then, we analyze the potential value the practical aspect of the solution (an iOS application) can bring to users.

We follow up with an overview of the state of the art about current Apple technologies — hardware and software — which is important to understand what the platform offers users and developers.

2.1 Context & problem

In January 2007, Steve Jobs unveiled a product category that would change the computer industry forever. The iPhone, released in June 2007, was not the first mass-marketed smartphone; in fact, millions of smartphones ran Windows Mobile, Blackberry OS or Symbian years before Apple's announcement. However, the iPhone was the first phone to ship with a desktop-class operating system, enabling innovative technology to be the cornerstone of the mobile experience: Multi-Touch (multi-finger gestures) [6].

Just like previous interfaces which revolutionized user interaction (the mouse, and the iPod click wheel), Multi-Touch made natural gestures possible, such as panning and zooming. Coupled with a large touchscreen (at the time) and an adaptive virtual keyboard, the iPhone rapidly shaped the modern smartphone era — no styluses involved.

Notwithstanding, it was not until version 2.0 of iPhone OS (released July 2008) that Apple introduced the App Store, allowing third-party developers to publish apps. As of June 2016, the App Store is home to over two million iOS apps, making it the second largest app store behind Google's Play Store (2.2 million) [3][7].

When Apple first announced the iPad in January 2010, it complemented the current lineup of mobile devices by ushering in a fresh wave of apps designed for tablet use. The 9.7" tablet made developers rethink about their apps' designs, features, and user experience (UX) to take advantage of the extra screen size[1]. Four months later, it led Apple to rebrand "iPhone OS" as "iOS".

[1] Previously, all iPhone OS apps were optimized for 3.5" screens (the largest screen size available at the time).

The releases of the iPad mini (October 2012), larger-sized iPhones (September 2014), and iPad Pro (September 2015) brought new screen sizes developers were encouraged to adapt their apps to. Moreover, modern hardware such as Touch ID (fingerprint scanner) and 3D Touch (pressure-sensitive displays) popularized features which were not available in years prior.

Problem

Due to users' rapid adoption of the latest version of Apple's mobile operating system, the majority of iDevices (iPhone, iPad and iPod touch) are running iOS 9 — as of August 2016, its adoption rate was 88% [1]. This consistent year-by-year adoption was a contributing factor to the App Store maintaining the top spot in mobile app revenue in 2015[2].

The tendency to update iDevices to the most recent firmware pressures developers and users in keeping their apps up to date. Developer benefits for updating apps before launch day[3] include: getting familiar with new technologies and Application Programming Interfaces (APIs), improving the user experience, reaping press benefits, and potentially being featured by Apple [9]. Users who update apps benefit from new features, bug fixes, and performance improvements.

Despite the extensive list of advantages for developers, some apps (including those on the top charts) receive belated or minor updates even after the release of a major iOS version that carries fundamental changes (i.e., in design with iOS 7, or to APIs — iOS 8, 9, and 10).

The main drawback to apps not receiving updates to support the current iOS version is the eventual degradation of user experience (e.g., freezing, crashing, random bugs, slow performance), causing growing dissatisfaction and negative App Store reviews [10]. In some cases, these reviews can cause severe damage to developers' and companies' reputations. Consequently, this leads to a reduction in the number of downloads, retained users (i.e., who come back to an app), and a search for alternatives that focus on providing a superior user experience.

If we consider that the average app loses more than 75% of its active users after just one day, then it's not surprising to see a significant number of apps being released simply to make money [11]. After the initial surge drops due to users abandoning the app, the developers relegate it to the app wasteland and work on their next flash project. As a result, the mobile market is flooded by low-quality apps that will never see a Top 100 chart again (and probably had not in the first place). The App Store's front page is carefully curated by Apple editors, but that represents only a small fraction of the total available apps (over two million). Fortunately, Apple is now taking measures to remove low-quality apps from the App Store [12].

The top apps are not safe from issues, either. Companies behind these successful apps usually have other priorities than their iOS or Android clients. Thus, they end up either receiving belated updates to support new features (provided by the operating system), or none at all. Some companies fail to realize that usability and experience are more important

[2] According to a 2015 App Annie report, Google's Play Store had 100% more downloads, but Apple generated 75% more revenue [8].

[3] This is the day Apple releases a major version of iOS, typically around mid-September, coinciding with the release of the new iPhones.

than brand name alone. In a 2010 study on mobile UX, more than one-third (38%) of mobile app users were disappointed with apps from their favorite companies [13].

There are varying factors users consider essential, ranging from the UI and battery life to data protection mechanisms and privacy concerns [14]. Apps should be "good citizens": respecting Apple's HIG, conserving as much battery life as possible, and having clear, easy to understand Privacy Policies (amongst other criteria). Developers' failure to acknowledge these conditions may lead to their apps receiving bad reviews and negative press — harming both users and developers in the process (as well as Apple's reputation).

Proposed solution

As briefly presented in Chapter 1, this dissertation attempts to raise awareness to how developers can benefit from updating their apps to the latest version of iOS, taking advantage of recent technologies and frameworks. The first section of Chapter 3 examines a list of popular offenders (i.e., popular apps that have received negative press due to lacking features or malpractices) in multiple aspects — battery life, performance on older devices, conformity to Apple's HIG, and others.

The dissertation will also showcase the development of an iOS app from its initial idea up to the deployment and maintenance phase. It is important to demonstrate the practical feasibility of creating an app from scratch which follows Apple's guidelines and respects factors users seek.

2.1.1 Engineering questions

According to *The Thinker's Guide to Engineering Reasoning* [15], one should seek answers to some important questions before solving any problem, so a good transition to a solution can be achieved. Thus, the answer to thirty-five engineering reasoning questions can be found below. They challenge the engineering purpose, assumptions, available information, implications, and more. Questions not relevant to this project have been omitted.

Engineering purpose

1. **What is the purpose of the proposed solution?**

   The purpose is to showcase the various aspects of iOS development, namely frameworks and new technologies. It is important to understand how to code apps for this platform by following Apple's official guidelines, and avoid common pitfalls such as battery drain and not taking advantage of the full capabilities of each device, as well as respecting user privacy and ensuring security of the handled data.

2. **What are the market opportunities or mission requirements?**

   There are two main mission requirements. The first is detailing the new frameworks and technologies introduced with iOS 9 and 10, and the Swift programming language (a fundamental building block of modern iOS development). The second requirement is the creation of an iOS application that takes advantage of new technology and frameworks, respecting the concerns listed in question 1.

3. **Who defines market opportunities/mission requirements?**

   In this case, the proponent (author) of this dissertation. However, market opportunities should be analyzed by conducting a market study prior to the development of the application, to assess a potential niche.

4. **Who is the customer?**

   The customer (or customers, in this sense) is the direct recipient of the application to be deployed on the App Store — iOS users on version 9.0 or higher.

Question at hand

5. **What system will best satisfy the customer's requirements?**

   The proposed solution will limit, by default, the client's requirements to any device capable of running iOS 9.0 or higher (non-functional requirement).

   A detailed list of requirements can be found in Chapter 4.

6. **How does the customer define "value"?**

   "Value" in this context can be defined as what iOS customers appreciate in an app. In general, they value stability, ease of use (simplicity), universality/responsive design (i.e., one app can run seamlessly on both iPhone and iPad, taking advantage of various screen sizes) and battery life. Thus, the application to be developed needs to adhere to these core requirements.

7. **Can an existing design be adapted?**

   If this was an existing app, it could (depending on the design and whether the app is reasonably updated for iOS 7+ design guidelines). However, since the solution contemplates starting from scratch, this is not applicable.

8. **How important is time-to-market?**

   Not very important, due to this being a project that is being developed by a single individual and for academic purposes. That being said, it needs to fit in a relevant timeframe, as it will also be an experience of the number of months necessary to develop an application for this platform from scratch that follows major engineering development stages.

Point of view

9. **A design and manufacturing point of view is typically presumed. What other points of view deserve consideration? Users? Regulators? Others?**

   Users deserve consideration, so it is necessary to contemplate what they will value (see question 6). The regulator, in this case, refers to Apple. Adherence to App Store policies is necessary, or the app will be rejected.

Assumptions

10. **What environmental or operating conditions are assumed?**

    Optimal, yet realistic working conditions are considered (e.g., modern and also legacy hardware with the latest version of iOS 9 for development and testing purposes).

11. **What programmatic, financial, market or technical risks have been considered acceptable to date?**

    There are not many programmatic and financial risks, the former because the programming language (Swift) is backward-compatible with Objective-C and the latter due to no required investment.

    Regarding the market risks, there is always a possibility of developing an application and it not succeeding as expected. This is an inherent risk of any market, including the App Store. As more apps continue to flood the market, it is increasingly difficult to stand out, barring being featured by Apple (i.e., having a app banner on the main App Store page).

    As for technical ones, we must consider that development overlapped the announcement of new features and technologies for iOS 10, at WWDC 2016. That said, the new features did not affect the app's development too heavily.

12. **What market/economic/competitive environment is assumed?**

    The App Store is a very competitive market, with some high entry barriers now that it has reached maturity. However, if a niche is exploited, it could pay off.

13. **What maturity level or maturation timeline is assumed for emerging technologies?**

    It is assumed that the Swift programming language, now in its third version, has not yet reached the phase of complete maturity, but it is adequate in complementing Objective-C as a primary way of developing Swift apps. This is because most third-party libraries — analytics, logging, crash debugging, utilities, etc. — are still coded in Objective-C, although they can be easily integrated and work alongside Swift.

14. **What happens if we change or discard an assumption?**

    This depends on the assumption and whether it has ties to other deeper implications. Fortunately, most of the assumptions made are reasonable and realistic, given the current state of the development technologies and the App Store as a content platform, so any slight variance in assumptions should not have a profound impact on the end product.

15. **What assumptions have been made on the availability of data/information?**

    Information is assumed to be present and available, as it is hosted online and there are many resources for learning about iOS development using current technologies, both free and paid. There are also support platforms that can be used during development (e.g., Stack Overflow and Hackhands).

Engineering information

16. **What is the source of supporting information (handbook, archival literature, government regulation, etc.)?**

    The primary sources are found online, such as a multitude of websites that detail the new technologies and Swift 2's documentation, both official and unofficial. Tutorials, videos, and developer API references are also freely available.

17. **What information is lacking?**

    We currently lack information on average App Store revenue by category, which categories generally produce the most engaging user bases and, most importantly, which model is best suited for new apps at the time of deployment (i.e., free, free w/ in-app purchases, or paid).

18. **How can we get it? What experiments should be conducted?**

    We can obtain a significant amount of this information by conducting a market study and researching the industry leaders in information measurement (e.g., Nielsen), as well as reading opinions and experiences by prominent self-employed developers of this platform.

19. **Have we considered all relevant sources?**

    For the most part, we have, although more sources may gain relevance during each phase of this project.

20. **What legacy solutions, shortcomings, or problems should be studied and evaluated?**

    It is important to take into account that Objective-C still dominates a vast majority of iOS apps, as well as most third-party tools and frameworks that provide insight and useful functionality. Although this language is not yet deemed legacy, in a few years (3–5), it is expected that Swift will hold a majority of the market share.

21. **Is the available information sufficient? Do we need more data? What is the best way to collect it?**

    We do need more data. The best way to collect it is with a market study, referred above (question 19).

Concepts

22. **What concepts or theories apply to this problem? Are there competing models?**

    This question is not very relevant to this project because the app to be developed does not directly stem from a market necessity. Its goals are showcasing modern technologies paired with a modern programming language, and exemplifying app development on this mobile platform.

23. **What emerging theory might provide insight?**

    Not relevant to this project, as explained above.

24. **What available technologies or theories are appropriate?**

    As far as technologies go, taking advantage of the new hardware of the iPhone 6/6S and 3D Touch technology to enable pressure-sensitive actions, as well as new software features like Size Classes, will allow for responsive design (i.e., that adapts to different screen resolutions). There are many more we can use, such as, but not limited to:

    - Touch ID authentication
    - App Thinning[4]
    - On Demand-Resources[5]
    - App Transport Security (ATS)[6]

25. **What are some important implications of gathered data/information?**

    One important implication is that since there are many resources on the web that provide high-quality information, one would expect most developers to follow the Apple's design guidelines and best practices. However, this is not always the case. There are apps that don't conform to design guidelines and end up feeling out of place, or worse — they haven't been carefully profiled to achieve optimal battery life.

Implications

26. **What are the most important market implications of the technology?**

    Swift 3 and the new technologies/frameworks introduced with iOS 10 make it easier for developers to create new or enhance existing iOS applications, by means of a robust, type-safe, and easier-to-use language than Objective-C. This lowers the entry barrier for others that are interested in coding for iOS and have prior experience with other mobile platform languages[7]. Thus, it should result in apps that are taking advantage of a big part of what is new, while optimizing for battery and taking into account security, privacy, and responsive design requirements.

27. **What are the most important implications of a key technology not maturing on time?**

    It could be quite the predicament when it comes to Swift, as Apple has invested considerable efforts into making it the primary programming language on their platforms going forward.

    The good thing about it is that Apple keeps a firm foothold on its platform, and controls every part of the ecosystem. Therefore, it is striving for mass adoption of its new programming language. One major step forward was making Swift 2 open-source. This move means that Swift will be able to be used for more than just coding specific apps for two platforms (OS X and iOS).

[4] This concerns delivering device-dependent binaries to reduce app size. For example, the iPhone 6 would only receive 64-bit slice, high Central Processing Unit (CPU), high Graphics Processing Unit (GPU); iPhone 4S would only get 32-bit slice, low CPU, low CPU.

[5] Loading content only when needed to save data, battery and flash storage; data is deleted after usage.

[6] Ensures that all communications the app makes are secured via HyperText Transport Protocol Secure (HTTPS).

[7] For example, Swift's syntax is much similar to Java (a widely-used programming language) than Objective-C.

28. **How important is after-market sustainability?**

    It's not essential, but it would be nice to see reasonable adoption (a few hundred users in the first month of release, let's say).

29. **Is there a path for future design evolution and upgrade?**

    Yes, there is. A roadmap will be created to aid future development and avoid suffering from "featuritis" (i.e., too many features, causing bloat). This allows for a somewhat regular cadence between features and bug fixes. It is crucial to take into account tester (via TestFlight) and customer feedback (via App Store reviews) to prioritize what needs to be implemented or fixed.

30. **Are there disposal/end-of-service-life issues we need to consider?**

    Not relevant for this project, because the end product is non-tangible. If the app stops receiving regular updates and is no longer supported down the road, it will be removed from the App Store if the host developer fails to renew their annual Apple Developer Program membership.

31. **What are the most important implications of product failure?**

    It is tough to gauge how the application will perform adoption-wise once it hits the market. Even if it fails to attract and capture users, it would still be a learning experience for future projects. Hypothetically, the most important thing would be to learn from what went wrong and the causes that led to product failure.

32. **What design features if changed, profoundly affect other design features?**

    As mentioned in question 11, a major redesign would change the design characteristics of the app quite profoundly, as they would have to be rethought and possibly reimplemented to follow the new guidelines.

33. **What design features are insensitive to other changes?**

    Most of the application's UI flow would remain untouched; however, the visual cues would have to be adapted.

34. **What potential benefits do by-products offer?**

    Not relevant until a market study is conducted to provide data for comparison between the segment leaders and what can be developed to improve upon existing solutions and include relevant features others haven't implemented yet.

35. **Should social reaction and change management issues be addressed?**

    Yes, or at the very least they should be considered, because feedback is valuable to small developers. Bad reviews can quickly discredit them; thus, it is important to deal with the most common issues with a proactive stance (i.e., fixing a particular bug in an earlier stage of the release cycle with the help of beta testers).

2.1.2 Existing restrictions

There are no technological restrictions apart from the main development language. Swift was chosen because it is Apple's primary focus as a way to develop iOS apps. While some Objective-C components are used in this project, they simply support the app's functionality.

It is relevant to note not all developers have temporal budgets as large as this project (approximately six months). That said, the duration can scale with the number of team elements. In this case, the app will be developed and maintained by a single developer, which explains the lengthy amount of time.

As far as financial restrictions are concerned, this is beyond the scope of this dissertation, because costs vary in a professional environment. The only necessary expenditure is a €99/year fee to pay for the annual Apple Developer membership. All other expenses (i.e., marketing costs, design outsourcing, and others) are optional and should be considered on a case-by-case basis.

2.2 Value analysis

This value analysis focuses on the value that the iOS application can bring to its users, **not** the theoretical overview and analysis done on frameworks, technologies, and popular apps in Chapter 3.

2.2.1 Overview

A value proposition is a "business or marketing statement that summarizes why a consumer should buy a product or use a service" [16].

Planning and creating a value proposition is a fundamental part of business strategy because it guides a company's business model. It allows a company to effectively target each customer segment and create value. According to Kaplan and Norton [17], satisfying customers creates sustainable value to the organization, and strategies vary by customer segments.

A company's value proposition consists of a series of value-based drivers (e.g., quality, reliability, customization) it sees in its products or services that can benefit the target customer segment. The worth that these factors have on consumers' minds is designated perceived value, which directly affects the price they are willing to pay for the good or service[8].

A good proposition focuses on delivering value to all interested parties: prospective and existing customers, and other groups in and out of the organization who are part of a specific customer segment. A value proposition must also analyze perceived and potential costs and benefits; therefore, it's considered a positioning of value, determined by the following formula:

$$V(value) = B(benefits) - C(costs)$$

2.2.2 Value proposition

Creating a value proposition for the practical aspect (i.e., the app) of this dissertation starts with defining an idea. Customers will quickly ask: *What does this app allow me to do?*. Therefore, we must think of its core functionality that can be used to create value.

[8] Different customers perceive distinct value for identical products/services. Also, organizations involved in the purchasing process may have different perceptions of customers' value delivery [18].

In this case, we have chosen to develop a wish list management app. We can define our value proposition as the following:

This wish list management app provides you [e.g., shopping enthusiasts, organized individuals, social wish list lovers] with an easy and fun way to take photos of items you desire, create wish lists, and share them with friends, family, and colleagues.

The above sentence provides answers to three fundamental questions every value proposition must feature [19]:

- **What is the product or service?**

  It is a wish list management app.

- **Who is the target customer?**

  Shopping enthusiasts, organized individuals, and social wish list lovers are the target customers[9].

- **What value does the product or service provide?**

  1. It saves time because users can quickly take a photo of something they like and have items organized in one place.
  2. It is a modern, easy-to-use app that takes full advantage of new iOS hardware and software features.
  3. It allows people to be more social by providing a means to share what they would like to have.

Value temporal position

Woodall [20] defined a longitudinal perspective on value for the customer (VC) based on four distinct temporal positions. In regards to our wish list management app, we can identify the following benefits and sacrifices in each position (table 2.1).

Table 2.1: Longitudinal perspective on view controller (VC) for a wish list management app

	Ex Ante-VC	**Transaction VC**	**Ex Post-VC**	**Disposition VC**
Benefits	Visibility Product characteristics	Product characteristics Functional benefits Features Support Quality (reviews)	Features Reliability Performance Enjoyment Convenience Social aspects	Support Reliability Enjoyment Convenience Social aspects
Sacrifices	Marketing expenses Effort Time	Price Deployment costs	Human energy Effort Time	Human energy Support costs Effort Time

[9] Although we tend to have target customers who are more likely to download and use this app, it is available to anyone. Notwithstanding, our value proposition is built around them.

2.2.3 Negotiation scenarios

[21] defined seven different types of negotiations and recommended approaching each with distinctive strategies. While some are beyond the scope of this dissertation's value analysis, it is important to highlight the three most typical negotiations:

- **Distributive negotiation (win-lose)**: these result in both parties trying to obtain maximum gain or minimal loss. Only one party can win in this type of negotiation.

  Since the app will be free with in-app purchases (IAP) or fixed price, there are no distributive negotiations in place. The customer will either willingly pay for the app/IAP, or seek an alternative. Either way, no contact was established with the (potential) customer.

- **Integrative negotiation (win-win)**: here, both parties collaborate to benefit mutually.

  A suitable example to our app would be beta-testers. These users voluntarily test and give feedback prior and after the app's deployment. As a reward, the app would be free for them (assuming a paid app). Both parties win because we receive valuable feedback to improve the app (and loyal customers); they keep using the app at no cost.

- **Lose-lose negotiation**: this is the worst kind of negotiation, where both parties come out negatively impacted.

  Translating this negotiation type to our app, if a customer emails us for support and we cannot solve their issue due to varying factors (e.g., bug impossible to reproduce, lack of information), they might become annoyed with us and request a refund from Apple. Consequently, we lose a customer, the sum paid by the customer, and they won't use our app again.

2.2.4 Business Model Canvas

The Business Model Canvas describes the nine basic building blocks that make up an organization's business model [22]. The Canvas for our app's business model can be found in Appendix A. Here is an explanation of each block:

- **Customer Segments**: these are the people we are providing value to — they are customers we want to target because of their increased probability of them downloading our app. As described above in Section 2.2.2, we defined **three customer segments**[10].
- **Value Propositions**: in this case, for each customer segment we offer similar value propositions (see Section 2.2.2 for details).
- **Channels**: the home to all iOS apps is **Apple's App Store**, so this is the only channel with our customers.
- **Customer Relationships**: this concerns the ways we build relationships with our users: **support** via email, social networks (e.g., Twitter, Facebook) and **customer feedback** that aids in improving the app's experience (i.e., App Store reviews).

[10] It is entirely possible that all these groups may apply to a single individual, as we offer them similar benefits (i.e., generally people who love to make wish lists also enjoy shopping, and sharing them).

- **Revenue Streams**: "show how and through which pricing mechanisms our business is creating value". If the app has an **upfront cost**, that is the primary revenue stream; otherwise, **in-app purchases** (e.g., to remove ads) will make up the largest portion of revenue. We can also monetize the app by **showing ads** for free users (if the app is free to download and use).
- **Key Resources**: "the infrastructure to create, deliver and capture value": the **technological infrastructure** of the app, which is created by an iOS developer.
- **Key Activities**: what needs to be done for the business to perform correctly. In our case, **developing and maintaining the app**, as well as **providing customer support**.
- **Key Partners**: "shows who can leverage the business model". **Apple** is our primary key partner, because we depend on them for app hosting & exposure. If we choose to show ads to free users, **Google** will provide them via their Adsense network. Another very important partner is the **back-end service provider** we choose to host our back-end stack[11]. Without this key partner, the app would not work as intended.
- **Cost Structure**: the obligatory cost here is the **Apple Developer membership** to deploy apps to the App Store (€99/year). There are other optional and varying costs, such as **marketing expenses** which can boost the exposure of our app.

2.2.5 Analytic Hierarchy Process

We can use the Analytic Hierarchy Process to make a structured decision based on criteria and possible alternatives. In this case, we want to figure out which wish list application best suits our needs from the three analyzed in Section 2.3.4: WishMindr, Gifster, or Giftry.

AHP models can be created by first defining an objective or goal. After this, consider a series of quantifiable criteria that will be the basis for pairwise decisions. We chose *User Interface* (I), *Features* (F), and *Customization* (C) as the three criteria to judge these apps. A visual model is shown in.

The *CP* matrix shows the criteria preference and weights in pairwise comparisons. "Comparing (...) objective i and objective j (where i is assumed to be at least as important as j), give a value a_{ij} as follows" [23]:

Table 2.2: Pairwise comparison values

1	Objectives i and j are of equal importance
3	Objective i is weakly more important than j
5	Objective i is strongly more important than j
7	Objective i is very strongly more important than j
9	Objective i is absolutely more important than j
2,4,6,8	Intermediate values

We consider *Features* the most important criteria, being 6 times more important than *Customization* and twice as important as the *UI*, which is 3 times more important than *Customization*.

[11] *Stack* in this context refers to the various layers of our app (e.g., front-end stack, back-end stack). The back-end stack, connected to its provider, deals with user authentication, and cloud data syncing and storage.

$$CP = \begin{array}{c} \\ I \\ F \\ C \end{array} \begin{array}{c} \begin{array}{ccc} I & F & C \end{array} \\ \begin{bmatrix} 1 & 1/2 & 3 \\ 2 & 1 & 6 \\ 1/3 & 1/6 & 1 \end{bmatrix} \end{array}$$

Figure 2.1: Criteria preference matrix

With our criteria weights defined, we now apply the same logic to each app by criteria (for example, we consider Gifster to have a UI 4 times better than WishMindr, but $1/4$ as good as Giftry). Each of the three pairwise matrices is shown below.

$$I = \begin{array}{c} \\ WishMindr \\ Giftster \\ Giftry \end{array} \begin{array}{c} \begin{array}{ccc} WishMindr & Giftster & Giftry \end{array} \\ \begin{bmatrix} 1 & 1/4 & 1/6 \\ 4 & 1 & 1/4 \\ 6 & 4 & 1 \end{bmatrix} \end{array}$$

Figure 2.2: User Interface pairwise matrix

$$F = \begin{array}{c} \\ WishMindr \\ Giftster \\ Giftry \end{array} \begin{array}{c} \begin{array}{ccc} WishMindr & Giftster & Giftry \end{array} \\ \begin{bmatrix} 1 & 1/4 & 1/5 \\ 4 & 1 & 1/2 \\ 5 & 2 & 1 \end{bmatrix} \end{array}$$

Figure 2.3: Features pairwise matrix

$$C = \begin{array}{c} \\ WishMindr \\ Giftster \\ Giftry \end{array} \begin{array}{c} \begin{array}{ccc} WishMindr & Giftster & Giftry \end{array} \\ \begin{bmatrix} 1 & 1/7 & 1/4 \\ 7 & 1 & 3 \\ 4 & 1/3 & 1 \end{bmatrix} \end{array}$$

Figure 2.4: Customization pairwise matrix

Next, we have to obtain the normalized principal Eigen vector (also called priority vector) for each criteria. This is done by adding each column of a pairwise matrix. Then, we divide each column of the matrix with the sum of its column, giving a normalized relative weight (i.e., the sum of each column is 1).

The normalized principal Eigen vectors for each criteria (column) are grouped in the priority matrix (figure 2.5).

$$PM = \begin{array}{c} \\ WishMindr \\ Giftster \\ Giftry \end{array} \begin{array}{c} \begin{array}{ccc} I & F & C \end{array} \\ \begin{bmatrix} 0.0819 & 0.0974 & 0.0786 \\ 0.2363 & 0.3331 & 0.6586 \\ 0.6817 & 0.5695 & 0.2628 \end{bmatrix} \end{array}$$

Figure 2.5: Priority matrix

We can now multiply the priority matrix above by the criteria weights (0.3, 0.6 and 0.1, respectively), which will give us the ranking of alternatives:

Thus, we can now assert **Giftry** would be the most appropriate app to download based on our criteria and weights, with just over 57% of the ranking.

$$RA = \begin{array}{c} \\ WishMindr \\ Giftster \\ Giftry \end{array} \begin{array}{c} \begin{array}{cccc} I & F & C & Result \end{array} \\ \begin{bmatrix} 0.0246 & 0.0584 & 0.0079 & \mathbf{0.0909} \\ 0.0709 & 0.1999 & 0.00659 & \mathbf{0.3366} \\ 0.2045 & 0.3417 & 0.0263 & \mathbf{0.5725} \end{bmatrix} \end{array}$$

Figure 2.6: Ranking of alternatives

To make sure how consistent the judgments have been relative to large samples of purely random judgments, we must calculate the consistency ratio (CR). This value should be less or equal to 0.1; else, the exercise should be repeated as it is deemed untrustworthy.

To calculate CR, we have to first calculate λ_{max} to also give us the consistency index (CI). Consider $Ax = \lambda_{max}x$ where x is the Eigen vector:

$$\begin{array}{cccc} A & & x & Ax & \lambda_{max} \end{array}$$
$$\begin{bmatrix} 1 & 1/2 & 3 \\ 2 & 1 & 6 \\ 1/3 & 1/6 & 1 \end{bmatrix} \begin{bmatrix} 0.3 \\ 0.6 \\ 0.1 \end{bmatrix} = \begin{bmatrix} 9/10 \\ 9/5 \\ 3/10 \end{bmatrix} = \begin{bmatrix} 3/10 \\ 6/10 \\ 1/10 \end{bmatrix}$$

Figure 2.7: Calculating λ_{max} vector

We need to average the values to give us λ_{max}, as such:

$$\lambda_{max} = mean\left\{\frac{0.9}{0.3}, \frac{1.8}{0.6}, \frac{0.30}{0.10}\right\} = 3$$

The CI is given by:

$$CI = \frac{(\lambda_{max} - n)}{(n-1)} = \frac{(3-3)}{(3-1)} = 0$$

As $CI = 0$, we can assert that $CR = 0/0.58 < 0.10$, so our calculations were trustworthy[12].

2.3 State of the Art

This section describes the state of the art for both the theoretical analysis and the development of the app, from a bottom-up perspective (i.e., hardware to software layers).

2.3.1 Hardware

Apple's mobile hardware has evolved significantly since the introduction of the first iPhone (2007). The latest generation — the iPhone 6s and 6s Plus — were considered the fastest phones in 2015 by multiple benchmarks [24, 25].

Apple's hardware evolution also extends to its other mobile lineups: the iPad and iPod touch. Regarding the iPad, the iPad Air 2 (the latest 9.7" model) boasts up to 40% faster CPU performance than its predecessor (iPad Air) and up to 2.5 times in graphics performance [26]. When it comes to the iPad Pro — Apple's biggest tablet offering — the new 64-bit A9X

[12] This value may suffer small variances due to the infinite fractions in our A vector (figure 2.7). However, the final value of CR will be very close to 0 even when considering a large amount of decimal places.

chip is Apple's fastest on any mobile device, with 1.8 times the CPU performance of the iPad Air 2, and up to double in graphics performance (despite having a display with nearly twice the pixels) [27]. The iPod touch (in its sixth generation) also received an update in 2015, now with an A8 chip (the same that powers the iPhone 6 and 6 Plus) and 1 GB RAM (it previously had an A5 and 512 MB RAM), a much-needed upgrade from its predecessor.

A total of twenty Apple devices spanning seven different screen sizes are capable of running iOS 9, ordered below:

- **iPhone/iPod touch**
  - 3.5" devices: iPhone 4S
  - 4" devices: iPhone 5, iPhone 5c, iPhone 5s, iPod touch 5th generation, iPod touch 6th generation
  - 4.7" devices: iPhone 6, iPhone 6s
  - 5.5" devices: iPhone 6 Plus, iPhone 6s Plus
- **iPad**
  - 7.9" devices: iPad Mini, iPad Mini 2, iPad Mini 3, iPad Mini 4
  - 9.7" devices: iPad 2, iPad 3rd generation, iPad 4th generation, iPad Air, iPad Air 2
  - 12.9" devices: iPad Pro

There is also a variety of new hardware technology available on the latest iPad and iPhone models, such as Touch ID, 3D Touch (iPhone 6s and 6s Plus only), and pressure-sensitive displays adapted to the Apple Pencil (iPad Pro only).

Touch ID

Originally on the iPhone 5s, Touch ID is a fingerprint recognition feature now available on more than half of available devices. There are currently two generations of the technology; the second generation Touch ID sensor is only currently available on the latest iPhone models and is twice as fast as its predecessor.

Built into the home button made of laser-cut sapphire crystal, Touch ID features three other components: a stainless steel detection ring, the sensor itself, and a tactile switch which acts as the traditional "home button" (figure 2.8) [28].

Touch ID can be used not only for unlocking the device but also for authentication in apps that support it. For example, the App Store allows purchasing with only a fingerprint as the confirmation method[13], as shown in figure 2.9 [29].

[13] In both cases, a passcode (or password) is still needed after a reboot, however. This method ensures the user is not subject to unwarranted access, as it is much harder to force someone to give up their passcode than their fingerprint.

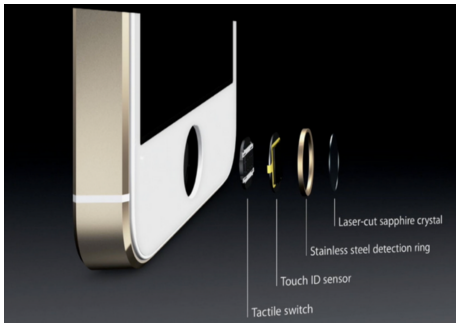


Figure 2.8: Components of Touch ID

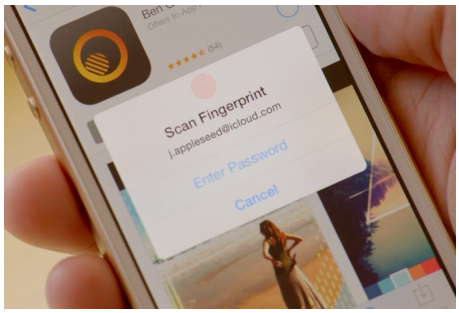


Figure 2.9: Authorizing App Store purchases

3D Touch

3D Touch (not to be confused with Force Touch[14]) is a pressure-sensitive technology built into the display of the iPhone 6s, iPhone 6s Plus, iPhone 7 and iPhone 7 Plus. It uses capacitative sensors along with the accelerometer to detect varying degrees of pressure, making it able to distinguish between normal and more forceful touches [30].

This technology allows the user to execute "Peek" and "Pop" gestures. "Peek" requires a little amount of pressure and displays bite-sized information that otherwise would cause a new view to show up — the contents of an email, the status of a flight, or a quick way to display relevant information (e.g., calling a business after tapping them inside the Maps app). Pressing harder after the initial peek pops the view onto the screen, expanding the available information. For example, after peeking into the latest messages of a conversation, popping it would show the whole thread (figure 2.10) [31].

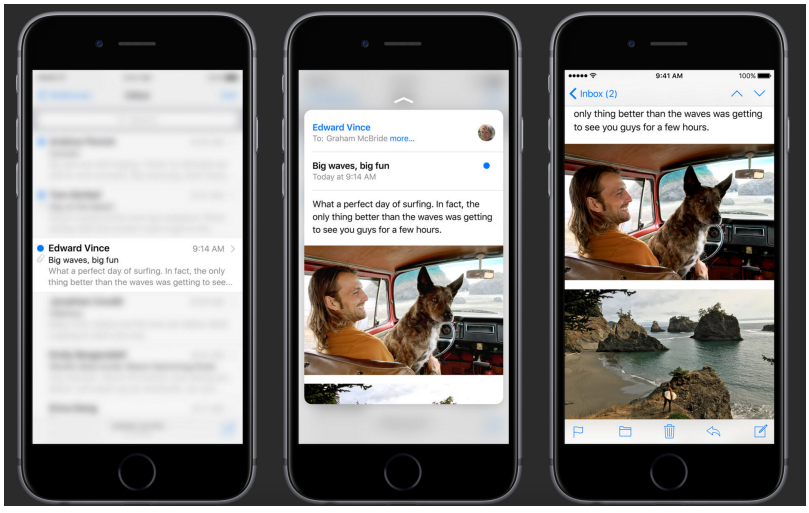


Figure 2.10: 3D Touch showing "Peek" and "Pop" actions for previewing emails

[14]Force Touch is a pressure-sensitive technology for trackpads (Macbook and Macbook Pro) and touch screens (Apple Watch and iPhone). The iPhone 6s and iPhone 6s Plus use a more precise version of Force Touch named 3D Touch.

When the user presses on icons on the home screen, apps compatible with 3D Touch show up to four "Quick Gestures", which act as shortcuts to actions (e.g., the Camera app has shortcuts that allow taking selfies and recording video quickly — figure 2.11) [31].

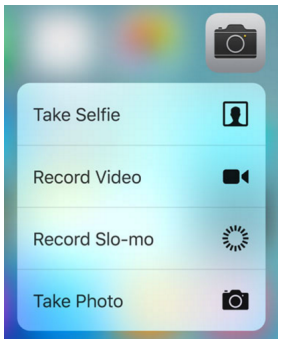


Figure 2.11: "Quick Gestures" of the Camera app

2.3.2 Technologies

iOS 9

iOS 9 is the ninth major release of iOS. Released to the public on September 16th, 2015, it currently holds the second largest percentage of market share [1]. Its adoption was the quickest of all iOS versions, with Apple reporting half its user base had upgraded after just one week of release [32].

While iOS 9 focuses more on performance optimizations than new features, it has significant changes, especially for iPad. Here is a brief list:

- **Proactivity**

  Now more aware of context, Siri will offer app suggestions depending on the time of day, and can read on-screen content during Siri prompts (e.g., "Who sings *this* song?"). Spotlight search is now also accessible by swiping left on the first home screen. This view shows recent contacts and used apps, as well as news and nearby locations.

- **Multitasking** (iPad only)

  One of the major features of iOS 9, newer iPad owners (iPad Air 2, iPad mini 4, iPad Pro) can take advantage of slide over and split screen multitasking, and picture-in-picture for watching videos. Slide Over pulls up a second app that occupies a third of the display with which the user can not interact. Extending this app will have it occupy half the screen and become fully interactive (Split View).

- **News**

  A new app that displays content from major news outlets. Users can choose their subscriptions and save articles for later reading. Currently only available in the US, UK, and Australia.

- **Notes**

  The Notes app was revamped, with the ability to draw sketches, add images and maps. More formatting options (e.g., bulleted and numbered lists) are also available.

- **Maps**

  Support for transit directions in major cities and recommendations depending on the user's location, interests and time of day.

- **Battery, performance and security improvements**

  Apple claims iOS 9 delivers up to one more hour of battery life compared to iOS 8 with normal usage. Activating the new Low Power mode extends battery life by an additional three hours.

  Performance has increased by adopting the Metal API. Users now have the option to choose a six digit passcode (as opposed to the standard four digit one). Apple now supports two-factor authentication for iCloud Drive.

- **Support for 3D Touch** (iPhone 6s and iPhone 6s Plus only)

iOS 10

iOS 10 is the current iteration of Apple's mobile operating system, released on September 13, 2016 and holds the largest amount of market share [1]. With this version, Apple re-designed and added features to many system apps (e.g., Maps, Messages, Music), made Siri compatible with third-party services such as ride booking and making payments (outside of Apple Pay), and 3D Touch can now be used system-wide [33].

It also sports a new lock screen with features like raise to wake, a Today view (where apps can show relevant information to the user via widgets) and rich notifications which allow the user to perform additional actions using 3D Touch.

iOS 10 drops support for all devices with the A5 chip, namely the iPhone 4S, the fifth-generation iPod touch, the iPad 2, the third-generation iPad, and the first-generation iPad mini. 19 devices support iOS 10 (including the new iPhone 7 and iPhone 7 Plus), illustrated by figure 2.12 below [33].

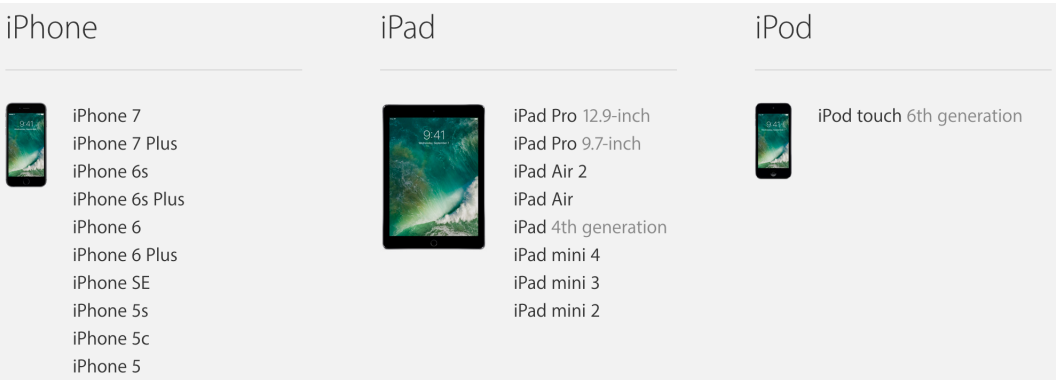


Figure 2.12: Devices compatible with iOS 10

Here is a quick rundown of the new changes and improvements made in this version:

- **Messages**

  The Messages app received the largest overhaul in iOS 10. Rich content such as links, videos, and other web content can be viewed without leaving the message thread.

Messages can now be sent with various bubble effects, adding vibrancy to otherwise static text. There is also support for image sketching similar to what is currently possible with Apple Watch.

Apple also introduced a Messages Software Development Kit (SDK) which will allow developers and designers to create various add-ons for the app. With its own dedicated App Store, users will be able to enjoy sticker packs which can be overlaid on existing messages, allowing for a wide range of customization.

- **Siri (digital assistant)**

  Significant improvements have been made to Siri, Apple's digital assistant present in iOS. With an open SDK, third-party developers can integrate their services to make it easy for users to book a ride, make payments, send messages via other apps (e.g., Whatsapp, Skype, Telegram), etc.

- **Maps**

  Maps has received another update in iOS 10, including deep learning by getting to know a user's habits, scanning calendar events for locations, and a new view while driving. The app can now alert a user to the location where their car was last parked (also displayed on the map with a marker).

- **Photos**

  Photos also benefits from deep learning. Users can now search photos with general keywords (i.e., searching for "beach" will display all user photos that match a beach setting). A new Memories feature compiles relevant photos into a short video which can be shared on social media.

- **Music & News**

  Both apps have received major overhauls, sporting thicker fonts and less visual density. With the Music app, Apple have also included lyric support for Apple Music subscribers.

- **Lock screen**

  The lock screen is the largest system redesign. The famous "slide to unlock" phrase has been replaced with "Press home to unlock". This change means that tapping the home button will now unlock the device (after unlocking with a passcode or Touch ID). Swiping to the left will reveal "Today widgets", which were only present in the Notification Center in previous versions of iOS. Swiping to the right will now reveal the camera interface (instead of swiping up in iOS 9 and below).

- **Other changes & improvements**

  With iOS 10, Apple is now allowing users to hide some default apps resulting in less home screen visual clutter. Hidden apps can easily be re-downloaded on the App Store.

Objective-C & Swift

Originally developed by NeXT for its NeXTSTEP operating system in the early 1980s, Objective-C has been the object-oriented programming language to develop iOS (and OS X apps) since the inception of the platform.

At WWDC 2014, Apple unveiled Swift: a multi-paradigm, static-typed programming language created for iOS, OS X, Apple Watch and tvOS development [34].

Built upon the frameworks that powered Objective-C (Cocoa and Cocoa Touch), Swift is interoperable (i.e., compatible) with Objective-C, allowing developing to maintain older codebases alongside Swift implementations. Swift was designed to be safer to errors (e.g., null pointers) than Objective-C, and also more concise.

In December 2015, Apple made Swift open-source, as well as launching resources to help developers contribute to the language [35]. Swift.org is the home for the open-source Swift community, and official Swift repositories are located on Github [36, 37].

Chapter 3 discusses Swift, including a brief overview of recent changes to Swift 3 (the latest version), benefits and drawbacks when programming in Swift, and its future roadmap.

2.3.3 Frameworks

UIKit & Foundation

The UIKit framework is part of the official iOS SDK. It handles events, views, view lifecycle, interface controls, and more.

Core Foundation is an Objective-C framework that provides wrapper and data structure classes. All classes are prefixed 'NS', referencing its NeXTSTEP origins (e.g., NSString for strings). This prefix is the base class for any object (NSObject). In Swift, "Core Foundation types are automatically imported as full-fledged Swift classes" (e.g., Array, String) [38].

Both frameworks are fundamental in iOS development. While Apple's iOS SDK has a ton of other frameworks and APIs, these two are present in almost every app.

Third-party frameworks

Below is a brief list of some of the most popular third-party frameworks used for iOS app development[15]. An in-depth analysis of the frameworks used to develop our wish list management app is provided in Chapter 3.

- **Beta distribution**
  - **Crashlytics**: a powerful cross-platform crash reporting and beta testing service.
  - **HockeyApp**: allows developers to distribute beta versions of their apps, collect live crash reports, get feedback from users, and analyze test coverage.
  - **TestFlight**: Apple's beta testing service hosted on iTunes Connect.
- **Databases**
  - **Realm**: a replacement for Core Data and SQLite: simple, modern and fast.
  - **FCModel**: an alternative to Core Data that offers direct Structured Query Language (SQL) access.

[15] A comprehensive, curated list of iOS frameworks, libraries and much more is available on Github [39]. These include both open-source projects and free and paid services.

  - **SwiftyDB**: a wrapper around SQLite databases written in Swift

- **Dependency/Package Manager**
  - **CocoaPods**: the most widely used dependency manager for iOS. It has over ten thousand libraries and helps scale projects elegantly.
  - **Carthage**: a simpler, decentralized alternative to CocoaPods.

- **Logging**
  - **CocoaLumberjack**: a fast and simple, yet powerful and flexible logging framework for iOS.
  - **CleanroomLogger**: a configurable and extensible Swift-based logging API that is simple and lightweight.
  - **NSLogger**: a high-performance, cross-platform (Mac OS X, iOS and Android) logging utility which displays traces emitted by client applications.

- **Networking**
  - **AFNetworking**: the most popular open-source iOS and OS X networking framework. It has a modular architecture with feature-rich APIs.
  - **RestKit**: an Objective-C framework for iOS that aims to make interacting with Representational State Transfer (REST) web services simple, fast and fun.
  - **Alamofire**: a HyperText Transport Protocol (HTTP) networking library written in Swift, from the creator of AFNetworking.
  - **Starscream**: a conforming WebSocket (RFC 6455 [40]) client library in Swift for iOS and OSX.

- **Testing**
  - **Kiwi**: a behavior-driven development (BDD) library for iOS development.
  - **Quick**: Quick is a BDD framework for Swift and Objective-C.
  - **Nimble**: a matcher framework for Swift and Objective-C.

2.3.4 App Store

Overview

As mentioned in Section 2.1, Apple's App Store generated the most mobile app revenue in 2015. A year prior, Evans [41] crunched some numbers to explain this trend:

> *Apple told us that it paid out $7bn in calendar year 2013 — given the growth trend, it probably paid $10bn in the last 12m. On a trailing 24m basis, there were 470m iOS users in March 2014.*
>
> *So, Google Android users in total are spending around half as much on apps on more than twice the user base, and hence app ARPU*[16] *[App Revenue Per User] on Android is roughly a quarter of iOS.*

[16] Defined as the total revenue divided by the number of users.

This is consistent with App Annie's 2015 report that states Apple generates 75% more revenue [8]. Evans offers a few key suggestions to this disparity:

- Android is the dominant operating system in low-income countries.
- Many Android users do not own credit cards. Contrast this to Apple, who reported it had 800 million cards on file as of March 2014 [42].
- Android devices are cheaper than Apple's offerings, and people willing to spend more usually choose iOS.
- Apple delivers a very solid value proposition to its target customers, with a strong ecosystem of apps and support.
- iOS attracts developers due to its increased revenue and excellent developer tools.

So how does this translate to App Store revenue by app category and business (i.e., pricing) model? A survey from April 2014 (figure 2.13) showed that gaming apps had the highest "free with IAP" percentage (92%)[17], while navigation had the lowest (21%) [43]. Regarding paid apps without IAP, 68% of medical apps adopted this model, with only 4.4% coming from social networking apps. The category with the most paid with IAP apps was navigation (28%), and only 0.9% of catalog apps used this model.

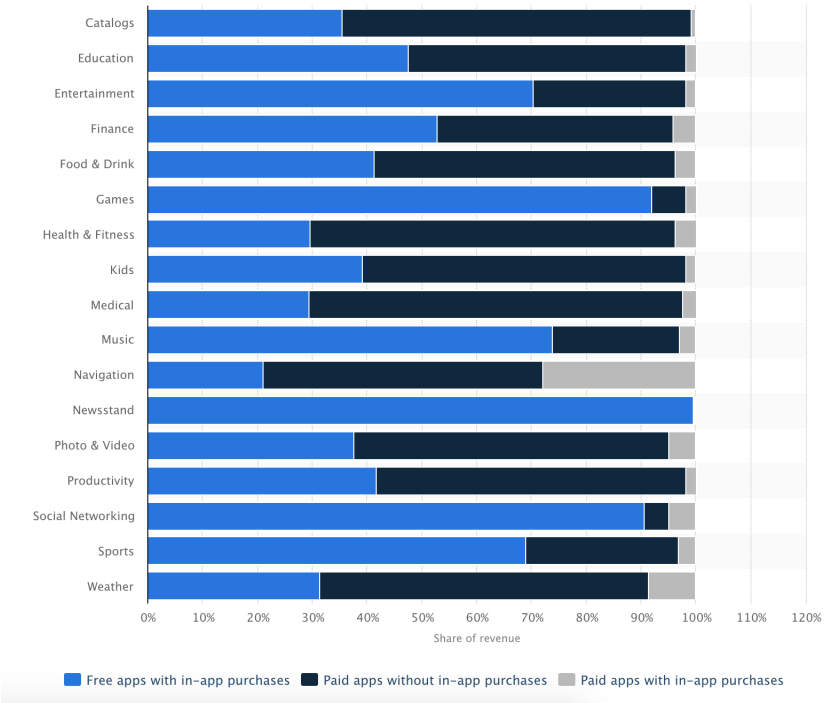


Figure 2.13: Worldwide app category revenue distribution in the Apple App Store in February 2014, by business model

[17] The Newsstand category was not considered as this category was removed after Apple released the News app with iOS 9. It is merely shown for statistical purposes.

Existing wish list management apps

To form the functional requirements of our wish list management app, we had to search the App Store to see which apps already existed in this niche. It is not a very popular Shopping sub-category because apps from high-profile companies (e.g., Amazon, eBay, Walmart) and that help users obtain discounts at those stores dominate the category's top lists.

We analyzed three free apps[18] considered closest to the two core requirements a wish list management app must have: the ability to add items and share wish lists.

1. **WishMindr** — Create & share wish lists for any occasion (v. 1.1.8)

   This is a cross-platform app (iOS and Android) that offers integration with popular stores, such as Amazon, eBay, and etsy [44]. It has a very simple, almost "stock" UI (figure 2.14). Wish list sharing is done through their website which requires no prior registration or downloads to view.

   However, there are a few snags. There is no onboarding[19] for new users, the UI does not update in real-time, customization is limited, and there is no way to take photos of items.

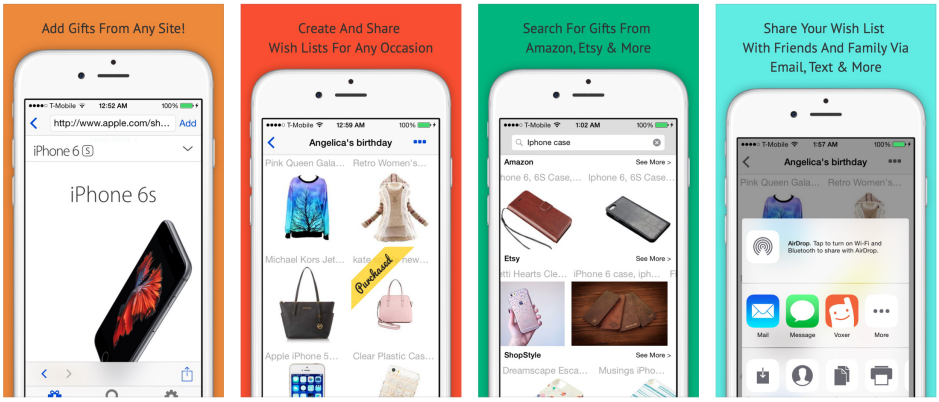


Figure 2.14: WishMinder app screenshots

2. **Giftster** — wish list register for holiday, birthday, baby (v. 4.0)

   This is a universal iOS app which provides good user onboarding, although the main login view is not native [46]. It provides many options for creating and sharing lists. Groups — a unique feature of this app — allows for collaborative lists between circles (e.g., family, friends).

   On the downside, sharing requires users to download the app to view lists, there is no 3D Touch support for quick actions or peeking into lists, and the app frequently crashed when trying to add an item.

3. **Giftry** — Universal Wish List and Gift Registry (v 2.6.3)

[18] Analysis was done based on the US App Store.

[19] User onboarding, as UserOnboard aptly describes, "is the process of increasing the likelihood that new users become successful when adopting your product" [45]. The onboarding process is typically the first experience users have with an app (usually post-registration), and as such, it should mention the core features and how to quickly get started. In essence, it is similar to a (brief) tutorial.

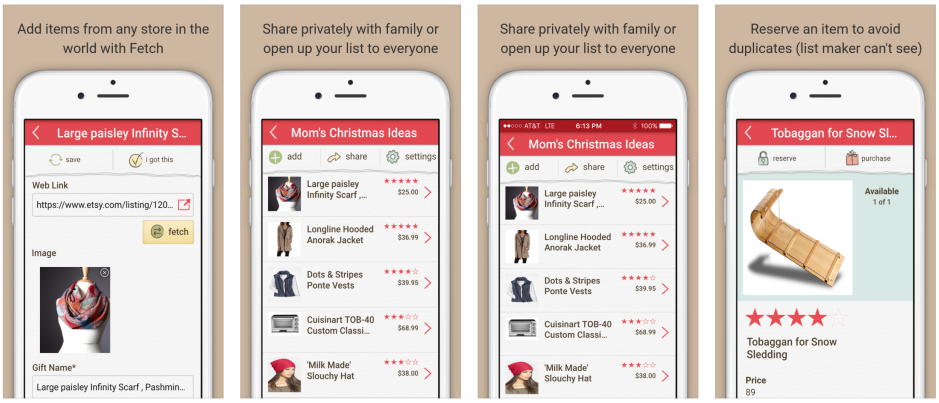


Figure 2.15: Giftster app screenshots

An iOS and Android app that features a custom UI with clean lines and modern, flat design (figure 2.16) [47]. Out of the three, Giftry provided the best user onboarding experience, albeit the process was too lengthy (requiring users to swipe on 15 items after signing up, which cannot be skipped).

Feature-wise, it provides multiple ways to sign up (Facebook login and regular sign-up), easy sharing of wish lists, and allows taking and selecting photos of items. Users can also collaborate on lists.

Downsides include users needing to have the app to view lists, poor gift website integration (compared to WishMindr), and scanning a barcode almost never recognized one as valid.

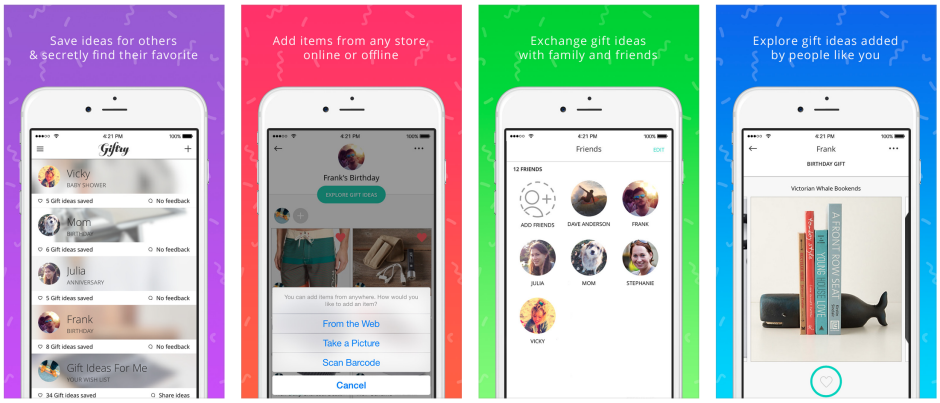


Figure 2.16: Giftry app screenshots

Table 2.3 summarizes the comparison between these three apps. It is important to note all three apps failed to work offline: WishMindr popped up an intrusive alert view, Gifster would not load its startup UI (because it is mostly a web app), and Giftry crashed if users tried to create a wish list.

Since WishMindr's main use case is adding items from online stores, working offline is not a requirement, but the other two apps could offer users an offline cache that showed them their current wish lists and allowed them to take photos of items. Changes would then sync as soon as a data connection could be established.

Given the circumstances of a niche market segment with no apps that stood out, we can assert that the key requirements for our app is offline cache support and usage, sharing without requiring users to download our app, and 3D Touch support. Collaborative wish lists would be a nice feature, but not essential because it is necessary to first perfect features which would make our app unique and create value for our potential customers.

Table 2.3: Comparison between iOS wish list management apps

Name	Features	Limitations
WishMindr	• Cross-platform app • Integration with gift websites • Wish list sharing • Website component	• Account required • No onboarding for new users • No real-time updates • Default UI • Limited customization • No way to take photos of items • No offline support
Gifster	• Good onboarding • Wish list sharing • List privacy options • Customization options	• Web views • Sharing requires app download • No real-time updates • Frequent crashes • No offline support
Giftry	• Cross-platform app • Wish list sharing (collaborative) • Website component • Can add photos of items	• Lengthy sign-up process • Sharing requires app download • Hard to search gift websites • No offline support

2.4 Summary

This chapter provided a perspective to comprehend the issues that arise with problematic apps; namely, the degradation of the user experience. This degradation is a serious consequence, and we proposed a solution that will analyze the current tendencies concerning criticized apps to try and discover their common issues, as well as develop a wish list management app that showcases best engineering practices. To aid with this, we also included answers to thirty-five engineering questions.

The value analysis section presents business aspects such as our value proposition, how we can deliver value to the customer, what sort of negotiation scenarios we might face, and a description of the nine building blocks that make up the Business Model Canvas (the Canvas itself is in Appendix A). It also includes an example of the AHP applied to determining which is the most appropriate wish list app to download according to a series of criteria.

The state-of-the-art section is laid out in a bottom-up perspective, which means that we first detailed the hardware that powers iOS and subsequently move our way to cover technologies (both hardware and software-based), frameworks and finally, an overview of the App Store and related apps. Regarding the latter, we analyzed three free wish list management apps — WishMindr, Gifster, and Giftry — and their features and weaknesses were concerned. None had real-time sync or offline support (features we plan on implementing in our app).

Chapter 3

Analysis of iOS Apps, Technologies & Frameworks

This chapter introduces some aspects of Apple's iOS Human Interface Guidelines, including its fundamental principles, app anatomy, and details on color and typography. Next, we assess ten popular iOS apps in quantitive and qualitative metrics (e.g., battery consumption, lack of user onboarding, and adherence to the iOS HIG) in an attempt to discover common patterns of nonconformity.

Furthermore, we focus on detailing some technologies and frameworks developers can use in iOS app development including, but not limited to an overview of the Swift programming language (and its benefits versus Objective-C); popular backend providers; and package managers to streamline project dependencies.

3.1 Apple's Human Interface Guidelines

The iOS HIG are the basis for app design [48]. Developers must follow these guidelines or risk having their apps rejected during the App Store review process[1].

During the iOS 7 redesign, Apple defined three key iOS design principles:

- **Deference.** *The UI helps people understand and interact with the content, but never competes with it;*
- **Clarity.** *Text is legible at every size, icons are precise and lucid, adornments are subtle and appropriate, and a sharpened focus on functionality motivates the design;*
- **Depth.** *Visual layers and realistic motion impart vitality and heighten people's delight and understanding.*

Deference pertains to guaranteeing the user's content is at the heart of the UI: taking advantage of the whole screen, favoring content over heavy visual indicators (e.g., bezels, gradients, drop shadows) and using translucent elements appropriately to hint at content behind them.

[1] According to section 10.1 of the App Store Review Guidelines: "Apps must comply with all terms and conditions explained in the applicable Apple Human Interface Guidelines: [...] iOS Human Interface Guidelines." [49].

Clarity focuses on the use of negative space to highlight content and make functionality more noticeable. A key color helps simplify the UI (e.g., the stock Calendar app's color is red, Messages is blue, Notes is yellow). Ensuring legibility by using the default system font is also a good strategy, as Dynamic Type[2] automatically adjusts spacing and line height to fit content. Finally, buttons should be borderless unless there are strong visual cues that separate them from non-interactive UI elements.

Depth gives the sense of content in distinctive layers that convey hierarchy and position. For example, 3D Touch's peek, pop and quick actions give users access to additional functionality without losing their context.

3.1.1 iOS App Anatomy

Almost all iOS apps use the UIKit framework for their UI components. Apple divides these into four ample categories [50]:

- **Bars.** *Bars contain contextual information that tells users where they are and controls that help users navigate or initiate actions (e.g., navigation bar, tab bar);*
- **Content views.** *Content views contain app-specific content and can enable behaviors such as scrolling, insertion, deletion, and rearrangement of items (e.g., table view, scroll view);*
- **Controls.** *Controls perform actions or display information (e.g., buttons, sliders, labels);*
- **Temporary views.** *Temporary views appear briefly to give users important information or additional choices and functionality (e.g., alert views, action sheets).*

To manage a set of views, one must use a *view controller*. It manages the display of views, user interactions, and transitions between view controllers (screens[3]). Figure 3.1 shows an example of the anatomy of an iOS app and its hierarchy.

3.1.2 Color and Typography

Color

In iOS, color is essential in conveying vitality and visual continuity, and indicating interactivity. One should experiment with different color schemes to make sure they look good on light and dark backgrounds and provide enough contrast[4] to ensure visibility in different lighting scenarios.

[2] Introduced in iOS 7, this is a setting allowing apps which support it to adjust to the user's preferred reading size.

[3] Users typically consider an app to be a collection of screens. In this context, we can assert that a user's definition of *screen* is very similar to a developer's definition of *view controller* (from a pure UI perspective) — "a distinct visual state of mode in an app" [50].

[4] Apple recommends apps have a contrast ratio of 4.5:1 or higher [51]. This ratio means that, at the same brightness level, the brightest color (white) should, at least, be 4.5 times brighter than the darkest color (black) that the device's screen is capable of reproducing.

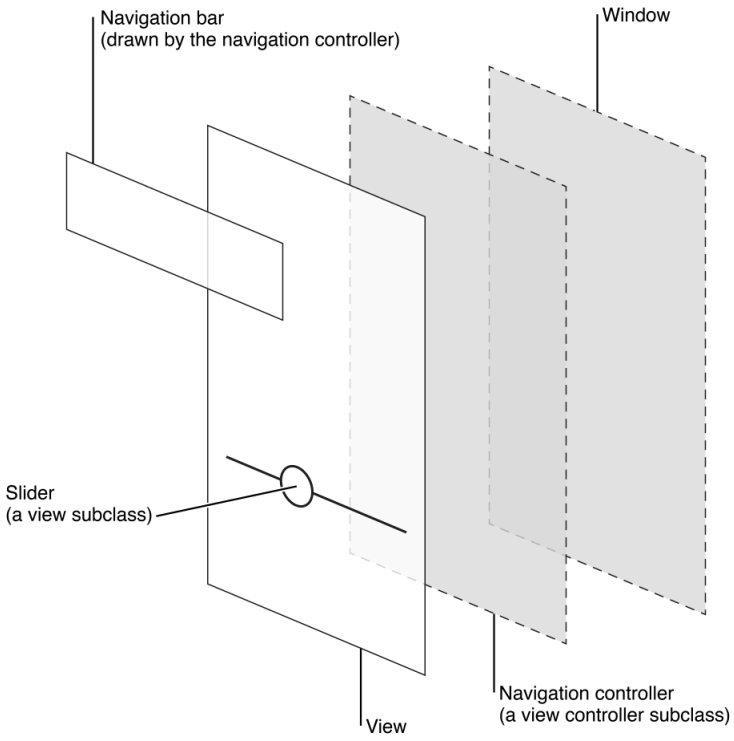


Figure 3.1: Anatomy and hierarchy of an iOS app

Apple recommends eight main guidelines to follow regarding color usage [51]:

- **Appropriate color scheme**: it should match other branding elements (e.g., App icon, marketing materials) and the app's style (e.g., pastel colors, vibrant colors);
- **Attention to context**: to make UI elements discernible, taking into account contrast, lighting and usage patterns (i.e., where the app will predominately be used) must be part of the design process;
- **Translucency**: should be used sparingly to avoid making content illegible (e.g., white text against a light translucent backdrop);
- **Color blindness**: avoiding the usage of red and green to indicate two distinct visual states, as many color blind people cannot distinguish between the two;
- **Color diversity**: developers should avoid using the same color for interactive and non-interactive elements (e.g., using blue for both buttons and labels);
- **Cultural differences**: different countries and cultures perceive color in distinct ways. It is important to research how the meaning of certain colors can affect the app's target market;
- **Avoiding distractions**: colors should not detract from an app's most important asset — its content.

Typography

In iOS 9, Apple replaced Helvetica Neue with the San Francisco family of typefaces to provide "a beautiful, consistent typographic voice and reading experience across all platforms" (figure 3.2) [51]. San Francisco contains two optical sizes: Text and Display. Text is used for sizes below 20 points[5]; 20 points and higher use Text. iOS automatically switches between sizes when using the system font in an application.

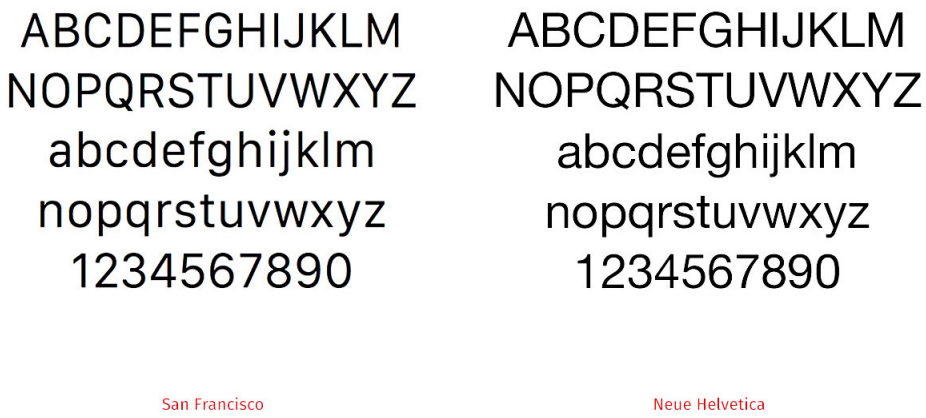


Figure 3.2: Comparison between San Francisco and Helvetica Neue

Apple recommends developers to focus on prioritizing relevant content when adapting to text size changes, making sure all styles of a custom font are legible at different sizes and using a single font to showcase consistency throughout an application's design.

3.2 App Analysis

The Apple HIG is the major parameter in making sure an app conforms to iOS design guidelines. Nonetheless, there are some popular apps on the App Store which do not follow these guidelines (and other factors).

There are many qualitative (i.e., subjective) measures we can take into account to judge an app (e.g. design quality, fancy animations), but we must seek a *quantitative* evaluation — metrics — as these are testable and objective.

The following metrics were used in evaluating these apps:

- Popularity[6] (App Store ranking);
- App Store ratings;
- User onboarding;
- Adherence to iOS design guidelines;
- Battery consumption;

[5] Note that *points* are not necessarily equal to *pixels*: on a retina display (scaling factor of 2), 1 point = 2 pixels; on the iPhone 6+ (scaling factor of 3), 1 point = 3 pixels.

[6] Accurate as of October 23, 2016.

- App Transport Security[7];
- Data retention and privacy policies.

This section presents ten popular apps that have been criticized by the media and users for various offenses, such as failure to follow iOS design standards, excessive battery consumption, lack of or confusing onboarding, among others. The goal is to gather insight regarding repeated bad practices and their popularity, which can be used as a starting point for our app's testing and user experience goals (e.g. testing battery consumption to make sure it is within acceptable parameters, implement user onboarding). Data is relevant as of September 2016.

3.2.1 Facebook

Facebook's official iOS app has been widely criticized for years, primarily for its exaggerated battery consumption [53, 54]. Here are the stats (v. 63.0):

- **Popularity**: Tenth in Top Free iPhone Apps (Second in Social Networking)
- **App Store ratings**: 3.5/5 stars (current version, 878 ratings); 3.5/5 stars (all versions, approximately 2.9M ratings)
- **User onboarding**: Facebook for iOS does not feature any sort of onboarding.
- **Adherence to iOS design guidelines**: designed for both iPhone and iPad. The app uses iOS design elements (figure 3.3 [55]) as well as standard and custom transitions.

Figure 3.3: Facebook app screenshots

- **Battery consumption**: the major issue with this app. Installing the Facebook app and leaving the device idle for 24 hours resulted in Facebook taking the top spot for battery consumption. Facebook was widely criticized for playing silent background audio to extend its background timeout period[8] [54], enabling it to run indefinitely. Many users still report this issue (even after Facebook claimed they would fix it) [56].

[7] Introduced in iOS 9, App Transport Security "improves the privacy and data integrity of connections between an app and web services by enforcing additional security requirements for HTTP-based networking requests" [52]. Specifically, apps must use HTTPS when making these requests.

[8] By default, iOS limits apps' background execution to ten minutes, except for special cases (e.g., playing music, VoIP apps, navigation apps).

- **App Transport Security**: Facebook has **opted out** of ATS entirely [57].
- **Data retention and privacy policy**: Facebook collects a myriad of information (e.g. personal information, networks & connections, payments, device information) that can be shared with third-parties for advertising purposes. However, Facebook is transparent on its data retention policies [58]:

  > *We store data for as long as it is necessary to provide products and services to you and others, including those described above. Information associated with your account will be kept until your account is deleted, unless we no longer need the data to provide products and services.*

3.2.2 YouTube

The official iOS app for the world's most popular video sharing website suffered heavy backlash from users in October 2015 [59]. Its redesign ported Android's "Material Design" elements over to iOS, instead of focusing on the user experience and new features introduced in iOS 9 such as iPad multitasking and picture-in-picture. Here is a stat breakdown as of version 11.33:

- **Popularity**: 11th in Top Free iPhone Apps (Third in Photo & Video)
- **App Store ratings**: 3.5/5 stars (current version, 243 ratings); 2.5/5 stars (all versions, approximately 222k ratings)
- **User onboarding**: YouTube does not feature any sort of onboarding.
- **Adherence to iOS design guidelines**: designed for both iPhone and iPad. However, Google **does not adhere** to Apple's HIG and instead imports "Material Design" design elements from Android (figure 3.4 [60]).

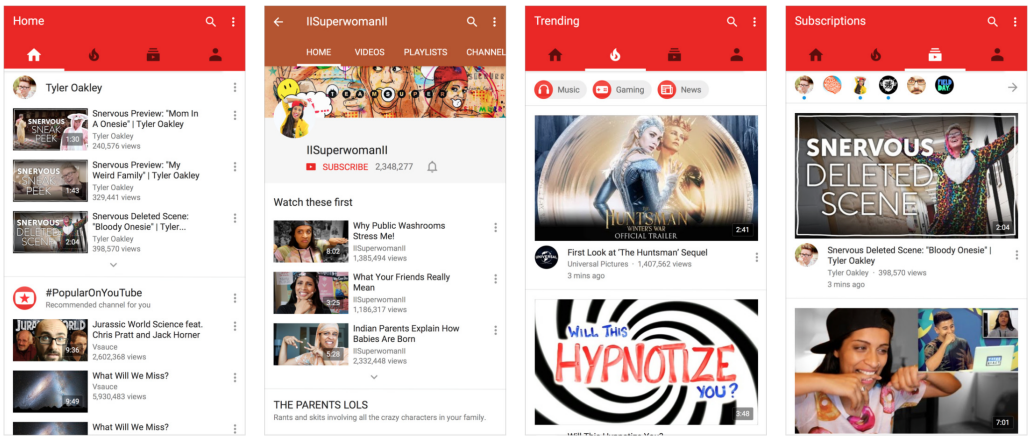


Figure 3.4: YouTube app screenshots

- **Battery consumption**: there have been user complaints throughout the years that YouTube drains battery at a rapid pace when in the foreground, but this is to be expected using a a significant amount of data (Wi-Fi/cellular) to stream video content. That being said, YouTube pauses the video when it ceases to be the foreground app or when the device is locked to preserve battery life.

- **App Transport Security**: YouTube has **opted out** of ATS entirely [57].
- **Data retention and privacy policy**: Google uses data gathered from YouTube (e.g. likes, watched videos, searches) to enhance the user's experience when using any other Google products or services. For example, if a user searches for and watches videos related to technology, they are more likely to see technology-related ads when using Google Search.

  It is also important to note that videos deleted from YouTube by the user are not actually deleted from Google's servers, and that content can be removed at any time at their discretion [61].

3.2.3 Dropbox

Dropbox's iOS app — currently in version 15.2 — has seen mixed reviews, with a third of its reviewers deciding it deserves a measly one star. Problems cited include difficulty syncing photos, an outdated UI, lack of reliability when storing data, and slow upload speeds. Its breakdown is listed below:

- **Popularity**: 49th in Top Free iPhone Apps (Seventh in Productivity)
- **App Store ratings**: 3/5 stars (current version, 45 ratings); 3.5/5 stars (all versions, approximately 48.5k ratings)
- **User onboarding**: Dropbox offers user onboarding which shows new users how to use their file hosting service and what they can do with the app[9].
- **Adherence to iOS design guidelines**: designed for both iPhone and iPad. For the most part, Dropbox's iOS app uses Apple's standard controls to simplify its User Interface (figure 3.5 [63]). However, it has not seen changes since 2013 (to fit in with iOS 7's new design guidelines).

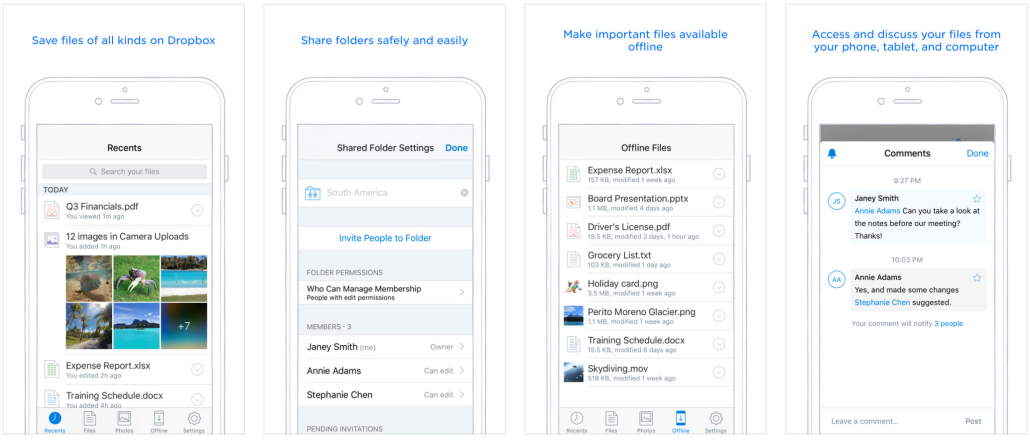


Figure 3.5: Dropbox app screenshots

- **Battery consumption**: the Dropbox app utilizes background uploading to sync Camera uploads (if the feature is turned on). Dropbox notes that when the device's battery drops below 30%, syncing is automatically disabled to preserve battery life.

[9] A detailed overview on how Dropbox built the onboarding experience can be found on their tech blog [62].

- **App Transport Security**: Dropbox has **opted out** of ATS entirely [57].
- **Data retention and privacy policy**: in a 2014 blog post, Dropbox stated "all files sent and retrieved from Dropbox are encrypted while traveling between you and our servers" [64]. That said, it does not encrypt the actual files when they are present on the devices, which sourced criticism from Edward Snowden. He also stated an alternative in SpiderOak, which provides full encryption of local files.

3.2.4 Snapchat

The current leader of iOS's social networking category, Snapchat's iOS app (v. 9.38.0.0) has been recently criticized by its users for loading issues and crashes when opening Snaps (i.e., pictures or videos), resulting in lost streaks[10]. Snapchat's stats are listed below:

- **Popularity**: Seventh in Top Free iPhone Apps (First in Photo & Video)
- **App Store ratings**: 3.5/5 stars (current version, 242 ratings); 2.5/5 stars (all versions, approximately 249.9k ratings)
- **User onboarding**: Snapchat provides user onboarding that is specifically tailored for the interactions in the iOS app [65].
- **Adherence to iOS design guidelines**: only designed for iPhone. Given the current state of iPad hardware (i.e., decent camera quality), it is not clear why Snapchat did not develop a universal app.

  The user interface is custom-built (figure 3.6 [66]), but iOS's standard view controller flow is present. Snapchat takes advantage of Apple hardware and APIs to do image processing (e.g., filters) on the fly.

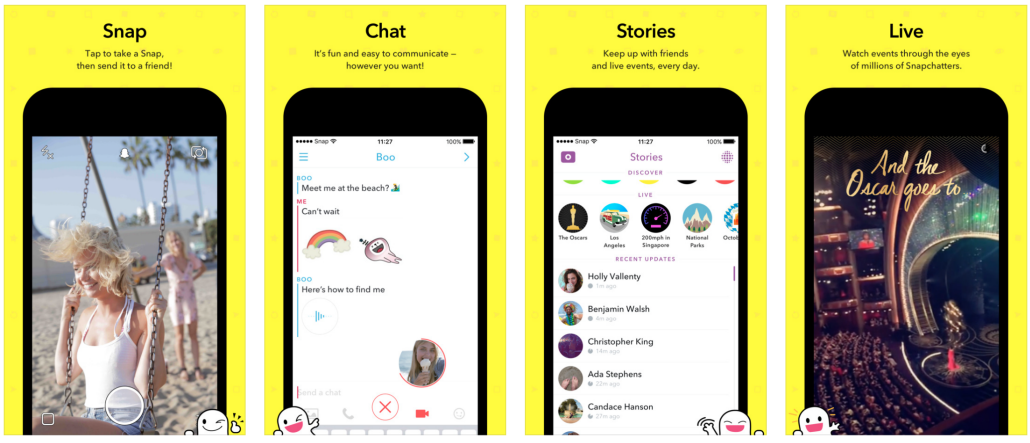


Figure 3.6: Snapchat app screenshots

- **Battery consumption**: Snapchat is frequently a battery hog if Background App Refresh (BAR) is turned on. It is recommended that users disable BAR entirely or for Snapchat itself, and enable Snapchat's "Travel Mode" (which only loads data on-demand to save cellular data — and battery) [67].

[10] A streak (or "Snapstreak") represents the number of consecutive days when two people send each other Snaps.

- **App Transport Security**: Snapchat has **enforced** ATS on their domains; however, they allow arbitrary loads for insecure domains that the app accesses [68].
- **Data retention and privacy policy**: Snapchat operates in a way that a Snap is deleted from its servers as soon as the recipient has visualized the content of the Snap. However, this does not prevent the recipient from using external methods to save a Snap. Thus, users should be careful when sharing content they intend to be ephemeral.

  Also, when creating Stories (i.e., a collection of Snaps), if it is public, Snapchat may retain it indefinitely.

3.2.5 Facebook Messenger

This is Facebook's companion app dedicated to messaging. In August 2014, Facebook made this app obligatory for all users who wished to use Facebook's chat feature on iOS and Android. The app has been met with average to mixed reviews, with many users lamenting removed functionality in the latest update, and the requirement of two separate Facebook apps (among other sparse issues). Here is how it stacks up:

- **Popularity**: Eighth in Top Free iPhone Apps (First in Social Networking)
- **App Store ratings**: 3.5/5 stars (current version, 125 ratings); 3/5 stars (all versions, approximately 326.5k ratings)
- **User onboarding**: Facebook Messenger offers simple onboarding [69]. However, there could be more emphasis on its distinct features (e.g. stickers).
- **Adherence to iOS design guidelines**: designed for both iPhone and iPad. The app uses iOS design elements and standard transitions (figure 3.7 [70]).

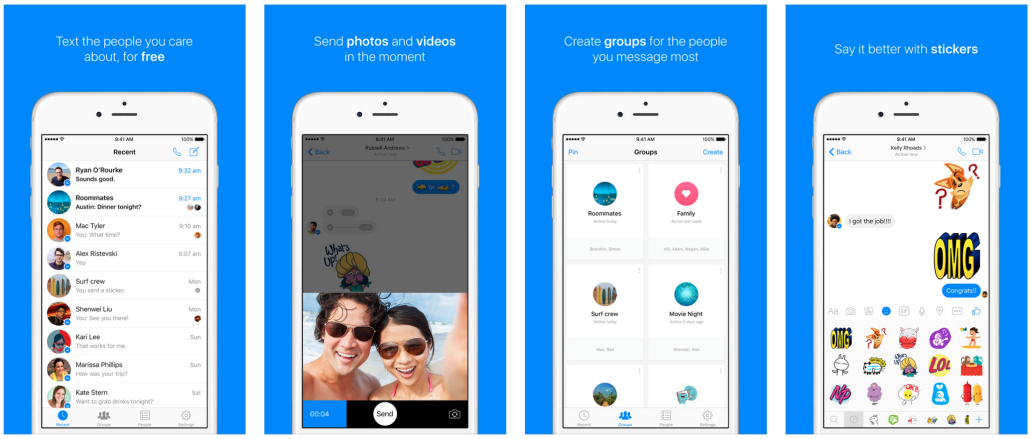


Figure 3.7: Facebook Messenger app screenshots

- **Battery consumption**: Messenger suffers from the same issue that plagues the Facebook app: excessive battery consumption due to intentional backgrounding [53].
- **App Transport Security**: Facebook has **opted out** of ATS entirely [57].
- **Data retention and privacy policy**: as Facebook Messenger is a companion app to Facebook's official iOS app, the same policies apply (mentioned above).

3.2.6 Amazon

The world's most popular e-commerce website — Amazon — has fittingly the top spot in the top free iOS shopping apps. Despite its popularity, it is not without its flaws: users have consistently complained about slow loading speeds, random crashes, lack of Apple Pay support, and laggy scrolling on older devices. These are its stats:

- **Popularity**: 23rd in Top Free iPhone Apps (First in Shopping)
- **App Store ratings**: 3.5/5 stars (current version, 476 ratings); 3.5/5 stars (all versions, approximately 121.6k ratings)
- **User onboarding**: Amazon does not feature any sort of onboarding.
- **Adherence to iOS design guidelines**: designed for both iPhone and iPad. Amazon's app uses their own voice recognition and camera item detection to identify items the user wishes to order, making it easier to shop. It takes advantage of Apple's hardware (e.g., microphones and camera), as well as having a simple user interface that conforms to the HIG (figure 3.8 [71]).

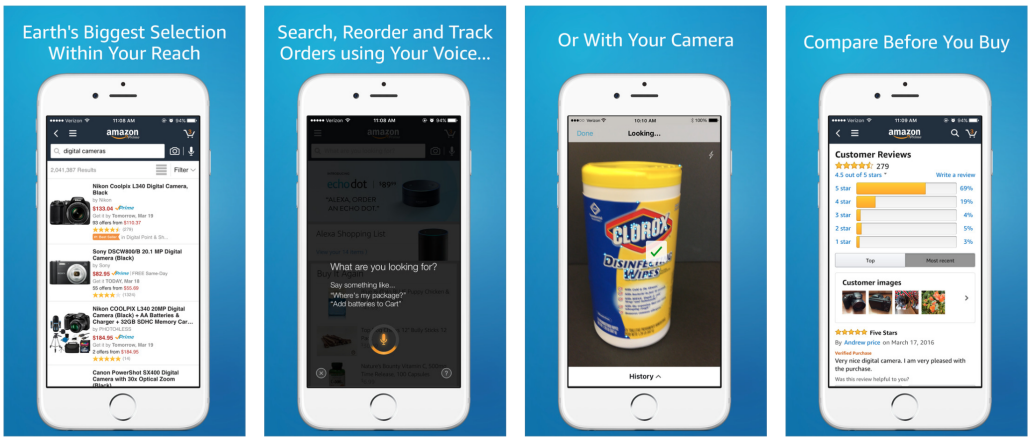


Figure 3.8: Amazon app screenshots

- **Battery consumption**: there have been no reports of excessive background battery usage. Standard battery consumption tips apply (i.e., turn off BAR for Amazon).
- **App Transport Security**: Amazon has **opted out** of ATS entirely [57].
- **Data retention and privacy policy**: Amazon's privacy policy states that their service enables third-party advertisers to track customers **by default** and that they may sell user datas as part of a business transfer. Amazon also tracks users on other websites to recommend targeted products. This applies across all their services (website and mobile apps). Users can disable advertising tracking and the user info Amazon has access to [72].

3.2.7 Twitter

The social networking giant's iOS app has come a long way in the last few years, with some exclusive features such as "Moments" and polls. Despite this, some users are displeased due to stuck/duplicate notifications, delays when receiving them, and slow content loading

times. Also interesting to note is that Twitter's official app does not support "Streaming"[11], a feature introduced over six years ago to developers and which currently many third-party clients take advantage of. As of version 6.54, here are its stats:

- **Popularity**: 26th in Top Free iPhone Apps (First in News)
- **App Store ratings**: 3.5/5 stars (current version, 34 ratings); 3.5/5 stars (all versions, approximately 341.2k ratings)
- **User onboarding**: Twitter features user onboarding when signing up, helping new users find and invite friends, follow suggestions, and set up their profile [73].
- **Adherence to iOS design guidelines**: designed for both iPhone and iPad. The iPad version, however, is just an upscaled version of the iPhone's design with too much white space, as shown in figure 3.9 [74].

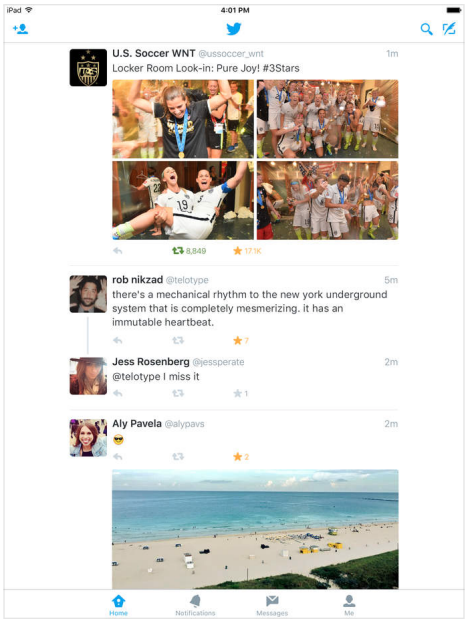


Figure 3.9: Twitter app running on an iPad

- **Battery consumption**: it is recommended that users disable BAR for Twitter to avoid excessive refreshes in the background which consume data and battery at a rapid pace [75].
- **App Transport Security**: Twitter has ATS enabled for their own domains, but allows arbitrary loads for third-party content (such as in-app advertisements — iAds).
- **Data retention and privacy policy**: Twitter's retention policy states that any deleted account (initiated by the user) is purged after 30 days of inactivity, but data rights are kept by Twitter in the event.

  Users can request an archive of their tweets. Tracking data is deleted after 10 days and can be opted out [76].

[11] This allows clients to display incoming tweets on a user's timeline in real-time (i.e., no refresh required).

3.2.8 Whatsapp

Whatsapp is the worldwide leader in cross-platform messaging with over one billion monthly active users (MAU) [77]. Despite the average review score for its iOS app being above average (4 stars), Whatsapp has been criticized in the past for releasing belated updates to support new iOS features.

- **Popularity**: 16th in Top Free iPhone Apps (Third in Social Networking)
- **App Store ratings**: 4.5/5 stars (current version, 586 ratings); 4/5 stars (all versions, approximately 172.8k ratings)
- **User onboarding**: Whatsapp provides user onboarding when signing up, helping new users register their number with Whatsapp, inviting friends not on Whatsapp from their contact list, and setting up their profile.
- **Adherence to iOS design guidelines**: designed solely for iPhone. As Whatsapp needs a phone number to work, it does not support iPad or iPod touch.

  Whatsapp complies with Apple's HIG (figure 3.10 [78]), although there have been many delays in the past to support new UI features. In 2013, it took Whatsapp almost 3 months to release an updating complying with iOS 7's interface guidelines (e.g., new design, Background App Refresh) [79]. A year later, when Apple introduced larger screen iPhones, Whatsapp only released an update 2 months later which supported the screen sizes of the iPhone 6 and 6 Plus [80].

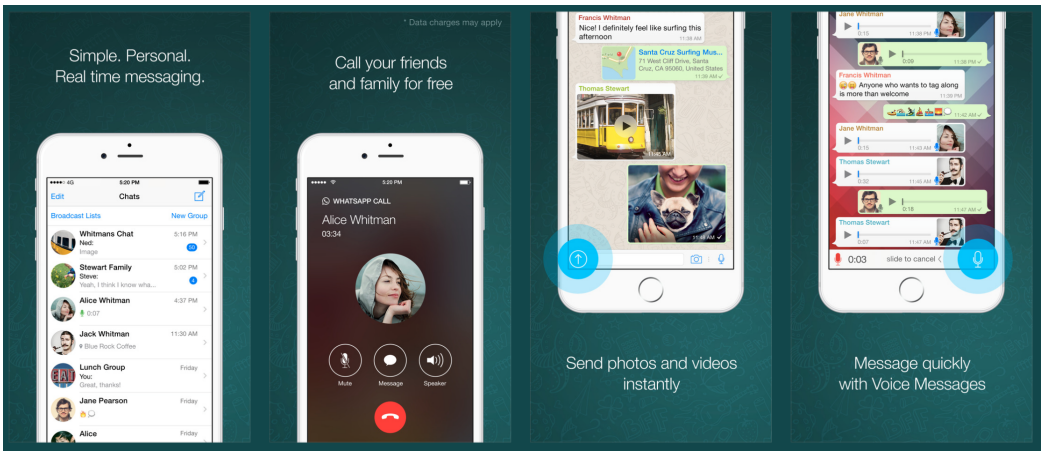


Figure 3.10: Whatsapp app screenshots

- **Battery consumption**: Whatsapp users should disable BAR to avoid it auto-downloading received media in the background. This does not affect or add delay to notifications of incoming messages.
- **App Transport Security**: according to Whatsapp's security white paper: "All communication between WhatsApp clients and WhatsApp servers is layered within a separate encrypted channel." [81].
- **Data retention and privacy policy**: Whatsapp was ranked joint-worst on an Electronic Frontier Foundation (EFF) report, being criticized for not publishing a transparency report or law-enforcement guide. It does not also provide a data retention policy [82].

3.2.9 SoundCloud

The leading online audio distribution platform had over 175 Million Active Users (MAU) as of December 2014 [83]. In the past year, however, the company's iOS app has received criticism for frequent crashes which have not yet been fixed (at the time of writing). This has caused its ratings to plummet from four stars out of five to 2.5.

On a larger scale, the majority of users are complaining about listening limitations and in-app ads pushing users to subscribe to SoundCloud Go, the company's premium subscription service.

- **Popularity**: 43rd in Top Free iPhone Apps (Third in Music)
- **App Store ratings**: 4.5/5 stars (current version, 699 ratings); 4/5 stars (all versions, approximately 112.2k ratings)
- **User onboarding**: SoundCloud has user onboarding showing users how to search for artists/producers, build collections and explaining the way their visual player works [84].
- **Adherence to iOS design guidelines**: designed for iPhone and iPad. The app uses iOS design elements and standard transitions, as seen in figure 3.11 [85].

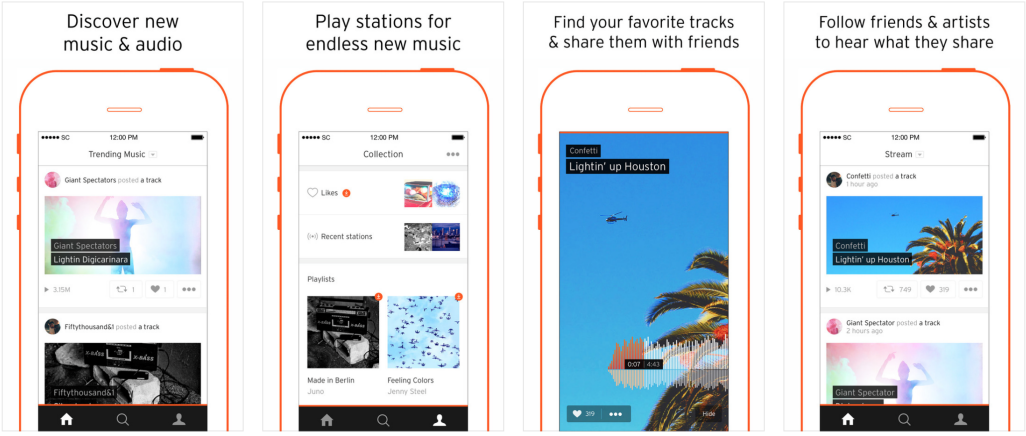


Figure 3.11: SoundCloud app screenshots

- **Battery consumption**: there have been no reports of excessive background battery usage, but users who turn off BAR may experience extended battery life.
- **App Transport Security**: SoundCloud enables ATS on their domain (soundcloud.com), but allows arbitrary loads for third-party media.
- **Data retention and privacy policy**: SoundCloud states that users are in control of their copyright (i.e., uploaded media) and collected data is used for limited purposes [86]. Changes to the Terms of Service (ToS) take effect 6 weeks after users are notified [87].

3.2.10 NFL Fantasy Football

NFL.com's official fantasy iOS app is used by fantasy football players during the season (which runs from August to January). While the app finally received an update to support

larger iPhone screens in August 2016, users' negative reviews consider the redesigned UI and missing features (e.g. inability to view weekly match-ups and chat removal) a significant step back from the previous version.

- **Popularity**: 121st in Top Free iPhone Apps (Third in Sports)
- **App Store ratings**: 1.5/5 stars (current version, 465 ratings); 3.5/5 stars (all versions, approximately 5.4k ratings)
- **User onboarding**: there is no user onboarding for this app.
- **Adherence to iOS design guidelines**: designed for iPhone and iPad. The app uses iOS design elements with a simple grey and blue design (figure 3.12 [88]).

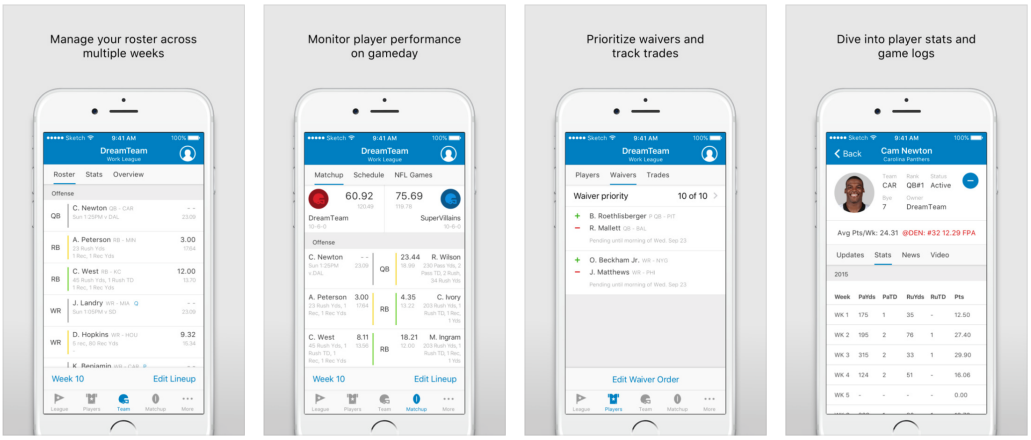


Figure 3.12: NFL Fantasy Football app screenshots

- **Battery consumption**: there have been no reports of excessive background battery usage. The app does drain a lot of battery in the foreground during live games as it has to poll NFL's API frequently.
- **App Transport Security**: NFL.com has opted-out of ATS entirely for this app.
- **Data retention and privacy policy**: NFL.com's privacy policy applies to its iOS mobile app [89]. It states that data collected by the NFL can be used for promotional and contact purposes. Some data may also be shared with third-parties (e.g., member clubs, advertisers, business partners). Users can opt-out of traffic and cookie tracking on NFL's website.

3.2.11 Conclusions

Our analysis of these ten apps spanning multiple categories provided an adequate sample size to find similarities. Here are our conclusions for each reviewed aspect.

User onboarding

Six out of ten apps provide user onboarding (the offenders being Facebook, YouTube, Amazon, and NFLFF). Of the apps that onboard new users, Snapchat and Twitter both offered intuitive ways for these to learn how to use their apps and specific basic interactions.

Between chat apps, Whatsapp wins as Facebook Messenger fails to communicate the unique features of the app. For example, stickers are part of Facebook, but we only discovered that we could use them on Messenger after sending a few messages and exploring the emoji section. We were intrigued yet disappointed with the lack of onboarding for the Facebook app because Facebook provides onboarding for their companion Messenger app (interestingly, the latter is more popular according to the App Store rankings).

Overall, while many popular apps have some form of user onboarding, there is still a significant percentage that don't (40%). Apple should make this mandatory as an effort to enforce user experience standards (which may lead to higher user retention percentages).

Adherence to iOS design guidelines

For the most part, the apps we analyzed adhere to Apple's HIG (90%). However, YouTube stands out because it uses Android's Material Design instead of UI elements familiar to iOS.

Battery consumption

Out of the ten apps, Facebook's were unsurprisingly the worst offenders. Even after Facebook disabled some of its shady backgrounding last year, both apps continue to consume battery at a rapid pace, especially in the foreground.

When it comes to active battery drain (i.e., in the foreground), NFL Fantasy Football and YouTube consistently tally high percentages. This is due to frequent polling and streaming video, respectively.

App Transport Security

The results here show that 60% of the apps we tested still opt-out of ATS entirely on their app almost a year after Apple announced this feature, a disappointing scenario.

WhatsApp is the only app that enforces ATS between clients and their servers. The other three companies (Snapchat, Twitter, and SoundCloud) only enable it on their domain, but allow arbitrary loads for third-party content. While having all content secured would be the goal, this is still indicative that these companies have paid attention and enabled this feature for the benefit and security of their users.

Given these circumstances, Apple will make ATS mandatory for all iOS apps by the end of 2016 [90]. This move will finally spur developers to secure their domains and content to achieve a (more) protected environment for every user.

Data retention and privacy policy

We were disappointed with most companies' privacy policies.

Whatsapp was the worst in our analysis, as it does not provide a transparency report or even a data retention policy.

Facebook (and Messenger), YouTube, Dropbox also ranked poorly. While these companies are transparent about their affairs, any privacy-conscious user should avoid them due to their retention of user information for commercial and data mining purposes.

Twitter and SoundCloud ranked the highest. They are transparent about their policies and, most importantly, provide opt-out possibilities. SoundCloud also informs its users about policy changes in advance.

Summary

We found that App Store ratings were rarely indicative of the quality of the app itself. This was due to a couple of factors: varying sample size (proportional to popularity), and users lamenting unrelated issues. Ratings also tend to vary depending on the quality of a specific app update. Nonetheless, we included them as a preliminary guideline (for the current version and overall).

Overall, this analysis provided crucial insight to the major issues plaguing widely-used App Store apps. As a result, a testing process should consist of, but not be limited to:

- Energy consumption tests
- Memory allocation and leak tests
- Time profiling
- Network requests

We must also focus on non-quantitative metrics such as security and usability by adopting ATS and making sure we provide a concise and clear onboarding experience (among other aspects).

3.3 Technologies

Nowadays, there are many ways to develop an iOS app, starting with the programming language. One can pick between the mature Objective-C, the modern and fast Swift, or even other languages provided by cross-platform mobile application development tools such as Xamarin (which uses C# as the main programming language).

This section covers core technologies and libraries which can be used when developing an iOS app. This includes the programming language, an overview on some iOS-friendly backend providers, and a description of the commonly used dependency managers and their catalog of libraries and frameworks.

3.3.1 The Swift Programming Language

When it comes to developing native iOS apps, Apple supports two programming languages: Objective-C (which has been around since 1984 when Steve Jobs was at NeXT) and Swift, a relatively modern language that is just over two years old.

Apple unveiled Swift at WWDC 2014, "a language that builds on the best of C and Objective-C, without the constraints of C compatibility" [91]. Swift is a general-purpose, multi-paradigm, static-type compiled[12] programming language which adopts safe programming patterns. As of September 2016, it is currently in version 3.

[12] Swift and Objective-C must pass through a program called a compiler (Swift uses the high-performance LLVM compiler) before they execute. These contrast with **interpreted** programming languages such as Ruby, Python, and Java (i.e., instructions execute directly without the need for compilation).

One of the major differences between Objective-C and Swift (apart from the syntax) is that the former is a dynamic type language, while the latter is a static type language[13] [92]. This difference means that the Swift compiler must have information about all classes, functions and other symbols at compile time. When used with the Cocoa framework, Swift can communicate with the Objective-C runtime (using Swift syntax) which gives it access to dynamic classes, message passing, and other features from Objective-C.

Swift 3 is a major release, and as Swift 2 before it, it introduces source-breaking changes due to naming differences in syntax and Apple APIs. Apple's main goal with version 3 is to implement the last essential source changes so that Swift can turn into a mature and stable language going forward. Swift 3 (and all previous releases) do not have a stable Application Binary Interface (ABI)[14], which means apps, libraries and utilities built with older versions of Swift will not compile under Swift 3 without significant refactoring.

Swift's unstable ABI is keeping some iOS developers (and almost all large companies) from replacing their Objective-C codebase. Many issues can arise due to this instability: quickly outdated SDKs and complicated dependencies are just a couple that businesses do not want to deal with when programming at scale [93]. For example, if a company builds an SDK compiled with Swift 2, they would need to distribute a new version to support subsequent major releases of Swift until the ABI becomes stable.

In January, Olson [94] analyzed the Top 100 Free Apps and found that 89% contained no Swift code. Even the remaining percentage (11%) of apps were not guaranteed to have converted entirely to Swift. This reluctance is expected, as Swift is a new language that is still undergoing development. De Simone [95] cites Lattner (one of the authors of Swift), who stated ABI stability is the primary goal for Swift 4, slated to be announced at next year's WWDC.

Nonetheless, adopting Swift has many benefits [96]. For a small company or an independent developer with a small codebase and not many dependencies, these will likely outweigh the costs. Potential benefits include, but are not limited to, the following:

- Less verbosity;
- Supports multiple programming paradigms (e.g. object-oriented, functional, protocol-oriented);
- Faster than Objective-C for most operations (due to lack of legacy C API);
- Open-source;
- Safer language due to its static typing;
- Interactive via Swift Playgrounds (which allows for learning through the Swift Playgrounds iPad app);
- Can be used as a scripting language.

Even if developers refuse to convert their codebase to Swift, they can still use Swift alongside Objective-C code, which allows developers to implement features using Swift but still retain older code in Objective-C. This interoperability permits Swift to call Objective-C libraries

[13] This is what allows Swift to enforce type safety (i.e., if the compiler detects a variable as a `String`, it must not be of any other type).

[14] An Application Binary Interface is the interface that defines the layout of data structures, how arguments are pushed onto the stack, the way applications make system calls, and other low-level details.

and frameworks as if they were Swift code (the only requirement being the use of a bridging header[15]).

3.3.2 Backend as a Service provider

An essential part of the application's stack, a backend needs to deal with authentication, data management, push and email notifications — and that is just the bare minimum. Some Backend as a Service (BaaS) providers offer all these in one, but the backend can be made up of various interoperable components from different vendors (e.g. one to handle authentication and data management, and one solely for sending push notifications).

Up until January 28th, 2015, Parse (owned by Facebook) was the most prominent BaaS, a service that thousands of apps relied on for storing and managing data. Unfortunately, Facebook announced they are going to shut down this service in a year [97]. One positive aspect is that Facebook open-sourced the Parse project, allowing anyone to set up their backend built on Parse (with a few restrictions).

There are two major issues with Parse shutting down. First, apps that are no longer maintained but still used will "die" in a year, rendering them useless. This will cause a lot more distress down the line than at present (as developers are currently seeking alternatives), leading to the second issue: developers now need to seek a viable alternative for their product, make sure it works and deploy it to their existing user base. This is definitely not a trivial task: any mistakes will anger users and have them consider other apps.

Ojala [98], in his article on Parse, stated: "There's no inherent safety in buying from a big vendor". This certainly rings true for other BaaS in the past, with StackMob being acquired by Paypal in 2014 (and subsequently shut down), and Parse following suit after the acquisition by Facebook in 2013.

Building a backend stack from scratch is not a trivial task, and there are many other non-functional requirements to consider even after considering features: scalability and reliability are just two critical factors. We decided to seek a suitable alternative for Parse: a vendor that can provide a stable, reliable backend while also providing an SDK for iOS and adequate documentation.

For our wish list app, a backend service was needed to implement the ability to have data syncing automatically (preferably in real-time) with scalable servers in the cloud, and providing a way to access data on multiple devices. Ideally, it should be easy to deploy, and provide authentication and access controls for security and privacy.

After some initial research from a Github post on Parse alternatives [99], we narrowed it down to four vendors based on the highest level of discussion and their adequacy to our project requirements. A comparison between them is listed in table 3.1.

[15] A bridging header allows classes to be accessed by both languages. Xcode can add this automatically in a Swift project when importing Objective-C code.

[16] Included in Apple's developer membership.

[17] And 5 paid plans.

[18] With functional limitations.

[19] Syncano is free to test and build, but production environments start at this amount.

Table 3.1: Comparison between BaaS vendors

Name	Price	Features	Limitations
CloudKit	€99/year[16]	• Seamless iOS integration • Uses Core Data directly • Real-time changes	• iOS only • Not customizable
Firebase	Free[17]	• Native real-time changes • Multi-platform • Media storage • Push notifications • Offline capabilities • Extensive documentation	• Low limits for free users
Backendless	Free[18]	• Multi-platform • Push notification support • Analytics • File hosting	• Not real-time • No offline support
Syncano	$25/month[19]	• Multi-platform • Real-time changes • Extensibility	• No push notifications • No free production plan • No offline support

3.4 Frameworks

iOS apps can use frameworks to leverage the power of Swift/Objective-C APIs with the convenience of having a prepackaged library ready to use in development for a multitude of tasks (e.g. logging, networking, testing). It saves time on auxiliary tasks that can be spent in the actual development of the app (e.g. features and user interface).

3.4.1 Dependency/Package Manager

Every iOS app should use a dependency manager as it frees developers from having to import frameworks and libraries manually (among other benefits). There are two main package managers for iOS: CocoaPods and Carthage.

CocoaPods has been around since 2011. Written in Ruby, it provides a standard format for handling external libraries, stored in a Podfile. A Podfile contains a list of all project dependencies. As an example, the following Podfile (listing 3.1)[20] installs the CocoaLumberjack (logging) and Realm (local database) frameworks in Swift:

```
platform :ios
pod 'CocoaLumberjack/Swift'
pod 'RealmSwift'
use_frameworks!
```

Listing 3.1: Podfile example

[20] `use_frameworks!` integrates CocoaPods into a project using frameworks instead of standard libraries.

Carthage was created as an alternative to CocoaPods. The main difference is that the former is a decentralized dependency manager, as there is no central list of projects (CocoaPods' website allows developers to search for libraries).

This approach makes project discoverability more difficult when using Carthage; however, it leaves automatic framework integration up to the user (Carthage simply compiles binaries, while CocoaPods compiles and integrates them into a Xcode workspace), making Cartage the more flexible option.

Cartage uses a Cartfile, which is similar in structure and syntax to a Podfile. Listing 3.2 shows an example of a Cartfile that integrates the ReactiveCocoa and Mantle libraries.

```
# Require version 2.3.1 or later
github "ReactiveCocoa/ReactiveCocoa" >= 2.3.1

# Require version 1.x
github "Mantle/Mantle" ~> 1.0  # (1.0 or later, but less than 2.0)
```

Listing 3.2: Cartfile example

3.5 Summary

To understand guideline conformity, we first gave an overview of Apple's HIG and the three key iOS design principles: deference, clarity, and depth. Then, we presented a list of metrics used in the analysis of ten popular apps.

We discovered that the Facebook app was the worst battery offender, with unresolved issues dating back to 2013. Reports surfaced in 2015 accusing Facebook to have abused iOS's background timeout to keep their app active at all times, which in turn killed battery life.

The rest of this chapter focused on detailing the most popular third-party technologies, frameworks, and utilities. In this first milestone, we focused on giving information on Parse's closure and a comparison between the alternatives that remain.

We also covered some third-party dependency managers (namely CocoaPods and Carthage) which can be used to simplify the integration of third-party libraries and frameworks with our project.

Chapter 4

App Requirements & Design

In this chapter, we introduce the thought process behind the name of our wish list management application, followed by a detailed list of its functional and non-functional requirements.

We then discuss the app's architectural design, include an overview of the MVC design pattern applied to iOS apps, and present its data model using Xcode's object graph.

Additionally, a section dedicated to the Swift API design guidelines and details to take into account when designing iOS apps is also available in this chapter.

4.1 App Name

One of the first steps we took in the design process was figuring out a name for our application. We wanted a name that would convey the app's fundamental use case — taking a photo of an object that one desires — and pair it with the concept of organizing said objects (that we call "items") into wish lists. Thus, we combined the words *snapshot* (i.e., of a photo) and *wish*:

snap	+	*wish*	=	**snapwish**
(taking a photo)		(wish lists)		(photos [of items] in wish lists)

4.2 Requirements

This section details the functional (i.e., features) and non-functional requirements of our wish list management application named Snapwish.

4.2.1 Functional

The design process started with the analysis of the requisites mapped to a use case (UC) diagram[1]. This procedure was crucial in understanding which features to implement.

Functional requirements answer one key question: "What does this app do?". With Snapwish, a user can perform the following actions (use cases):

[1] A use case diagram represents a user's interaction with the system and the specifications of a use case. In simpler terms, it focuses on what the application does, by listing the ways different users (named actors) can interact with the system.

UC-1: Create, update, or delete an item;

UC-2: Create, update, or delete a wish list;

UC-3: Add or remove items from a wish list;

UC-4: Share wish lists with other users of the app;

UC-5: Share a specific item on social networks (Facebook & Twitter);

UC-6: Login to the application using existing Facebook/Twitter credentials;

UC-7: Login to the application with an existing in-app account;

UC-8: Register a new account;

UC-9: Reset password (if current one was forgotten).

These actions have been translated into the UC diagram shown in figure 4.1.

4.2.2 Non-functional

Regarding non-functional requirements, it is expected that our app (and its backend components) fulfill these important aspects (ordered by level of highest importance):

1. **Reliability**: the backend must be available at least 99.9% of the time, so the provider we choose must be trusted by the industry;
2. **Resilience**: in the event the user does not have access to an Internet connection (or the server is down), our app must maintain an acceptable level of operation (such as allowing a user to create items and add them to existing wish lists). Changes will then sync as soon as a connection is established and, in the case of cellular data, only if the user has allowed the app to use that data;
3. **Platform compatibility**: our app supports iOS 9.3 or higher on all twenty devices that can run it (see Section 2.3.1);
4. **Response time**: real-time sync would be ideal so our app can react and display changes with very minimal delay:
5. **Usability**: our app's user interface and flow need to be straightforward and efficient, so users do not get confused when performing tasks;
6. **Security**: we must guarantee that users do not have more privileges than specified, and that the backend provider offers authentication mechanisms which can be abstracted from the client;
7. **Privacy**: the data collected from our users (email addresses, passwords, wish lists, friends) have to be stored securely and privately, with the assurance that no one can decrypt sensitive information and abiding all European Union (EU) data privacy laws;
8. **Scalability**: we need a backend provider that can scale appropriately according to the number of users, data storage, and bandwidth;
9. **Price**: our app will either be free with in-app purchases or paid to try and recoup the developer membership investment and provide adequate support.

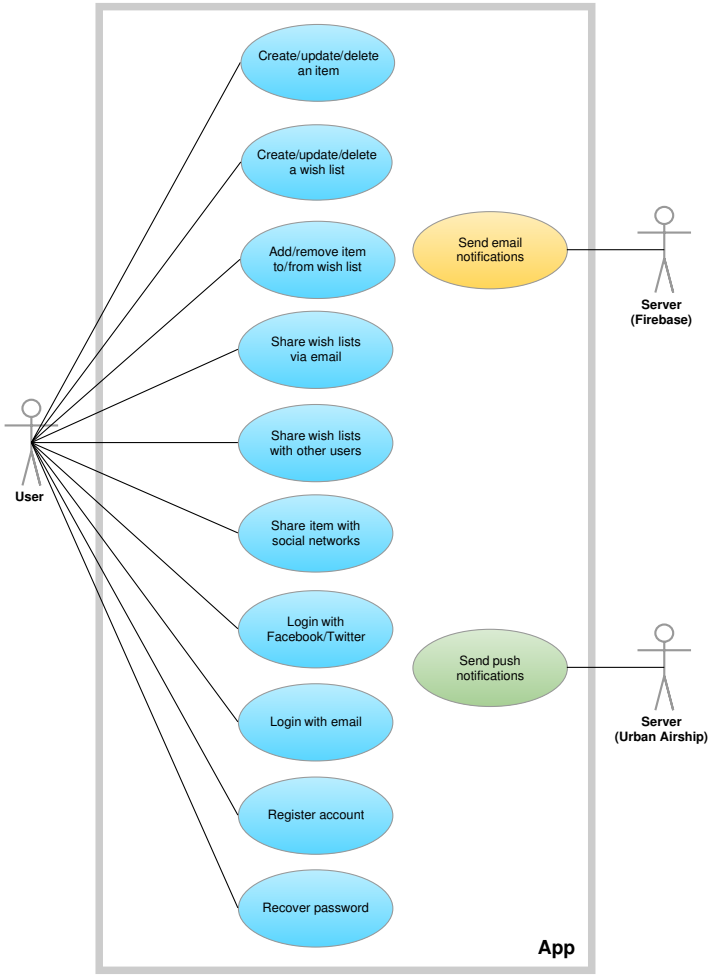


Figure 4.1: Snapwish's use case diagram

4.3 Architectural design

This app's architecture adopts the **client-server** model.

In Chapter 3, we analyzed four backend providers: CloudKit (Apple), Backendless, Syncano, and Firebase (Google). All four of these providers were good choices for an iOS app, with some offering more features than others (e.g., push notifications) yet lacking in other desirable aspects (e.g., real-time support, offline sync). Offline sync was a feature we needed, so we had to eliminate Backendless and Syncano from our choices early. That left CloudKit and Firebase as contenders. Both had real-time sync and offline cache support, but CloudKit came with one major snag: the upfront cost. We didn't want to pay Apple's developer fee until we were ready to ship, as the app's development process would take valuable time off the membership. Thus, we went with **Firebase** as the backend provider.

4.3.1 Deployment diagram

The user only interacts with one piece of software — our wish list app — that can be installed on a variety of iDevices. It communicates with Firebase's backend servers for authentication

and real-time syncing (Firebase Real-time Database), and uploading of media (Firebase Storage) via a REST API. Password resets are sent by Firebase's servers to the Mail app (or equivalent alternative) on a user's device.

Sharing is also a feature of the app. The user can share items with social networks, such as Facebook and Twitter. These use HTTPS[2], a widely used protocol for communication throughout the Internet for data transmission.

Figure 4.2 shows the various components of this project in a deployment diagram[3].

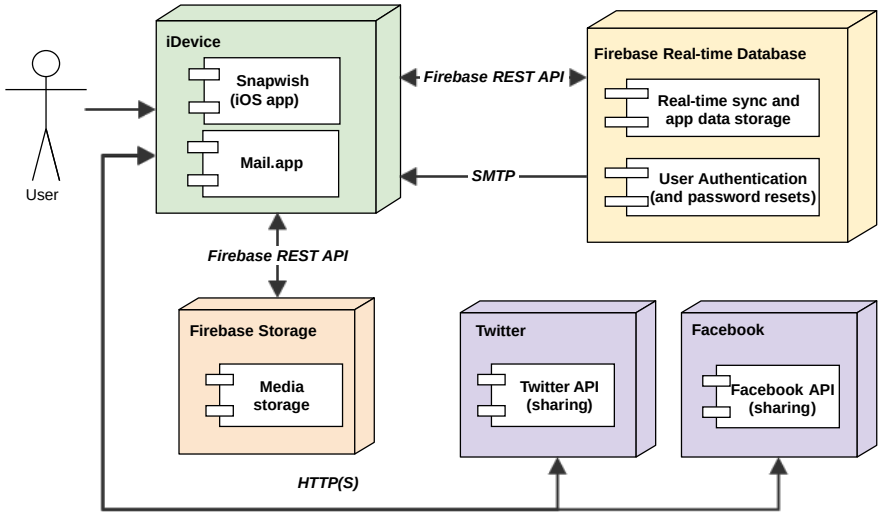


Figure 4.2: Deployment diagram for a wish list management application

4.4 Design patterns

Model-view-controller (MVC) is the fundamental architectural pattern in iOS development. It assigns an app's objects one of three roles: model, view, or controller. From Apple's documentation on iOS MVC: "The pattern defines not only the roles objects play in the application, it defines the way objects communicate with each other." [100].

4.4.1 Model-View-Controller

To exemplify how MVC works in an iOS app, let's take our project, a wish list management app. The **model** is what the application is (but not how it is displayed). It contains no information about the number of wish lists displayed on screen or how they are displayed. This is the **controller**'s job — interpreting model information for the view, i.e., presenting the model to the user (UI logic). Our model would be the wish lists, the items contained inside them, the users they belong to, and friends they've shared them with (among others).

[2]This is because the app needs to be 100% ATS-compliant.

[3]A deployment diagram shows the architectural view of the entire system. Essentially, it displays the hardware of the system, which software is running on each piece of hardware (e.g., servers, mobile devices), and communication the communication between systems.

The **views** are the controller's "minions" and serve one purpose: to display data on screen via the controller's logic. Views should be as generic as possible, i.e., buttons and sliders know nothing about wish lists. They are *reusable components* and do not own the data they display (if needed, they have a protocol[4] to acquire data).

A controller can always talk directly to its model, and to its views using *outlets*[5], as it is the controller's task to bridge communication between what the application is (model) and these generic views. On the other hand, the view and the model should **never** talk to each other directly, because the former is interface independent (one could build both a command-line interface (CLI) and a graphical user interface (GUI) using the same model).

Views can talk to a controller using *target-action*. When an action occurs in one of the controller's views, the view fires an action at the previously set target (owned by the controller). Examples include a button being pressed and a slider being touched. This communication is blind because the view does not know which class it's sending the action to.

Sometimes the view needs to synchronize with the controller; for that, *delegation*[6] is used. The controller sets itself as the view's delegate and gets notified of certain actions, allowing it to intervene (e.g., refreshing a table view when new model data is updated).

The model can also notify a controller of changes. There are two ways to accomplish this: notifications, and Key-Value Observing (KVO). Both methods use the concept of "radio station broadcasting". The controller signs up for changes it wants to be notified about. When a change occurs in the model, the model broadcasts the change and anyone that is "tuned in" to receive those alerts gets notified. This mechanism can also work between a view and a controller, regarding changes that occur in a view which the controller wants to be informed about.

A representation of MVC and its communication mechanisms is depicted in figure 4.3 [101].

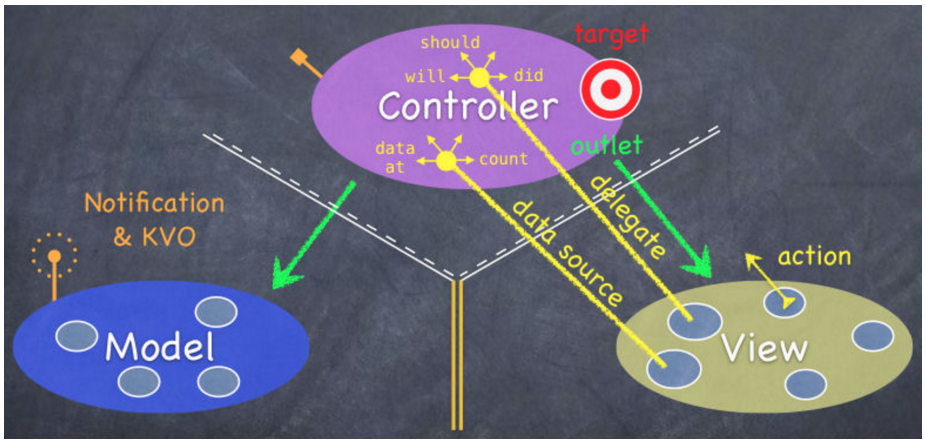


Figure 4.3: Communications in the Model-view-controller pattern

[4] This uses the concept of delegation between a controller and a view. For example, if a table view wants to know how many wish lists exist in our model, it has a delegate method that allows the controller to tell the view how many there are at a given instance.

[5] Outlets are simply properties in a controller that it uses to talk to its views.

[6] Generally, delegate methods are prefixed with `should`, `will` or `did`, handling an action which should occur, will occur or did occur (respectively).

4.5 Database model

As part of the MVC pattern, the model specifies the entities and relationships of a particular representation of data. As we are using Firebase as our backend provider, there is no need for Core Data. Firebase deals with the data storage, caching, and offline usage scenario.

In Firebase, "database data is stored as JSON [JavaScript Object Notation] objects. There are no tables or records." [102]. However, for the sake of simplicity, figure 4.4 shows an object graph of our three entities and their relationships[7].

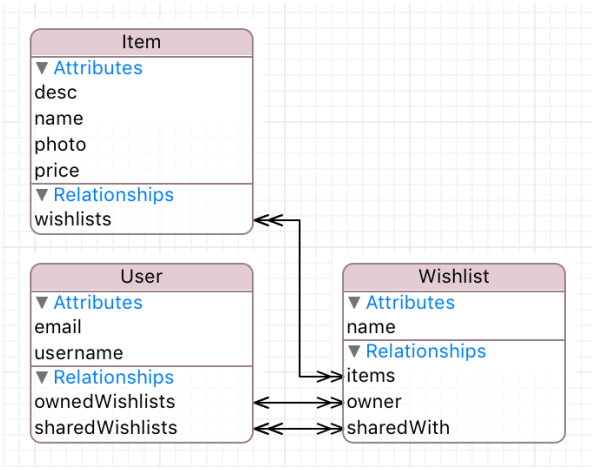


Figure 4.4: Object graph for a wish list management app

The `Wishlist` entity stores the name of a wish list in the attribute `name`. It has three relationships: a bilateral *many-to-many* relationship with the `Item` entity with name `items`[8]; another bilateral *many-to-many* relationship with `User` with the name `sharedWishlists`[9], and a *one-to-many* relationship from `User` to `Wishlist` with the name `ownedRelationships` (a wish list belongs to only one user, but they can own multiple wish lists).

An instance of the Item entity has several attributes: its `name`, a `description` (optional), a `photo` of the item (an image taken with the camera or retrieved from a user's photo library), and `price` (optional). It has a *many-to-many* relationship with the `Wishlist` entity with name `wishlists` (the wish lists it belongs to).

There is also a User entity, which stores user data[10] (`email` and `username`), and has two relationships (described above). The password field is never cached locally for security reasons.

[7] Since this is an object graph, all database-specific implementations (e.g., primary keys) have been abstracted.

[8] An item can be in zero or more wish lists, and those wish lists can also have zero or more items in them.

[9] A wish list can be shared with many users; consequently, those users can also be invited to many wish lists.

[10] The user provides this information during registration. However, if logging in via Twitter/Facebook, the details will be supplied by the social network itself after user authorization.

4.6 Design guidelines & good practices

The world-renowned chef Marco Pierre White once said: "Perfection is lots of little things done well" [103]. In cuisine, this certainly is true, and it also can be applied to many other fields including computer science.

This section covers guidelines and details developers and designers should consider to enhance their workflow and positively impact user experience in development and production environments.

4.6.1 Swift API Design Guidelines

In addition to Human Interface Guidelines specific to each platform (iOS, OS X, watchOS, and tvOS), Apple also offers API design guidelines for their programming language, Swift. While not strictly enforced, Apple strongly recommends developers adhere to the guidelines to write clear and concise Swift code which can be easily understood by others.

To aid this process, the Swift website contains a list of all the API design guidelines complete with explanations [104]. A WWDC video, "Swift API Design Guidelines" [105], covers this subject in-depth, presents Apple's philosophy on the matter, and exemplifies how developers can apply the guidelines to their APIs.

From the Swift API guidelines, three core tenants we consider every developer should use are as follows:

- Clarity at the point of use;
- Strive for fluent usage;
- Write documentation for every declaration.

Clarity at the point of use

Methods and properties are declared once but used repeatedly, so their declaration should be clear and concise. One should always consider a use case to ensure a declaration fits its context.

It is always preferable to write an easy to understand API than one that limits itself to the least amount of characters. For example, consider an extension method[11] on the class `List` that removes an element in a collection at a given position (listing 4.1).

```
extension List {
  public mutating func remove(at position: Index) -> Element
}
users.remove(at: x)
```

Listing 4.1: Extension on List which removes an element in a collection

[11] In Swift, an extension is a way to add new functionality to an existing class, structure, enumeration, or protocol type. This includes the ability to extend types that one does not have the source code for (e.g., Apple frameworks and libraries). Extensions are akin to categories in Objective-C.

The keyword `at` in the method's signature provides context to which position is being removed. If it were omitted, the reader may imply that the method searches for and removes an element equal to `x`, instead of removing the element at position `x`.

Strive for fluent usage

This tenant states that method and function names should form grammatical English phrases. One should be able to understand what a method does and the role of each parameter by reading the method's signature. Listing 4.2 shows some examples of good API design (taken from Swift.org):

```
x.insert(y, at: z)          // "x, insert y at z"
x.subViews(havingColor: y)  // "x's subviews having color y"
x.capitalizingNouns()       // "x, capitalizing nouns"
```

Listing 4.2: Examples of good Swift API design using fluent English phrasing

Conversely, this would be considered bad design (listing 4.3):

```
x.insert(y, position: z)  // "x, insert y position z", which isn't fluent
x.subViews(color: y)      // what does "color" mean in this method call?
x.nounCapitalize()        // doesn't read well
```

Listing 4.3: Examples of bad Swift API design

Write documentation for every declaration

Documentation written at the declaration point gives target users helpful insight on an entity's role in an API. Each comment should summarize what the entity *does* and what it *returns*, omitting null effects and edge cases (listing 4.4). Documentation should be written using Swift's dialect of Markdown.

```
/// Encodes a UIImage to the WebP format and returns its encoded data.
func encodeToWebPData(completionHandler: (NSData?) -> ())
```

Listing 4.4: Entity documentation at the declaration point

If the entity is an initializer, it should describe what it creates (listing 4.5).

```
/// Creates an instance containing "n" repetitions of "x".
init(count n: Int, repeatedElement x: Element)
```

Listing 4.5: An initializer documented according to Swift API guidelines

The summary is sometimes not sufficient to comprehend the use case. In this case, further explanations may be included, but they should be separated from the summary by a blank line (see listing 4.6 for an example).

```swift
/// Resizes and returns an image as close as possible to the specified maximum
  dimensions.
///
/// This method does not resize (upscale) images when the provided dimensions are
  larger than the image's dimensions.
func imageScaledToMaxWidth(_ maxWidth: CGFloat, maxHeight: CGFloat) -> UIImage?
```

Listing 4.6: A declaration containing a summary and explanations

In Xcode, after documenting the example above, the documentation appears as shown in figure 4.5.

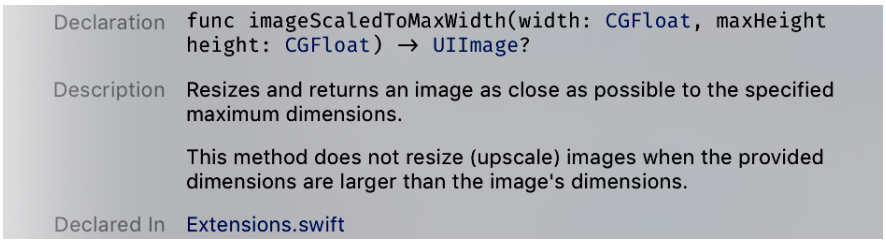


Figure 4.5: imageScaledToMaxWidth:maxHeight: method documentation (in Xcode)

4.6.2 The "little" big details

The success of an app lies intrinsically in the user experience it provides [106]. From a mobile UX standpoint, minor details often are what separate a good app from a great app. However, it is easy to forego design elements that are considered unnecessary at development time due to tight schedules, or lack of designers who carefully consider these seemingly minor but often crucial elements. Below, we present some details and show why they are as important as any app's biggest features.

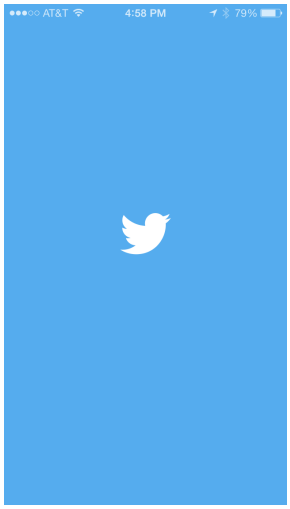
In addition to the details exemplified herein, Apple also provides a list of design "do's and don'ts" on their developer website which should be followed to avoid having an app rejected from the App Store [107].

Splash screen

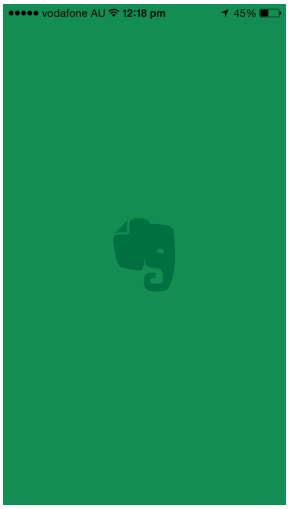
A splash screen is a view that typically consists of the app or company's branding and appears while an application is launching (figure 4.6 shows examples). Apple recommends using a splash screen (referred to as *loading screen*) to enhance the user experience by simulating faster loading times. If an app takes more than ten seconds to load, one should consider an animated splash screen that is both eye-catching and distracts the user long enough not to notice the delay.

Empty states

When an app is in the development stage, developers (and designers) tend to focus on a populated user interface where all aspects of the app's layout are on-screen. However, there



(a) Twitter splash screen



(b) Evernote splash screen

Figure 4.6: Examples of splash screens

will be moments where the app does not have data to display, or in the event of a user or system error. Those *empty state* moments are equally important because users expect an app to guide them — especially on first use.

The purpose of an empty state is not only to provide meaningful actions but to serve as a way of user onboarding (i.e., they introduce the app's main use case(s) and demonstrate what they do) or error recovery by displaying a friendly screen when an unexpected event occurs.

When designing an empty state for a first-time user, consider focused experiences which indicate the primary goal's intent. The design should be simple, providing a clear message and an actionable button (as displayed in figure 4.7 [108]).

Regarding an empty state to handle user or system errors, it "must find a balance between helpfulness and friendliness": explaining how to recover from the problem — regardless of the user's culpability — and using humor (if possible) to mitigate the frustration of an error [106]. Figure 4.8 [109] below shows how Azendoo (a task management app) manages network errors.

Animated feedback

Babich [106] writes: "Good interactive design provides feedback". Regardless of the type of design (e.g., skeuomorphic or flat), the use of animations provides a level of visual feedback that resembles interactions with objects in the physical world.

Animations must be fast to survive long-term use and instantly convey information to the user, yet avoid being boring or distracting. For example, since transition animations between view controllers occur frequently, they must execute quickly and place all focus on the content of the destination view controller instead of the animation itself.

Another example is any animated button that triggers a network request (e.g., liking a post). Feedback should be immediate, so the user knows the app registered the action (e.g., a tap or long press). The network request will likely take a few seconds to complete, so even

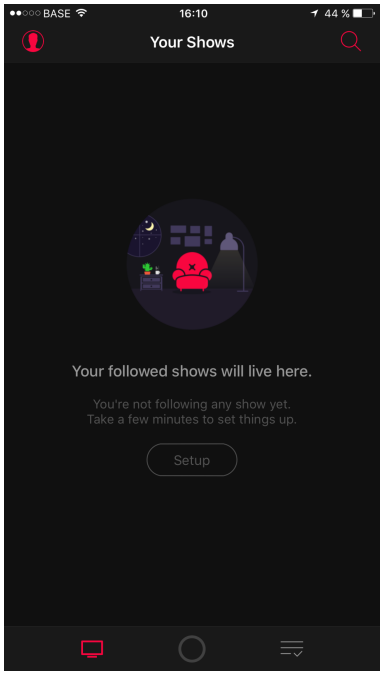


Figure 4.7: First-time empty state in the Serist app (when no TV shows have been followed)

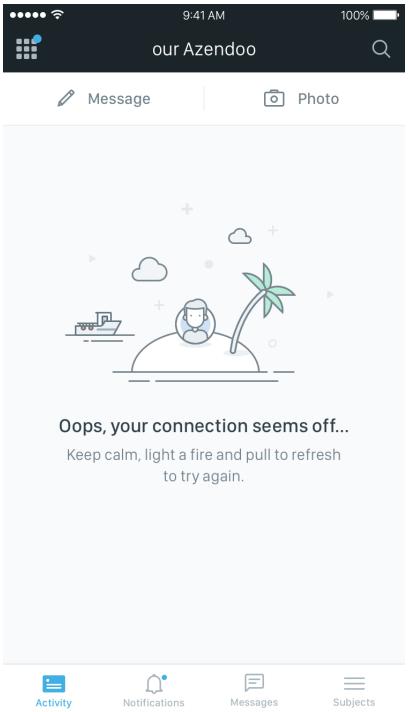


Figure 4.8: An empty error state in Azendoo's iOS app

before the result of the action is reflected on the server, the app allows the user to perform other actions (e.g., liking other posts or switching to another app).

4.7 Summary

Design aspects of our wish list management app were the main focus of this chapter.

We started by defining the functional and non-functional requirements. The former resulted in ten use primary cases (which can expand throughout the development lifecycle and feedback). The latter focused on making sure reliability, resilience, and platform compatibility were at the top of our priorities. This emphasis is due to the desire of developing an app that can work both online and offline, gracefully syncs changes automatically, and can adapt to any device's screen size.

An overview of the main iOS pattern — Model-view-controller — was also presented to explain the motives behind the division of app objects into three groups and the communications established between them (and how this communication is done).

We also included a basic database model which will be the starting point for our backend. However, we have not yet translated the object graph to JSON objects. To avoid complexity and anti-patterns, we must consider denormalizing[12] the data when creating the data model on Firebase.

The chapter rounded off with an overview of the Swift API guidelines and how developers can benefit by adhering to core principles. We also highlighted three app design considerations which can be considered small details but are crucial to the user experience: splash screens, empty states, and animated feedback.

[12] The process of attempting to optimize a database's read performance by adding redundant data or by grouping data.

Chapter 5

App Implementation

This chapter covers all aspects related to the implementation of the wish list management application, Snapwish. We begin by listing the main tools used when developing the app, as well as explaining the structure of the app's data model and how the chosen backend provider stores data (including its permission model).

For each view controller, we detail the main use cases and discuss their implementation (with relevant screenshots for illustration purposes).

We also justify the reasons behind the use of every technology, framework or library (referred in Chapter 3) in our application if an alternative exists.

5.1 Tools

For Snapwish's development, we decided to employ tools Apple recommends and provides free of cost for developers. The Integrated Developer Environment (IDE) used was Xcode 7.3.1, the version released to support iOS 9.3. With iOS 10's public release, we switched to Xcode 8 which supports the latest version of iOS. Xcode's companion app Instruments was pivotal in profiling our app (as seen in the next chapter).

We used Swift for development, namely version 2.3. Released to the public on September 13th, 2016, Swift 3 limited our ability to upgrade because some of our dependencies (using CocoaPods) lacked support for this version.

Apple is encouraging developers to learn Swift to broaden their skill set. We had previous experience in both Objective-C and Swift), but Swift's benefits (mentioned in Chapter 3) ultimately made it our development language of choice.

In regards to testing during development, we used a combination of the simulator and real test devices including, but not limited to, an iPhone 6 Plus (iOS 9.3.3), a 5th generation iPod touch (iOS 9.3.2), and a 4th generation iPad (iOS 10.0.1).

5.2 Data model

In the MVC architecture, the model is the component directly bound to the data, logic, and rules of the application. Contrary to the controller and its view(s), the model is interface-independent and responsible for what the application is rather that what it *does*[1].

[1] That is the responsibility of the controller.

When developing for iOS, there are many ways to represent an application's data model. If developers are using Core Data (Apple's object graph and persistence framework), Xcode will automatically generate a data model used for modeling entities and relationships (no class declarations required). On the other hand, if one chooses to use a third-party framework for storing data (e.g., Realm, a mobile database), then modeling data will usually pass through classes or structs.

5.2.1 Structs over classes

As we are not using Core Data for Snapwish's data model, we decided to use structs instead of classes to model its entities. Swift provides some features that make structs a better choice in many circumstances, and Apple encourages their use [110].

Structs are **value types**, where each instance keeps a unique copy of its data — a safer mechanism than having multiple references to the same instance (as happens with classes). This mechanism is important because we will be performing operations that take an instance of an entity, modifying it slightly (by updating a field), and saving it back to the database. If we hold two references to the same object, and we modify the first but not the second, structs guarantee that only the first object is modified (listing 5.1), whereas if we were using classes the object to which we point by reference to would get modified irrespective of the number of references we hold, as exemplified by listing 5.2.

```
struct Temperature { var temp: Int = 20 }  // in celsius
var bedroom = Temperature()
var kitchen = bedroom                      // bedroom's temp is copied to kitchen
bedroom.temp = 25                          // changes bedroom's temp only
print("\(bedroom.temp), \(kitchen.temp)")  // prints "25, 20"
```

Listing 5.1: Controlling temperature using a struct

```
class Temperature { var temp: Int = 20 }   // in celsius
var bedroom = Temperature()
var kitchen = bedroom                      // bedroom's temp is copied to kitchen
bedroom.temp = 25                          // changes the temperature referred to by
  bedroom (and kitchen)
print("\(bedroom.temp), \(kitchen.temp)")  // prints "25, 25"
```

Listing 5.2: Controlling temperature using a class

Due to the nature of unique copies, developers using structs (and other value types such as `enum`) do not need to worry about memory leaks or multiple threads racing to access or change the data of an instance. Inherent thread safety is crucial in multi-threaded environments where concurrent data modification can lead to bugs that are very hard to debug. Thus, it is not a surprise that the two main collection types in Swift are both value types: `Array` and `Dictionary`. Apple chose this implementation in the "spirit of improving [thread] safety" and to help developers "write more predictable code in Swift" [111].

5.2.2 Firebase Real-time Database

Built as an inherently social app, Snapwish's value proposition (see chapter 2) pitches the ability to create and share wish lists. Any app that needs to store and access data frequently

requires a backend which provides excellent reliability (i.e., uptime) coupled with scalability (accounting for an increase in users and stored data).

Users do not like to be kept waiting, so fast network read and write times are a necessity. Additionally, an app that refreshes data without user input and gracefully handles offline state places almost no burden on users having to verify whether it is performing as they would expect (regardless of network conditions).

Firebase's real-time, "non SQL" (NoSQL)[2], cloud-hosted database is an option to consider when developing an iOS app[3] because it offers three key capabilities that directly address the requirements listed above. Also, its API is designed to only allow operations that can be executed quickly for the sake of responsiveness.

From Google's Firebase website [112], these capabilities are as follows:

- **Real-time**: instead of using HTTP requests, Firebase uses data synchronization to receive data change updates across all connected devices within milliseconds;
- **Offline**: because data is cached locally, a loss of network connectivity does not shut down the app's capabilities. When the connection is reestablished, the Firebase SDK syncs to the cloud database automatically;
- **Accessible from client devices**: any client device or mobile browser can access the Firebase database directly without the need for an application server.

As mentioned during the creation of Snapwish's object graph (Chapter 4), the Firebase database does not use tables or records to store data. Instead, all data is stored as JSON.

Structuring Data

The process of building an adequately structured database is not trivial. Apart from planning based on the type of database (e.g., NoSQL or relational), one needs to place greater consideration on the complexity of the application's data model based on the number of existing entities and their relationships.

Fortunately, Firebase abstracts the underlying complexity away from the application developer by structuring data as a JSON tree. When data is added to the JSON tree, it becomes a node in the existing JSON structure with an associated key (similar to a primary key[4]). One may specify custom keys or let Firebase generate them automatically [113].

For example, consider our wish list management application which allows users to store a profile and their list of friends. For simplicity, each user is stored under the path `"/users/<uid>"`, where `uid` corresponds to their username (which is unique). The user `swift` could have a database JSON tree which looks similar to listing 5.3.

[2] This is a database that uses a different mechanism for storage and retrieval of data than those employed in traditional relational databases (i.e., tables and records).

[3] The Firebase real-time database can be deployed to apps written in Swift and Objective-C using CocoaPods as the dependency manager.

[4] In an SQL database, a primary key is a special, non-null field that uniquely identifies all table records. For example, an identification number (ID) is automatically generated by the database.

```
{
  "users": {
    "swift": {
      "name": "Taylor Swift",
      // Index Taylor's friends in her profile
      "friends": {
        // the value here doesn't matter, just that the key exists
        "timcook": true
      },
    },
    "timcook": {
      ...
      "friends": { "taylorswift": true },
     },
    "pschiller": { ... }
  }
}
```

Listing 5.3: Example of a user profile model stored in the Firebase database

Here, a user's friends would be listed as `<uid>:true` under `"/users/<uid>/friends"`. As a friendship is mutual (i.e., two-way), the information representing it can be found in two locations in our JSON tree. This data duplication is a basic yet effective way to index many-to-many[5] database relationships using Firebase.

For more complex cases, denormalization[6] is a necessity when a relationship exists between two *distinct* entities instead of between the same entity (which is the case with friends — both parts of the relationship are users).

Instead of nesting data to create multiple sub-nodes, entities own their root path. For example, we could nest a user's wish lists under their `users` path. This would, however, be ill-advised since wish lists represent a separate entity of our model and their only connection to a user is via ownership in a many-to-one[7] relationship. Thus, the data would be stored in Firebase as shown in listing 5.4.

```
{
  "users": {
    "swift": {
      "name": "Taylor Swift",
      // Index Taylor's wish lists
      "wishlists": {
        "gadgets": true,
        "fashion": true,
        "christmas": true
      }
    ...
  },
  "wishlists": {
    "gadgets": {
      "name": "Tech Stuff",
      "owner": "swift"
    }
  }
}
```

Listing 5.4: Denormalization applied to user and wish list entities

[5] This means that a user may have zero or more friends, and their friends (who are also users of the application) may also have zero or more friends.

[6] This is the process of splitting data into separate paths.

[7] Put simply, users can create and own multiple wish lists, but a wish list has a single owner.

As mentioned before, this does duplicate some data due to the denormalization process. The wish list `gadgets` is listed under Taylor's profile, and this list shows her as the owner. So to effectively delete or transfer ownership of the wish list, two operations must be performed (one to update the relevant data under the `wishlist` path, and the other to remove it from Taylor's owned lists). Even when the database scales into the millions, knowing which wish lists Taylor owns is easier because we only have to look up her user data.

Saving Data

To save data to the Firebase database, we used two methods that Firebase provides (there are more but for our app these sufficed): `setValue` and `updateChildValues` [114].

The method `setValue` saves data at the given path in the database, **replacing** any data that may exist. For example, writing data at the path `"users/<user-id>/username"` will either create a new user or overwrite the username for an existing user. For instance, we can add a new user as such (listing 5.5):

```
// Create a reference to our users node
let usersRef = FIRDatabase.database().reference().child("users")
usersRef.child(user.uid).setValue(["username": username])
```

Listing 5.5: Creating a new user using setValue

If one needs to write to specific children without overwriting other child nodes, the solution is to use `updateChildValues`. This method can also write to lower-level child values. It takes a dictionary where the keys are the children to be updated (relative to the parent path), and the values represent the data to be written. Listing 5.6 showcases a way to update only some of the fields of an item while keeping the others intact (assume an item has a name, description, price, and a small and large image under `"items/<item-id>/images"`).

```
let itemUpdates = ["name":         name,
                   "description":  description,
                   "images/small": smallImgURL]
item.ref.updateChildValues(itemUpdates)
```

Listing 5.6: Updating specific fields of an item using updateChildValues

To delete data, one can call `removeValue` at the location of the data, or use `setValue` with a `nil` value.

Retrieving Data

Firebase data is fetched by attaching an asynchronous listener to a `FIRDatabase`[8] reference. The listener is triggered once for the initial state of the data and again every time the data changes at that reference [115].

[8] This is a class in the Firebase SDK that represents a Firebase database.

Listeners can be activated almost anywhere in the code of an iOS app, but they typically are set in a view controller's lifecycle methods: `viewDidLoad`[9] or `view[Will|Did]Appear`[10]. This is common practice because most view controllers need to access the model as soon as they are loaded into memory.

In almost all cases, we opt to attach a listener in `viewWillAppear` instead of `viewDidLoad`, and detach it in `viewDidDisappear`[11], which East [116] recommends. Apps should be good citizens of battery life, memory, and data usage. If a listener is attached once but never removed, it will waste valuable resources when that part of the app is not on-screen by fetching data and updating its views. Listings 5.7 and 5.8 show how to attach and remove listeners from the recommended lifecycle methods.

```
override func viewWillAppear(animated: Bool) {
  let refHandle = itemRef.observe(FIRDataEventType.value, with: { (snapshot) in
    let itemDict = snapshot.value as! [String : AnyObject]
    // do something with the item object here
  })
}
```

Listing 5.7: Adding a listener to observe changes at a given path

```
override func viewDidDisappear(animated: Bool) {
  itemRef.removeObserverWithHandle(refHandle)
}
```

Listing 5.8: Removing a Firebase listener

Security & Rules

To help developers define the structure of their data and protect it from unauthorized access, the Firebase real-time database provides an expression-based rules language (similar in syntax to JavaScript). Rules can be configured from Firebase's website using the provided console [117].

There are three types of security rules: `.read`, `.write`, and `.validate`. The first two they define if and when data is allowed to be read or written at a given node, respectively. `.validate` defines what a correctly formatted value must be, its data type, and other attributes. For example, an `age` field can be validated to only accept storing integers which range from 0 to 130.

An important thing to note with read and write rules is that they work from the top-down[12], meaning shallower rules override deeper rules. If a rule grants read or write permissions at a given path, no rule below it can revoke access.

To understand the structure and how these rules work, listing 5.9 showcases Snapwish's rules. Comments have been added for clarity.

[9] This method is called when the view controller finishes loading but has yet to appear on-screen. It is called only once in the view controller's lifecycle.

[10] Triggered after `viewDidLoad`, they allow the programmer to do last minute setup before or after the view is visible, and are called every time the view is displayed

[11] Also part of a view controller's lifecycle methods, `viewDidDisappear` is called when the view has completely disappeared from the screen.

[12] This does not apply to validation rules.

```
{
  "rules": {
      "items": {
        // Index server-side for faster fetching
        ".indexOn": "addedByUser",
        ".read": "auth !== null", // only let logged in users read items
        "$item": {
            // only let users create or modify items they added
            ".write": "auth.uid === newData.child('addedByUser').val() || auth.uid ===
   data.child('addedByUser').val()"
          }
      },

      "wishlists": {
        ".indexOn": "owner",
        // A user must be authenticated to write to the wishlists node
        ".read": "auth !== null",
        ".write": "auth !== null",
        "sharedWith": {
          "$uid": {
            // Index on the user IDs a wishlist is shared with
            ".indexOn": ".value"
          }
        }
      },

      "users": {
        // Anyone can read our users node (for authentication purposes)
        ".read": true,
        ".indexOn": "username",
        "$uid": {
        // only the user themselves can change their profile data
        ".write": "auth.uid === $uid",
      }
    },
  }
}
```

Listing 5.9: Snapwish's Firebase rules

5.2.3 Immutable models & data consistency

As an application developer, an aspect to consider when creating data models is immutability, which means that models cannot be modified after initialization. Changes are only possible by instantiating a new copy of the data with the modified values.

Updating our data model is only performed using the Firebase methods `setValue` and `updateChildValues`. A three-step process then takes place:

1. Changes to the data are saved to the Firebase database (if there is no network connectivity, it is stored to the local cache and will automatically sync when back online);
2. The save triggers any relevant active Firebase observer with a snapshot of the data at that location;
3. This observer updates local model variables with the contents of the snapshot (e.g., used to display data in a table view) by creating new model objects.

Immutable models may seem like an inconvenience from a functional standpoint, but the main benefit lies in the lack of shared state. If two threads (A and B, let's say) reference

an object C, any modification to C which one of the threads does not expect may cause serious data consistency issues. Immutability eliminates this issue because threads access the model in read-only mode. Any write operation results in a new thread-local object [118]. Snapwish's data model is immutable for this reason.

5.2.4 Structure

Each entity in our model is contained in a Swift file with its name (e.g., `User.swift`). Inside, the file follows a similar structure across each entity containing two structs, one representing the entity itself and the other a list of property names (i.e., fields) and their relative storage paths in the Firebase database. The entity struct includes four elements:

- All the properties that make up that entity (common properties are the key that uniquely identifies an instance of that entity — generated by Firebase — and the Firebase reference of that instance);
- An initializer from a Firebase snapshot (mandatory);
- A convenience initializer with all the entity's properties as arguments (optional);
- An instance method that converts an object of the entity's type to JSON (to then save in Firebase).

5.3 View controllers

A view controller is a fundamental iOS component, and the foundation of every app depends on one of more view controllers. Each view controller controls a portion of an app's user interface and is responsible for the interactions between that interface and the data model. View controllers can also ease transitions between user interface elements that are visually or functionally distinct [119].

Snapwish's view controllers are divided into four sections and described thoroughly in subsections that follow. While these are not isolated, they fulfill one or more use cases which can be logically separated. These sections are the following:

- User authentication (signing up and logging in);
- Items (showing and managing a user's items);
- Wish Lists (owned and shared);
- User profile (displaying and editing user information).

For reference, the name of each view controller is accompanied by its abbreviated type. A suffix of **TVC** symbolizes a table view controller, **CVC** a collection view controller[13] controller, and **VC** is a view controller without a table or collection view.

13 14

5.3.1 User authentication

Snapwish uses Firebase Authentication (Auth) to register and authenticate users. Knowing a user's identity allows Snapwish to save user data in the Firebase database securely [121]. Coupled with authorization methods like the Firebase Database rules (already covered), we can ensure that users' data is both secure and private.

Firebase Auth supports authentication using passwords, popular identity providers such as Facebook and Twitter using OAuth 2.0 (an industry standard authorization framework), and even custom authentication systems. Upon creation, a unique ID (UID) is assigned to each user and stored in the project's user database[15]. If the application supports this feature, users can tie additional login methods to the same account[16]; however, a user's UID will never change.

There are three view controllers in our app which concern user authentication, with each having a specific role. These controllers are as follows:

- `MainLoginVC`: gives users options whether to register with email, sign up with email, or use Facebook or Twitter as their identity provider;
- `SignUpEmailVC`: handles registration given an email address and password;
- `LoginEmailVC`: handles login with an email address and password.

The application's delegate (commonly referred to as "app delegate" or simply "delegate") is also involved in the login process as it checks if a user is logged in. If affirmative, it displays the user's items; if negative, it shows to the main login view controller (with options to log in or register). The delegate is part of every iOS app, and it gives developers the ability to handle setup after the application finishes launching, changes to state transitions (such as when the app moves from foreground to background execution), and incoming notifications [122]. We need to know if a user is logged in or not immediately after the app is launched, so the delegate is an adequate place for this type of check.

Registering an account and logging in are two features which require a network connection — verified as soon as the app delegate loads — to validate and authenticate a user with Firebase's servers. After login, almost all use cases take advantage of Firebase's data persistence which allows for offline usage.

MainLoginVC

This view controller is the first one shown when a user opens our app for the first time, or if they are logged out. It presents a user with four options illustrated in figure 5.1: log in via Twitter, log in via Facebook, register an account with an email address, and log in with an email address (complementing the previous registration option).

If a user chooses to use an identity provider, they are greeted with respective platform's OAuth login dialog. After inputting their account information and allowing access, the

[15] Not to be confused with the Firebase Real-time Database. The user database stores basic information about users; apart from the UID, it also includes a primary email address, a name and a photo Uniform Resource Locator (URL) for each user.

[16] For example, allowing users to register an account with the traditional email address and password method and posteriorly adding a Facebook or Twitter account which can also be used to log in to the account.

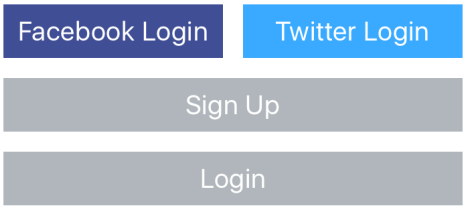


Figure 5.1: MainLoginVC options (cropped)

providers will supply the app with an authentication token[17]. We then pass this token to our wrapper instance method `signInWithCredential:provider:` to perform the login.

The wrapper method on `FIRAuth`'s `signInWithCredential:completion:` was written to include the provider used for the log in. The name of the provider is needed to update the user's profile photo (more on this below). Inside our wrapper method, we call `FIRAuth`'s sign in method and check for errors, displaying them if something went wrong.

If all is well, we check if the user already exists using an extension on `FIRAuth` named `checkUserExistsWithUID:completionHandler:`. We provide it with the unique ID generated when the user signed in, and it will check our database for its existence. If the user exists, we segue (i.e., transition) to the items view controller which shows the user's items. If the user does not exist, we add their UID to our database and present them with our onboarding experience.

In addition to logging in with a provider, the wrapper method also updates an existing user's profile photo (designated "avatar") by calling `updatePhotoURLForUser:provider:` in `MainLoginVC`. We check if we are hosting a user's profile photo. If so, it means the user has manually changed it inside Snapwish, so we ignore the update. If the avatar has not changed, the provider's name is necessary to check if the one we are logging in with matches the user's existing sign in method. We only update the avatar if it does, storing the image URL in our Firebase database (under the user's profile data).

SignUpEmailVC

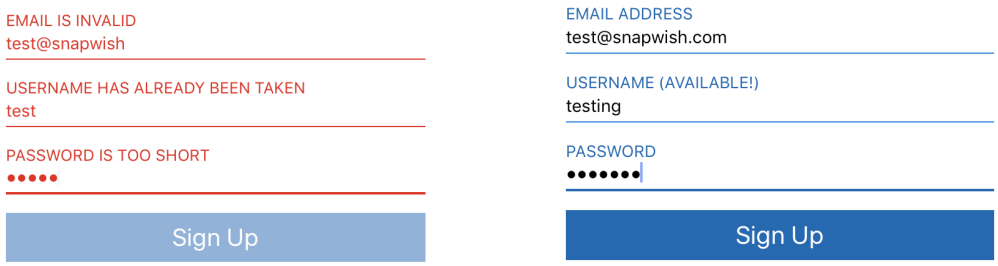
This is the view controller responsible for handling registration via email address. It presents three required text fields to the user — email, username, and password — during the sign-up process. The controller validates each field as soon as the user begins typing text into them. Three validations must pass for the sign-up button to be enabled (allowing the user to register an account):

1. The email field must pass a regular expression (regex) check which detects whether an email address is correctly formed;
2. The username has to be available[18] and contain between three and fifteen characters;
3. The password must have at least six alphanumeric or special characters.

We used a third-party library named SkyFloatingLabelTextField [123], which allows developers to use text fields that include an error message, as well as changing the color of the field,

[17] Twitter supplements this with a token authorization secret.
[18] We verify this by checking if the input username exists in our Firebase real-time database.

among other customizable properties. This is particularly useful when checks fail, as we do not need to display an alert view which covers the screen, detracting from the registration process. The user remains informed at all times about the validity of the text fields and consequently whether or not they can sign up. Figure 5.2 shows examples of validation error messages contrasted with optimal input.



(a) Text field validation error messages

(b) Optimal registration input

Figure 5.2: Registering a new account via the email method

Creating a user is similar to the identity providers, except here we invoke a method on `FIRAuth` named `createUser:withEmail:password:completion:` which will register a user with the email method in Firebase. If the completion handler contains no errors, we store the UID and username in the real-time database.

LoginEmailVC

The complement to the sign-up with email view controller. It is a simple controller with two text fields: email address/username and password (shown in figure 5.3), allowing a user to log in with a username or email. The fields are also validated in real-time to avoid performing unnecessary network requests containing invalid data between the client and Firebase's servers.

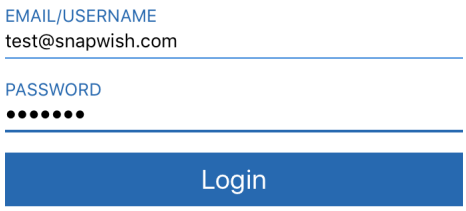


Figure 5.3: Logging in with email and password

The login button is enabled when both fields have valid data. We call the `FIRAuth` class method `signIn:withEmail:password:completion:`. If no error is returned, the user is now logged in. The controller informs the user on a login attempt using an inexistent or incorrect email or password. For security reasons, the specific field is not mentioned.

5.3.2 Items

Items are at the core of Snapwish's functionality as they are always involved in the main use cases of our app. We allow users to create and edit items, group them into wish lists and

share them with friends.

There are two view controllers which deal with visualization and management of items, and these are called `ItemsCVC` and `ManageItemTVC`. They make use of our `Item` entity extensively. Apart from the required unique key and Firebase reference common to every entity, a struct of type `Item` stores the following properties (the respective data type is listed in parenthesis):

- The item's `name`* (`String`);
- The UID of the user who added the item: `addedByUser`* (`String`);
- Its `description`: information about the item (`String`);
- Its `price` (`Double`);
- The URL of its `photo` (`String`);
- Whether it was `purchased` by someone[19] (`Bool` - default `false`);
- The `sortedOrder` of the item in the collection view when using default sorting (`Int` - default `-1`).

Properties marked with an asterisk (*) are mandatory.

ItemsCVC

This view controller is the first to be shown after the log-in flow. It falls back to an empty state if the user has not added any items yet; otherwise, it displays all the items they have added in a collection view. Each cell representing an item has a height of 175 points separated by a white line three points high, as shown in figure 5.4.

Items are sorted alphabetically by default, but additional sorting options are available (by price and by date, both ascending and descending). The user can also move items at will by dragging and dropping an item via a two-finger long press gesture.

Additional gestures include double-tapping a cell in quick succession to act as a shortcut to the `ManageItemTVC` view controller to edit an item; also, a long press gesture with one finger will bring a radial menu with quick actions [124]. This menu — illustrated in figure 5.5 — enables the user to edit or delete an item. Deletions always prompt for confirmation because this action will first remove the item from every wish list it was in and then permanently from the database itself.

Scrolling past the first item will automatically hide the navigation bar in an effort to give more screen real-estate to the collection view of items (figure 5.6 shows the status bar before and after scrolling). This technique can be found in many popular apps (e.g., Facebook and Instagram) and we implemented this using `TLYShyNavBar` [125], an open-source component.

[19] This is useful for shared wish lists: users can check off items as they are purchased. For secrecy's sake, we only show the purchase status to the buyer.

Figure 5.4: The items collection view controller (with items)

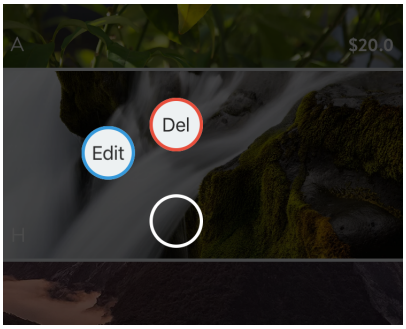


Figure 5.5: Radial menu with item actions

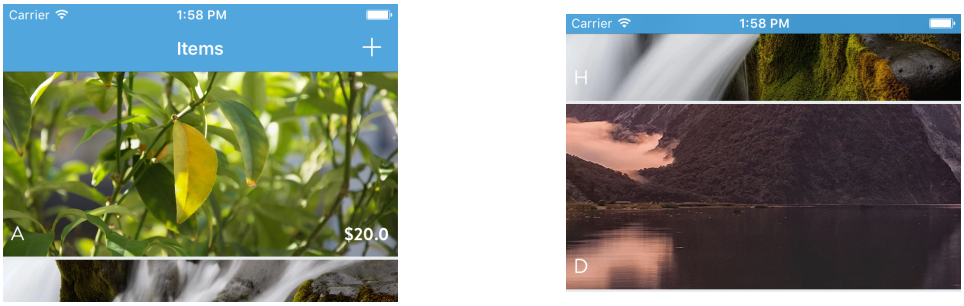
ManageItemTVC

Tapping the "Add" (+) button at the top of the `ItemsCVC` brings up this view controller. As illustrated by figure 5.7, it comprises a static[20] table view which contains an image view and multiple text fields and text views to edit the textual properties of an item. If invoked on an existing item, it pre-populates each cell with the item's existing data; if it is a new item, all fields will be blank, and the controller gives focus to the cell containing the item's name — this text field becomes the **first responder**.

The photo image view can be tapped to bring up an image picker (implemented using the `BSImagePicker` component [126]). Unlike the default system picker, this third-party alternative allows selecting and taking photos within the same view controller[21]. The user can switch albums by tapping the navigation bar title. Once a picture has been selected, we

[20] This means that the number of rows of this table view does not change at any time.

[21] With the system image picker, the user is almost always prompted beforehand if they want to take a photo or select an existing image from their library. This intermediate step is not required here.



(a) Status bar before scrolling

(b) Status bar after scrolling

Figure 5.6: ItemCVC status bar appearance changes

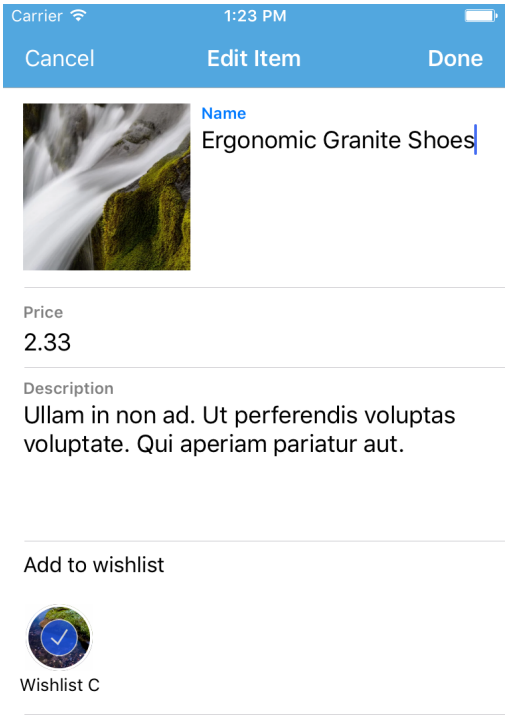


Figure 5.7: The manage item table view controller when editing an item

transition to a controller which crops the image to a specific aspect ratio. In this case, we want images with a 16:9 aspect ratio so they fit neatly inside our `ItemCVC` cells.

The image selection performed by the user provides us with a `PHAsset`, which in this case is a representation of a photo in the Photos library (this also includes photos taken with the camera). Assets must be fetched before we can work with them, so we call the `requestImage:for:targetSize:contentMode:options:resultHandler:` method in the `PHImageManager` class[22] which takes five parameters, the first being the asset itself. This method executes asynchronously, and after we verify that no errors have occurred during the fetch operation, the result handler assigns the obtained image to the image view. We also change the view controller property `hasImageChanged` to true, signaling that during the save operation — when the user taps the "Done" button — the old image must be replaced for the newly selected one.

[22] This class "provides methods for retrieving or generating preview thumbnails and full-size image or video data associated with Photos assets" [127].

The next step is to encode our newly obtained image to its data representation. iOS natively supports encoding images to a Joint Photographic Experts Group (JPEG) or Portable Network Graphics (PNG) representation. Both were adequate options for users' images throughout our app, but PNG's lossless ability weighs it down. Moreover, while JPEG without compression also looks great, it has no support for transparent alpha channels. Furthermore, delivering high-quality images to users with slow or metered Internet connections posed a challenge.

Earlier this year, we came across an engineering article by Dollar Shave Club (DSC) [128]. Their goal was to "shave" the image sizes in their app (they used PNG). They came across the WebP format by Google, and their testing concluded that "WebP-formatted images were 10x smaller than their PNG counterparts" [128], a significant reduction in size. Thus, we decided to try out WebP in Snapwish and compare it to JPEG and PNG. As iOS lacks native support for WebP, we employed the open-source `YYImage` library which contains an encoder for this format [129]. For JPEG and PNG encoding, we used the native encoder (as it is optimized for Apple hardware) by calling `UIImageJPEGRepresentation:image:compressionQuality:` and `UIImagePNGRepresentation:image:` [130], respectively.

Our tests measured **file size post-encoding**, which is the same regardless of the device used because the underlying data representation is equivalent. We encoded the same PNG image with 1, 5 and 10 megabytes (MBs) of size. All encoding was performed without added compression — except for a WebP compression test at 85% quality. Figure 5.8 shows the comparison in file sizes between the three formats post-encoding.

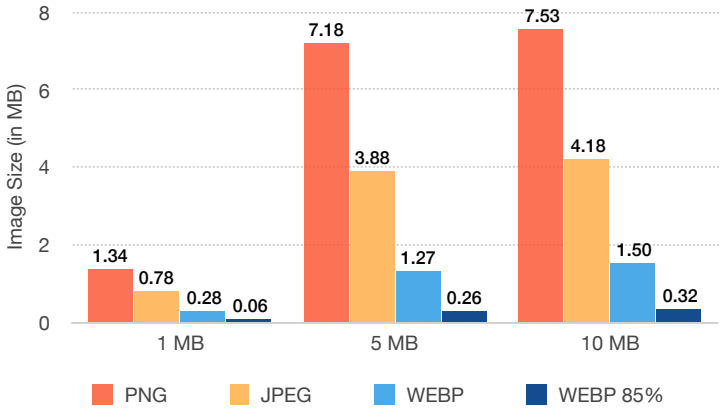


Figure 5.8: Comparison of Image Sizes Post-Encoding By Format

Our test results came out similar to DSC's. Overall, encoding a PNG image to a PNG representation usually resulted in a smaller file size, perhaps due to optimizations by Apple. The JPEG encoder offered minimal file size reductions across all tests, but the WebP format impressed delivering savings up to **66%** compared to the source PNG image – uncompressed. With 15% compression (at 85% quality), WebP output files almost **five times smaller** than their uncompressed counterparts. Also, when compared to the source 10 MB PNG, it managed to produce images **97.2% smaller** while still preserving sufficient quality for mobile usage.

The main drawback to using WebP is that images take longer to decode, but using a cache (namely `YYCache`, a component of `YYImage`) to save images after decoding and loading them in image views mitigates this inconvenience. The benefits of WebP also

largely outweigh the benefits: drastically smaller file sizes result in less required bandwidth to upload and download images and memory usage when decoding.

Before encoding, the image selected by the user is downscaled to a maximum resolution of 1920x1080 (1080p) (only if it is above these dimensions), preserving the original aspect ratio. This operation aims to reduce file size further before encoding to compressed WebP.

After the asynchronous encode process completes, we check for network connectivity. If the network is available, we upload the image to Firebase Storage using a method in the `FirebaseStorage` class named `put:data:metadata:completion:`. The Firebase Storage database works almost identically to the real-time database: our app is assigned a bucket and stores each file at a specific reference inside it [131]. A file name of each image is a universally unique identifier (UUID) and photos of items are stored at the path `"/images/<username>/<UUID>.webp"`. This guarantees that each user has a separate images folder and that they are all unique in name. To further save space in our app's bucket, we delete the old item photo before the upload of its replacement. The final part of this process saves the image's Firebase Storage URL to the real-time database so it can be downloaded at a later time.

If the user has no internet connection, however, they are still able to create or edit an item with a photo. The image is encoded as usual, but instead of uploading the image remotely, its data is stored in the real-time database's local cache (on the device itself) as a base64 string – Firebase cannot store `Data`[23] directly. This string can be handed over to our WebP decoder, and the result would be identical to decoding a WebP image's data.

5.3.3 Wish Lists

Wish lists compile an itemization of goods desired for special events, such as a birthday, an anniversary, Christmas, or simply as a personal collection. The goal of a wish list is to ease communication between the gifter and their intended recipient.

Just like items, there are two view controllers whose job is to display and manage wish lists: `WishlistsTVC` and `WishlistItemsTVC`. They work with our `Wishlist` entity, which is made up of the following fields:

- The wish list's name* (`String`);
- Its `owner`: the UID of the creator* (`String`);
- The `items` inside the wish list (an array of items; each element is a dictionary of format `<itemID>:true`);
- The `users` the wish list is shared with (an array of users; each element is a dictionary of format `<uid>:true`).

WishlistsTVC & WishlistItemsTVC

The `WishlistsTVC` consists of a simple table view where each cell represents a wish list (sorted alphabetically). A cell contains the name of the wish list, its item count, and the

[23] This Foundation class "provide data objects, object-oriented wrappers for byte buffers", i.e., data serialization [132].

number of people its shared with (as shown in figure 5.9). If the user has no wish lists, we display an empty stating prompting them to create one.

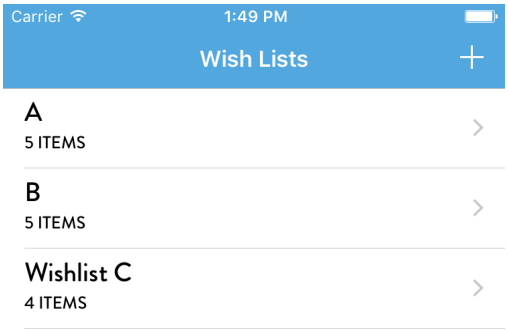


Figure 5.9: WishlistsTVC showing three wish lists (cropped)

Creating a wish list is as easy as tapping the "Add" button in the top right hand corner. An alert view will prompt for a name. Then, we present the `WishlistItemsTVC` modally which lists all the items owned by the user in alphabetical order. The user can tap each cell representing an item on this view controller to add items to the new wish list – each selected item is assigned a checkmark as figure 5.10 illustrates. There is no limit on the number of items one can place inside a wish list. Tapping the "Done" button on this controller without selecting any items will create a wish list with no items.

Users can edit wish list items and delete a wish list by swiping left on each cell. Deletions always prompt for confirmation to avoid inadvertent data loss. Double tapping a wish list's cell serves as a shortcut to edit the items inside it.

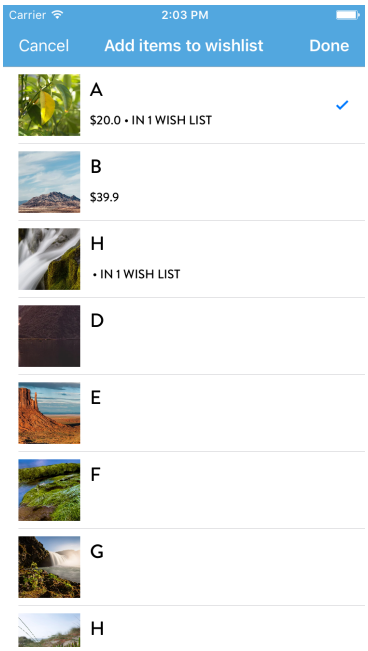


Figure 5.10: Adding items to a new wish list

5.3.4 User profile

Like items and wish lists, the user profile has its dedicated tab in Snapwish's navigation (a tab bar interface[24]). The profile houses all relevant information about a specific user, making use of all entities in our model. The user entity is made up of the following properties:

- A display name* (`String`);
- A username* (`String`);
- The URL of its avatar image (`URL`);
- A short bio of up to 160 characters (`String`);
- A random header hex color assigned by default (`String`);
- The URL of its header image (`URL`).

There are two view controllers that display and manage user profile-related info, respectively: `ProfileVC` and `EditProfileTVC`. Both are described below.

ProfileVC

An open-source recreation of the design of Twitter's user profile extensively inspired this view controller [133]. We chose to adopt this because it provides a modern and tested user experience that is clean and concise, yet also ideal for displaying a plethora of relevant information. It consists of two parts (illustrated by figure 5.11); from top to bottom: a header view and a portion of user information.

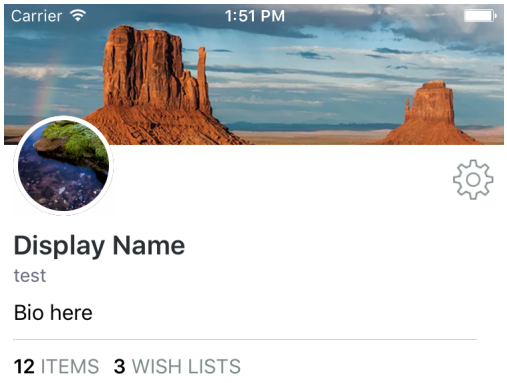


Figure 5.11: The profile view controller for user "test"

The header view was designed to fit a 1500x500 image[25]. It is made up of a stack of three views — illustrated by figure 5.12 — where the image view that displays the header image is the furthest at the back. We then overlay a darkening view to enhance contrast and legibility (which helps when the header is a bright picture). Finally, the original header image is blurred using a custom `UIView` class named `FXBlurView` and placed in a new image view at the forefront of the stack [134]. This view is hidden by default to accommodate the animation which only occurs when the user scrolls up on the `ProfileVC`: the header image

[24] A view at the bottom of the screen that separates view controllers into tabs.
[25] This is the same resolution used by Twitter.

slowly blurs and reveals two labels containing the user's display name and their count of items and wish lists. Figure 5.13 illustrates the latter.

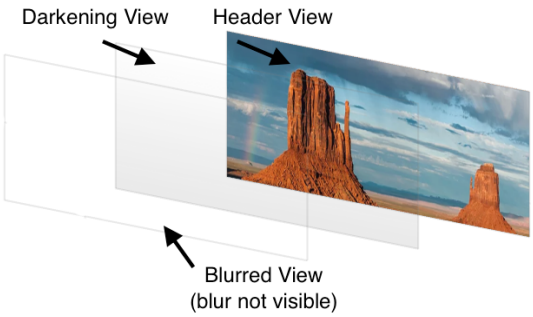


Figure 5.12: Header view stack

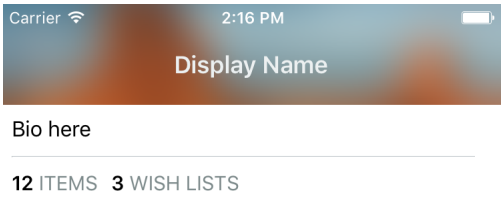


Figure 5.13: Blurred header in profile view (cropped)

If the user has not uploaded a custom image, during the first load of the profile view controller Snapwish will generate a random color using Chameleon, a third-party framework that makes working with colors easier [135]. It provides an extension method on `UIColor` named `randomFlatColor:` which generates a flat color[26] in either light or dark shades (we have excluded a few colors which aren't found on a common color wheel). The hex code of this color is then stored under a user's `headerHexColor` property which is referenced on subsequent loads. We convert its `UIColor` representation into an image which is assigned to the profile header image view as a placeholder until the user uploads a custom photo.

The user information part displays the user's avatar (i.e., icon) image, their display name, and username; we also display their bio if they have filled it in. A one point horizontal line divides personal *editable* information from dynamic yet *non-editable* content: the count of items added by the user, wish lists owned, and the number of friends. Snapwish updates all of this information in real time by using four Firebase observers: one for the profile user information, and the other three for the items, wish lists and friends count each. Regarding the latter, in each observer callback, we grab the number of children returned from our query by accessing the property `snapshot.childrenCount` and update the corresponding labels accordingly.

EditProfileVC

Tapping on the cog located on a user's profile brings up a series of options, including editing one's profile in a separate view controller (presented modally). The `EditProfileVC` —

[26] Twenty-four colors make up Chameleon's "flat" palette which includes those found on FlatUIColors — a website that lists 20 colors appropriate for flat design [136]. Since Chameleon provides a light and dark shade for each, there are 48 colors in total.

shown in figure 5.14 — allows a user to edit all of their profile information: display name, username, bio, and also upload or remove an avatar or header image.

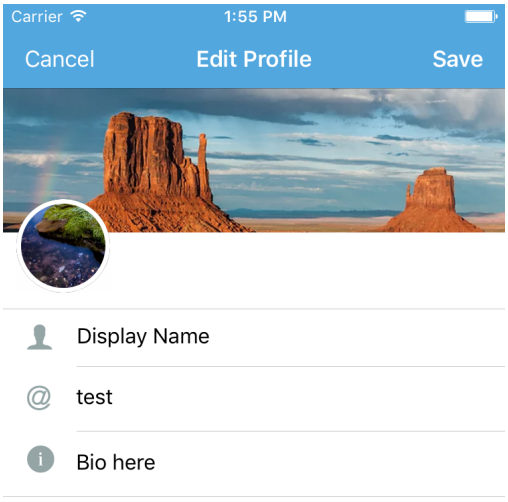


Figure 5.14: The edit profile view controller

The display name and username text fields (and text view for the bio) have a limit to the amount of alphanumeric characters which can be input: 30 for the display name, 15 for the username and 150 for the bio. The two mandatory fields (display name and username) have an additional minimum required count of one and three characters, respectively. The "Save" button only activates when these limits are respected.

The header and avatar image views contain a gesture recognizer whose handler displays a variety of options when tapped. Regarding the avatar, one can select to take a photo using the front-facing camera (by default) or select an image from their photo library. The header options include removing it and falling back onto the previously generated header template color, taking a photo with the rear camera (by default) or picking an image from the library. After the photo selection process, we use the `TOCropViewController` component to present a circular (for the avatar) or rectangular crop area (for the header) [137]. This component allows a user to pan the image around to include the area they desire which will be cropped and uploaded. It also helps in reducing the source image's dimensions if it is larger than the predefined targets: 400x400 for an avatar, and 1500x500 for a header.

The operation of saving changes to a user's profile information requires network connectivity, and is broken down into three main phases:

1. Make sure there are unsaved changes. If not, dismiss the view controller;
2. Verify if the avatar and/or header image has changed. If affirmative, upload the new image(s) to Firebase Storage and retrieve their URL;
3. Check the other user profile properties for changes and save those that have changed to the Firebase database, including any image URL's obtained from the previous step. Dismiss the view controller when the save has completed.

The first step checks if the `hasUnsavedChanges` boolean property is `true`, which gets set whenever the user performs an edit to any field or changes the avatar or header image. This is a pretty easy way to avoid saving unchanged data unnecessarily.

The second step is more complex. We make use of Grand Central Dispatch's (GCD) *dispatch queues* [138], which are used to aggregate similar operations. We can submit multiple independent work items and get notified when they have all completed. This is necessary because we can only proceed once all tasks are complete; in our case, we only want to save data to the Firebase real-time database after we retrieve the avatar/header image URL (or both).

We used two dispatch queues: one for uploading the images named `imageUploadDG` to Firebase Storage, and the other (named `userInfoDG`) for saving the rest of the data to our real-time database[27]. If the user has not changed their avatar or header, the former queue returns immediately and only the database save operation is performed. Otherwise, we check which images were modified and upload them[28]. The completion block for each image upload returns its respective download URL, which we store to later be accessed by the `userInfoDG` queue.

Once this queue gets notified, the changed fields will be part of a dictionary of changes — where its keys are the modified property names — which is used to update the values in our database (using the previously covered `updateChildValues:` method). This is the third and last phase of the save operation; its last part takes care of dismissing the view controller (now on the main thread).

5.4 Summary

This chapter focused on three aspects: our data model, its backend, and Snapwish's view controllers (whose task is to interpret the model and display it to the user). Before that, we covered the tools used for development of our application: the Xcode IDE with Swift 3 as the main development language.

We chose to use value types over reference types — i.e., structs over classes — which benefit from individual copies which guarantee that no other part of our app is changing the data. To further embody this, we made our model immutable by using constants (instead of variables) for all properties. The model requires creating a new object with updated values if data changes.

Next, we detailed how storing, retrieving and structuring data in a Firebase Database works. All data is stored as JSON which allows for fast data transfers and complex relationships via denormalization (a common trait amongst NoSQL databases). Furthermore, Firebase's security and rules (also structured as JSON) help validate data and restrict access to authorized users.

We divided our app into four distinct view controller groups to simplify its logical structure and explained the role of each controller. A group makes use of one or more model entities and usually performs one use case, but data is often shared between controllers to avoid hitting the network and take advantage of in-memory resources.

[27] All work in both queues is performed off the main thread (to avoid blocking the user interface).

[28] As with item photos, the old image is deleted from the cloud prior to the upload of the new image to save space.

To save space and data, we adopted WebP as the format for images uploaded by the user in Snapwish. We were able to reduce file sizes by about five times when compared to lossless PNG images while maintaining an acceptable level of quality (with 15% compression).

Chapter 6

App Testing, Profiling & Deployment

This chapter covers three main topics: app testing, profiling (using Instruments, bundled with Xcode), and deployment to the App Store.

Regarding app testing, we make use of tests Xcode supports — unit and user interface tests — to assess the functionality of Snapwish's implementation and its use case interaction flow, respectively.

Following the summary of our app analysis in Chapter 3, we perform a series of tests in the profiling stage to gauge Snapwish's performance and make sure that its battery and memory consumption are adequate. Once we are confident that Snapwish has received a high level of functional maturity and performance, the last section of this chapter discusses deployment: the App Store review guidelines and once (successfully) shipped, how developers can market their app to increase visibility and downloads.

6.1 Testing

Every moment we spend developing is an investment of a resource — in our case, time. One may then question why testing is needed when we can simply write code that complies with our use case specifications, debug it as needed and ship it when our app is considered "feature complete". The problem with this approach is that we miss out on the benefits of testing, and the most obvious one is finding bugs. There are a variety of bugs that testing can uncover, and the most common ones are called **regressions** [139].

Testing can find two types of regressions: functional or performance-based. Imagine a bug fix or a feature introduced in a specific version of an app which causes related tests to fail. In turn, this can impact other areas of the app and result in strange behavior, crashes and even loss of data — a *functional* regression. If the modifications introduce no new bugs but take a lot longer to run than before, this results in a *performance* regression.

Another benefit to testing is that it codifies the requirements of our APIs. Method declarations are not sufficient in understanding how to interact with the API itself or what kind of data is adequate to obtain an expected result. With tests, a developer can easily specify how the method is supposed to behave, which input to provide, and any other side effects that may arise. This clarity is particularly useful with a codebase shared among a team of developers [139].

Xcode supports three types of tests: functional tests, performance tests, and user interface tests. Functional tests focus on code functionality; performance tests are concerned about

how long the app takes to complete tasks, and UI tests are concerned about user interface flows.

This subsection covers the testing process of our wish list management application regarding functional and UI tests.

6.1.1 Unit tests

Unit testing is the most common type of functional test. It considers the smallest testable component in an app (e.g., a method in a class) that serves a unique purpose.

XCTest vs Quick + Nimble

Xcode unit testing uses the XCTest framework [140]. From Chapter 2, Section 2.3.3, there are many third-party frameworks available to make XCTest more accessible. Kiwi is the most popular one for Objective-C code [141], but we prefer to write our tests in Swift to maintain consistency between our functional code and our unit tests. Thus, used Quick, a behavior-driven development framework for Swift (and Objective-C) [102]. Quick also comes with Nimble (a matcher framework) and extensive documentation [142].

The main difference between XCTest and Quick is clarity. More than anything, unit tests should make clear exactly what the issue is. To illustrate this, consider the following function in listing 6.1 which, when given an array of items, returns only those which have a non-empty description.

```
func havingDescription(items: [Item]) -> [Item] {
  return items.filter { $0.description != nil }
}
```

Listing 6.1: A Swift function which returns only the items with descriptions

Now let's see how our unit test would look like using XCTest (listing 6.2).

```
let iphone = Item(name: "iPhone", price: 699, description: "An iPhone 7")
let galaxy = Item(name: "Galaxy S7", price: 650, description: nil)
let htc = Item(name: "HTC 10", price: 650, description: "")
let phonesWithDescription = havingDescription([iphone, galaxy, htc])
XCTAssertTrue(phonesWithDescription.elementsEqual([iphone]))
```

Listing 6.2: A unit test on the havingDescription function using XCTest

This method fails with the following message shown in figure 6.1. The error method is vague and does not provide us with information on how to solve the issue. While we could write a message to accompany `XCTAssert`, the testing framework should take care of this. Enter Nimble.

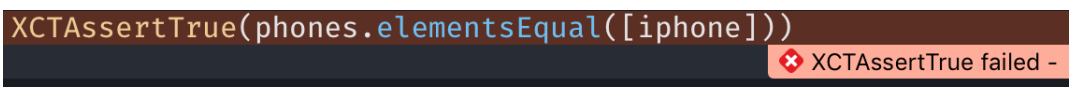


Figure 6.1: An error message using XCTAssert

Nimble improves readability on test assertions and failure messages. Instead of using `XCTAssert`, we can declare this expected result with the syntax shown in listing 6.3.

```
let iphone = Item(name: "iPhone", price: 699, description: "An iPhone 7")
let galaxy = Item(name: "Galaxy S7", price: 650, description: nil)
let htc = Item(name: "HTC 10", price: 650, description: "")
let phonesWithDescription = havingDescription([iphone, galaxy, htc])
expect(phonesWithDescription).to(equal([iphone]))
```

Listing 6.3: A Swift function which returns only the items with descriptions

This assertion reads like grammatically correct English and when our test fails, it includes a simple yet clear error message by default. Therefore, we get the ease of use paired with clarity at the point of failure, as figure 6.2 illustrates.

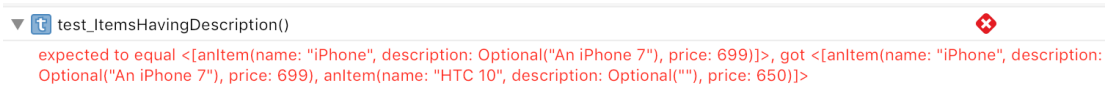


Figure 6.2: An error message using Nimble

Now we know that we forgot to add an extra check for non-null empty strings. We were expecting only `iphone` to be included as a result of running `havingDescription:`, but the result contained that and `htc` — whose description is a non-null empty string. This makes it easy to solve the problem by modifying our initial function as listing 6.4 shows.

```
func havingDescription(items: [Item]) -> [Item] {
  return items.filter { $0.description != nil && $0.description != "" }
}
```

Listing 6.4: A Swift function which returns only the items with descriptions (amended)

Methodology

We conducted extensive unit testing on methods which make up the majority of our app's functional requirements. Instead of simply testing code, we tested *behavior* (using Quick), i.e. tests should only fail if the application behaves differently[1]. For example, when we save a wish list to the database, we should make sure that it is the same object when reading data, instead of asserting whether the Wishlist entity count has increased by one.

To better organize our functional and performance tests, we grouped them into one test bundle containing multiple classes. These classes serve to functionally segregate tests by the four view controller sections mentioned in our implementation: authentication (and user onboarding), items, wish lists, and user profile.

The functional tests use the Quick framework, but because of its limitations, the performance ones rely on Apple's default XCTest framework. Thus, they are located in separate files (which does help with visual distinction) but located in the same test bundle.

Below we describe how we used Quick to test our app's behavior by including a couple of examples with code excerpts which showcase their structure and syntax.

[1] This is commonly referred to as *behavior-driven development*.

Functional testing with Quick

Each specification (spec) class using Quick subclasses `QuickSpec`. Quick uses `describe` and context closures to detail behavior and provide information about what is being tested. Inside each block, `it` closures specify expected *outcomes*. We can use Nimble assertions such as `expect(variable).to(beTrue())`, which make it easy to understand which variable we are testing and what its value should be after the individual unit test completes.

Consider listing 6.5 which uses our item entity. This example is about saving to a database and making sure the retrieved item key is the same we previously generated and saved.

```
class ItemsSpec: QuickSpec {
  override func spec() {
    // A reference to the test save path (/test/items)
    let itemsRef = FIRDatabase.database().reference().child("test").child("items")

    describe("an item") {
      // Create an item
      let item = Item(name: "test", description: "", price: 0.0, photo: "",
  addedByUser: "test", wishlists: [:])

      context("when saved to Firebase with a generated key") {
        // Generate a key for the new item locally
        let preSaveRef: FIRDatabaseReference = itemsRef.childByAutoId()
        var postSaveRef: FIRDatabaseReference?

        it("should have the same key post-save") {
          // Wait 5 seconds for the item to be saved at the generated key path
          waitUntil(timeout: 5, action: { (finished) in
              preSaveRef.setValue(item.toJSONDict(), withCompletionBlock: { (_,
  saveRef) in
                  postSaveRef = saveRef
                  finished()
              })
          })

          expect(postSaveRef?.key).toEventually(equal(preSaveRef.key))
        }
      }
    }
  }
}
```

Listing 6.5: A Quick spec with a context for saving an item to Firebase

We describe an item by creating it. When we mention saving to the database, we enter a new `context`, which is Quick's way of specifying an individual unit test. In XCTest, this could be a function named `testItemDatabaseSave`. Closure descriptions make tests read fluently e.g., "**an item when saved to Firebase with a generated key should have the same key post-save**", where blue represents `describe`, green `context`, and red `it` (i.e., an expected outcome) [102].

We expect that the key we generate for the item is the same after the save operation has completed. If the keys match, we are guaranteed to be dealing with the same item.

Another test example is to instantiate one of our view controllers and invoke its methods to make sure we obtain an expected result. In listing 6.6, we used the `deleteItem:at` method in `ItemsCVC`. This method takes a cell's row index and deletes the item at that index from the Firebase database (and consequently, our updated model).

The items collection view controller checked for nullability after instantiation (using `setupItemsCVC`). We then fetch items to populate the controller's array of items. The array is asserted to check if it is not empty. These are two preconditions which are necessary to delete an item. The item deletion function returns `true` if the item was successfully deleted or `false` if an error occurred. We can check this to guarantee the item at the row we specified is no longer present in the database with `expect(deleted).toEventually(beTrue())`. We use `toEventually` to signify that the `deleted` variable might take a while to change to `true` due to the nature of Firebase's asynchronous operations.

```
func setupItemsCVC() -> ItemsCVC? {
  let storyboard = UIStoryboard(name: "Main", bundle: Bundle.main)
  let nav = storyboard.instantiateViewController(withIdentifier: "ItemsNC") as?
  UINavigationController
  let itemsCVC = nav?.topViewController as? ItemsCVC
  _ = itemsCVC?.view // calls viewDidLoad:

  return itemsCVC
}

describe("the items collection view controller") {
  let itemsCVC = setupItemsCVC()

  it("should be instantiated") {
      expect(itemsCVC).toNot(beNil())
  }

  it("should have items") {
      waitUntil(timeout: 5, action: { (finished) in
          itemsRef.observeSingleEvent(of: .value, with: { (snapshot) in
              for item in snapshot.children {
                  if let snapshot = item as? FIRDataSnapshot {
                      let item = Item(snapshot: snapshot)
                      itemsCVC?.items.append(item)
                  }
              }

              finished()
          })
      })

      expect(itemsCVC?.items).toEventuallyNot(beEmpty())
  }

  context("deleting an item") {
      it("should be removed from the database") {
          // Remove item at row 0
          let deleted = itemsCVC?.deleteItem(at: 0)
          expect(deleted).toEventually(beTrue())
      }
  }
}
```

Listing 6.6: A Quick spec invoking view controller methods

Performance testing

A performance test takes a block of code and runs it ten times consecutively. Each time a run occurs, the test measures the amount of time taken and its standard deviation. The results are grouped to create a baseline used as the comparison for subsequent test runs.

A block of code can be measured temporally by passing it into the `measure` closure inside a test method. Listing 6.7 shows how we measured the performance of Snapwish's WebP image encoder with a 1 MB image (locally available to avoid network requests)[2].

```
func testWebP85EncoderPerformance() {
  // Load the image from our test bundle
  let testBundle = Bundle(for: type(of: self))
  let image: UIImage? = UIImage(named: "1mb",
                                in: testBundle,
                           compatibleWith: nil)
  expect(image).toNot(beNil())

  self.measure {
    waitUntil(action: { (finish) in
      // Encode the image to WebP
      image?.encodedToWebPData({ (data) in
        finish()
        expect(data).toNot(beNil())
      })
    })
  }
}
```

Listing 6.7: Measuring performance of our WebP image encoder

The `encodedToWebPData` method is an extension on `UIImage` which instantiates the encoder, sets its image format and quality to WebP at 85% quality and encodes the image, returning the data of the encoded image. This method is asynchronous, so we had to wrap it in a `waitUntil` block and signal to the test when it returned via the use of `finish()`.

After the first test run, we can set the temporal average value as a baseline. After this, we executed the tests again to see how they performed comparative to our first run. The results came in almost identical, with a negligible performance variance and standard deviation. Clicking on the gray checkmark in the gutter next to our test presents a pop-up with detailed information about our last test run (figure 6.3). The graph shows us how each of the ten tests performed according to the baseline. We can see that each encode takes about a tenth of a second (0.1s) to complete.

While performance tests are useful to measure whether future code changes impact the app's responsiveness, test results can vary greatly in production. This variance is because performance tests run with the **Debug** configuration instead of **Release** — the latter has (increased) compiler optimizations which may improve performance significantly.

Nonetheless, performance tests are useful to measure parts of our app which require higher CPU and memory resources and how long they are taking. Energy consumption directly ties to resource usage, so it is important to execute tasks quickly to avoid detrimental effects on battery life. Another benefit is quickly determining if an update to an involved third-party component impacts performance (so we can roll back to a previous version if required), or testing similar components to see which is faster.

[2] Nimble assertions can be sprinkled throughout unit tests to assert certain values — including performance tests. We prefer using them to XCTest assertions which favor verbosity over clarity.

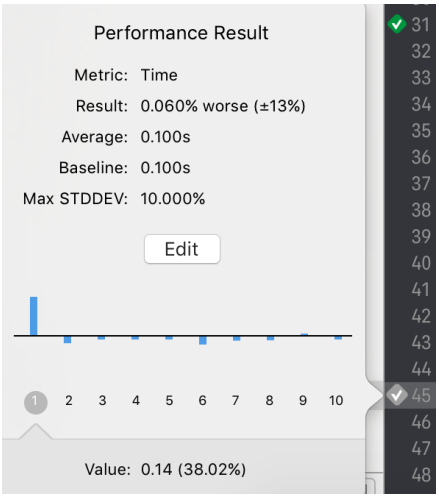


Figure 6.3: Encoding performance test result information

6.1.2 User interface testing

With Xcode 7, Apple introduced UI tests[3] — an intuitive way for developers to verify the state of their app's user interface [143]. A UI flow is recorded in the iOS Simulator, which is then played back and checked against assertions of the UI state (e.g. buttons, labels, image views). If any of the assertions fail, then the UI is in an unexpected or inconsistent state.

User interface testing is useful for debugging issues related to user interface elements not loading or incorrectly displayed values due to human error or erratic programming assumptions.

While UI tests do not intend to cover every usage scenario, they are a powerful and necessary component of a developer's toolset for the following reasons:

- They allow developers to record tests once for each major use case, and play them back as required;
- As they depend on correct labeling, this encourages developers to improve accessibility in their app which has a positive effect on the experience of disabled users;
- Developers can guarantee that the user interface stays consistent via the use of assertions;
- They save time as no manual input is necessary to test use case flows extensively. With continuous integration, user interface tests can be completely automated just like unit tests.

We used UI testing to quickly check if the flow of our various use cases was as expected (e.g., login flow from the welcome screen to the items view). As an example, listing 6.8 includes a user interface test that corresponds to creating a new item. This test assumes a logged in user on the `ItemsCVC`, whose header includes the "Add" button. UI tests depend on adequate accessibility labeling to identify views and controls when recording (which can be added from the Storyboard editor inside Xcode).

[3] Like unit tests, user interface tests make use of Apple's XCTest framework.

```
func testCreateItem() {
  let app = XCUIApplication()
  let faker = Faker(locale: "en") // Create a Faker object with an English locale
  app.navigationBars["Items"].buttons["Add"].tap()

  // Get the table view of the manage item TVC
  let tablesQuery = app.tables

  // Type a random name
  tablesQuery.textViews["Name"].typeText(faker.commerce.productName())

  // Type a random price
  let priceTextField = tablesQuery.textFields["Price"]
  priceTextField.tap()
  priceTextField.typeText("\(faker.commerce.price())")

  // Type 3 sentences of dummy text for description
  let descriptionTextView = tablesQuery.textViews["Description"]
  descriptionTextView.tap()
  descriptionTextView.typeText(faker.lorem.paragraph())

  // Add item to a wish list
  tablesQuery.images["Wishlist Image"].tap()

  // Tap on the photo image view
  tablesQuery.images["Photo"].tap()
  let imageCount = app.collectionViews.cells.count
  // We should have at least one image to select
  expect(imageCount).to(beGreaterThan(0))
  // Select a random image
  app.collectionViews.cells.element(boundBy: UInt.random(min: 0, max: imageCount - 1))
  .tap()
  app.navigationBars.buttons["Done"].tap()

  let addItemButton = app.navigationBars["Add Item"].buttons["Done"]
  // We should be able to save the new item
  expect(addItemButton.isEnabled).to(beTrue())
  app.navigationBars["Add Item"].buttons["Done"].tap()

  // Wait 10 seconds for the add to complete
  sleep(10)
}
```

Listing 6.8: A UI test that creates a new item

We make use of Fakery, a third-party component which generates fake data for a variety of locales [144]. Fakery added dynamism to our UI tests as each run uses a distinct set of data. Thus, not only did we have items with different information, but we could also check how the user interface reacted to varying string lengths (i.e., for item names).

Nimble assertions are also useful to assert the state of UI components. For example, when we decide to select a photo for the new item, it is expected that the user has at least one image in their photo library; else, the assertion fails, and the test terminates at that point.

The simplest way to wait for an asynchronous task to complete is to set a reasonable timeout period. Here, we wait for ten seconds before declaring the test as failed. For test purposes, this delay suffices under most network conditions[4] as the only network requests we make immediately are for uploading the item photo to Firebase Storage. The item's data can be cached locally. It is important to note that since UI tests run on a separate thread (to

[4]We tested multiple scenarios by using iOS's link conditioner which simulates different network quality and reliability conditions.

have no impact on the app's responsiveness), the `sleep` function does not block the user interface during its execution.

6.1.3 Beta-testing process

Beta-testing on our app was conducted to obtain feedback regarding the UI flow, different network conditions, concurrent device usage, and more. This stage assumed that the core functionality was at an acceptable level (after all alpha stages); moreover, that most functional requirements were available (even if not completed).

Development on Snapwish began on April 25, 2016. The alpha stage lasted until August 23, 2016, which coincided with the first public beta release to a group of five testers. TestFlight (Apple's testing platform) was used [145], which made it easy to invite, deploy and subsequently receive feedback from beta-testers. Snapwish betas required iOS 9.3 or later.

The beta release timeline is listed below, along with a brief change log and the most common feedback obtained from our testers. Please note that this timeline is **not final** as the app remains in development.

v1.0 beta 1 (August 23, 2016)

Testers were asked to experiment with the app as much as possible: test all available use cases, report any bugs and crashes found, as well as offering suggestions.

This seed lacked user onboarding (added in beta 7), there was no way to delete an item once created, and no sharing was possible.

v1.0 beta 2 (August 30, 2016)

Testers noted that the method to invoke editing an item was limited, so beta 2 introduced a radial menu to easily edit or delete an item (activated by holding down with one finger on an item's cell).

Also, we added the ability to reorder items and edit wish list items.

v1.0 beta 3 (September 21, 2016

Beta 3 was released almost a month after the previous beta and focused primarily on fixing bugs found by our internal tests and testers. The most prominent was an annoying bug where changes to avatar or display name in the profile view controller would fail to sync to multiple devices.

v1.0 beta 4 (September 26, 2016)

This beta represented a significant internal update, as we switched to immutable models to avoid concurrency bugs. We also fixed a major bug where adding an item would try to update it instead.

v1.0 beta 5 (September 29, 2016)

This beta came out just three days after beta 4 and represented an improvement of resources by adding and removing Firebase listeners in `viewWillAppear:` and `viewDidDisappear:`, respectively (instead of adding them in `viewDidLoad:` for the entire life cycle of the controller).

v1.0 beta 6 (October 6, 2016)

With Swift 3 being out for almost a month and third-party developers updating their open-source components and frameworks to this source-breaking update, we were finally able to convert our codebase. The conversion represented a significant undertaking as we had to make sure the app's functionality remained intact without introduced no new bugs.

We also diagnosed and fixed a case of excessive memory usage when retrieving a photo from the user's photo library by profiling our app with the memory allocation tool[5].

v1.0 beta 7 (October 23, 2016)

This beta was a significant release, adding user on-boarding, empty states, and enabling sharing of items between users. We also made it easier for users to add items to lists by showing a list of wish lists when adding or editing an item.

6.2 Profiling

In iOS, profiling is an important process to gauge the performance of an app. It concerns aspects such as performance, memory allocations and leaks, energy consumption (battery), network requests, and many others. It also helps developers spot and fix issues that are not commonly found by running unit or UI tests individually[6] [146].

Profile builds use the Release schema and compiler optimizations usually come into play here compared to Debug builds. This schema represents an App Store build of the application so we can test performance against the most optimized version of our app.

Xcode includes a Debug Navigator which includes information in a dashboard view. Hitting the command (cmd) + F6 key combination brings up this view showing all the CPU, memory, energy, disk and network requests an application is performing. This serves as a good indicator of where to start profiling.

[5] This particular issue is covered in the Profiling section under "Memory allocations and leaks".

[6] We recommend profiling on a physical iOS device instead of the simulator to accurately represent a production environment.

Figure 6.4 shows the Debug Navigator providing information about a debug run of Snapwish. The spikes of CPU activity represent various operations, with the heaviest being decoding images of items in parallel as soon as the app is opened. By comparison, user-initiated operations like creating a new item prove to use much less CPU resources because the only intensive task is encoding a single image.

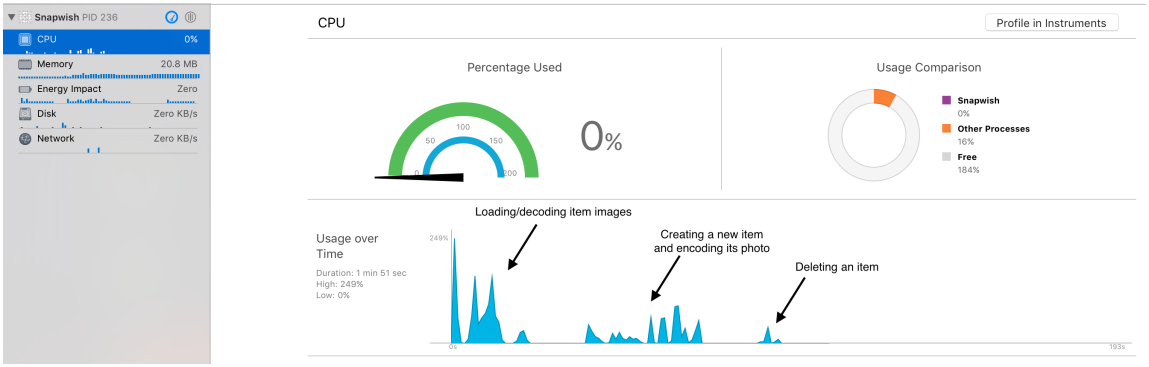


Figure 6.4: Debug Navigator showing the CPU tab during a debug

Profiling is performed with an OS X app called Instruments, bundled with every Xcode installation. This app includes a variety of profiling templates — shown in figure 6.5 — of which we used a subset to profile the following:

- Method response time (how long it takes to complete tasks, assuming optimal and sub-optimal conditions) using the Time Profiler;
- Memory allocations and leaks;
- Network requests (errors and data);
- Energy consumption (which methods are consuming the most amount of battery).

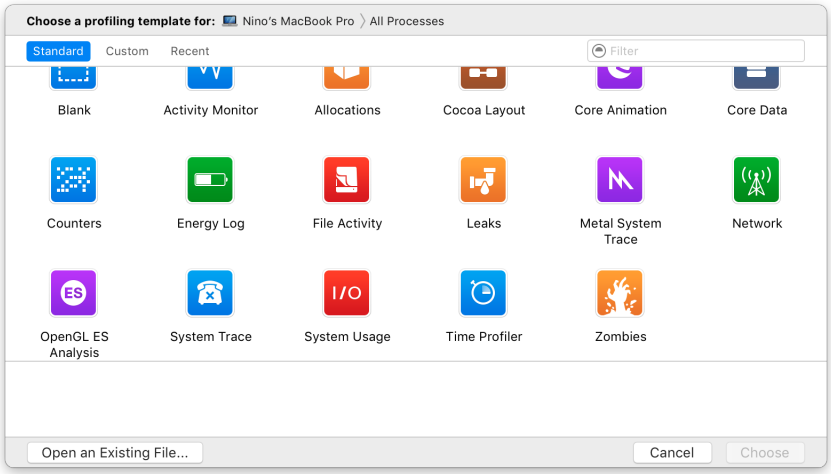


Figure 6.5: Profiling templates in Instruments

We list details about each profile test below, as well as examples applied to our app. We ran each test **five times** (sometimes under different conditions) to guarantee that abnormal results were not simply outliers but regular, reproducible occurrences.

6.2.1 Time Profiler

To start profiling CPU usage, we can click the "Profile in Instruments" button in the Debug Navigator and select the Time Profiler tool. This test needs to be run on an actual device to obtain accurate results, as the test will report CPU information from that device. Ideally, running the app on legacy hardware can prove to be an effective gauge on how it will perform on the slowest compatible device (in our case, an iPhone 4S or 5th generation iPod touch – we tested on the latter).

CPU profiling works by sampling running processes at specific intervals. By default, it checks every millisecond but this interval can be modified. This allows it to determine how long processes have been running for between each snapshot.

Figure 6.6 displays the Time Profiler's user interface, split into two halves. The top half shows a chart of CPU usage over time, while the bottom half shows the process call tree. The black flag in the time track shows the instant Snapwish became the foreground app. When running the profiler with default settings, analyzing the call tree is not a trivial task because it shows activity not related to the app itself. Therefore, developers should activate the following options located at the right of the call tree [147]:

- **Separate by Thread**: shows processes by thread to help find overloaded threads;
- **Invert Call Tree**: reverses the stack;
- **Hide System Libraries**: hides non-app related information;
- **Flatten Recursion**: combines recursive calls into one single call for less verbosity;
- **Top Functions**: shows the functions that are consuming the most CPU time (helpful to find expensive methods).

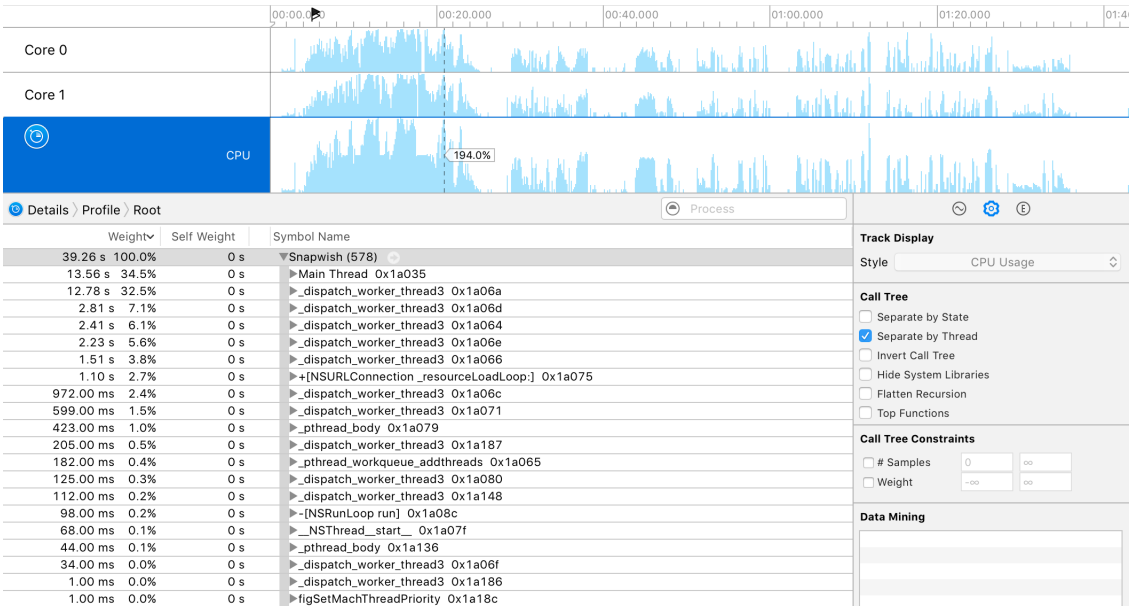


Figure 6.6: Time Profile with default options selected

As shown by figure 6.7, these options help simplify our analysis as the call tree focuses solely on our app's methods.

Our sample above consisted of opening the application which loads a list of ten items and their photos. Then, we created a new item with a photo, saved it, and proceeded to delete it once it was saved successfully. We also created a wish list and edited the user's avatar and header photo.

The Time Profiler states that the app spent just over a third (37.4%) of the time performing operations on the main thread. The dispatch queues handle image encoding and decoding, fetching data from our Firebase Database, and uploading images to Firebase Storage, so more time is spent there.

All cores show almost identical amount of work, so our app is successfully load-balancing work between both cores of the iPod. CPU usage shows peaks of high activity but they rarely go above 100% usage[7] (excluding the multiple image decoding process at the start). We are also caching images once they are decoded and loaded in our collection view of items. Finally, we don't make use of expensive animation or drawing functions in this use case.

Figure 6.7: Time Profile with custom options selected

If a developer followed good engineering practices during implementation, there might not be many optimizations to perform here. However, one should always keep these principles in mind as a path to CPU optimization:

- Perform any expensive operation (e.g., image or JSON decoding, and database operations) on a background thread, using the main thread only for user interface updates (and other very light tasks);
- Ensure that all cores are being equally taxed (i.e., load balanced);
- Cache any frequently accessed resource using appropriate caching mechanisms or libraries to avoid reloading (this also helps with energy efficiency);
- Only update the user interface when necessary (as these involve multiple operations in the main thread's run loop).

[7] A full load for this device would be 200% because it has two cores.

6.2.2 Memory allocations and leaks

The Time Profiler is important, albeit just one piece of the profiling puzzle, so we should not infer that our app is App Store ready without further testing. While time profiling can help make apps more responsive and save battery life, it can not determine how it consumes and frees memory.

Instruments includes two tests for memory testing — "Allocations" and "Leaks". The latter includes most features of the former, but also checks if objects are being leaked. This is a crucial test because we must ensure the app consumes only the necessary amount of memory without leaking. If not, it could lead to memory warnings being triggered, and in the worst case, iOS forcefully terminating the application which negatively impacts user experience.

Although our test showed no leaks, we noticed that memory would not be freed after fetching an image from the user's photo library when adding or editing an item, causing an increase every time this happened. This was determined to be caused by assigning the newly obtained image to a `UIImage` variable. Whenever its value changed, it automatically assigned the image to the item's image view (in `ManageItemTVC`). We mitigated the issue by simply assigning the new image to the image view. This allowed us to save about 4 MB of memory each time, a decent improvement especially if a user uploads or changes many items in quick succession.

Figure 6.8 shows the test graph pre-fix, while figure 6.9 displays the post-fix one. In the former, one can easily notice spikes in the graph which symbolize sharp increases in memory usage; in the latter, the graph has less sudden increases which leads to more linear consumption.

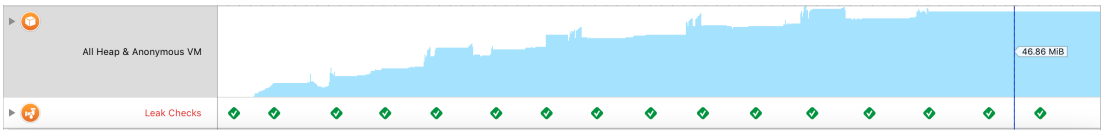


Figure 6.8: Memory graph pre-fix

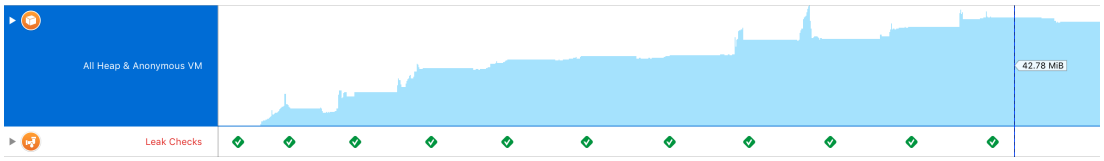


Figure 6.9: Memory graph post-fix

We can use memory profiling to find issues which otherwise would not be uncovered by unit or user interface tests. They are typically issues that become gradually worse (i.e., memory increases without being freed) as long as the application is being run. A user might encounter memory issues and without profiling developers would have a hard time figuring out that excessive memory consumption caused iOS to terminate the application (in a worst case scenario).

Note that superfluous resources should be freed when the application switches to background execution, because iOS will terminate applications consuming the highest amount of memory first to accommodate the foreground app's needs. Conversely, we should be responsible with our app's memory to play nicely with other running apps.

6.2.3 Network requests

For the network requests test, we preferred the dashboard information presented by the Debug Navigator over the Network profiling tool because we can see a visual distinction between received and sent data, as well as the active connections and the quantity of data transferred.

We performed some actions in Snapwish that would cause data to be exchanged between our app and Firebase's servers, namely uploading and downloading images and creating and deleting items/wish lists (shown in figure 6.10). This caused data transfer to and from the Firebase Real-time database, and the Storage database. Overall, the amount of data transferred totals of 0.3 MB sent and received, which is considerably low. We attribute this to our use of the WebP image format (with compression) and the fact that all the other data our app uses is in JSON. This allows for small data transactions which result in a responsive app even on slow(er) networks.

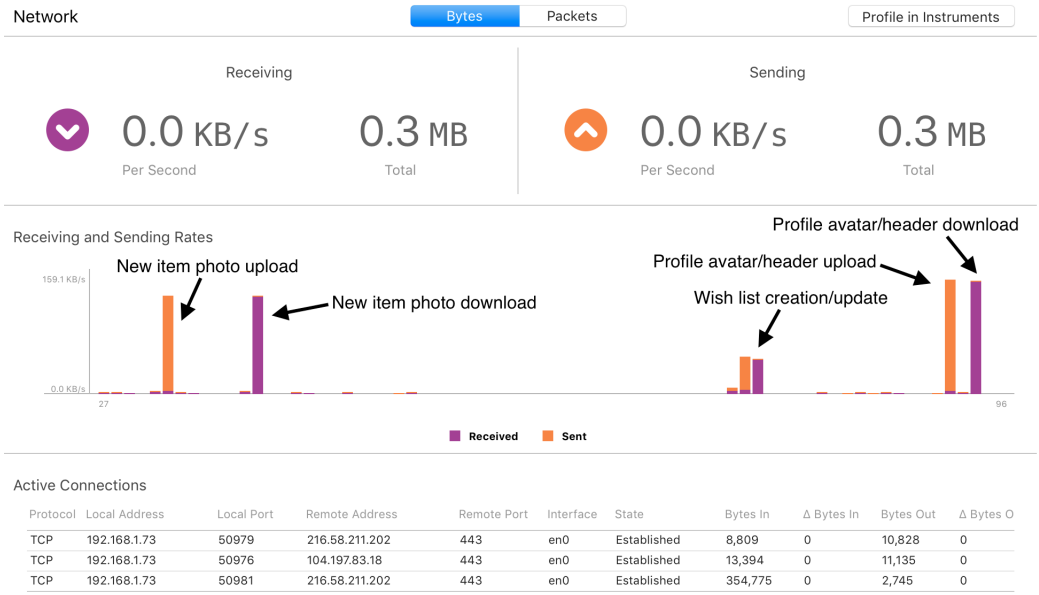
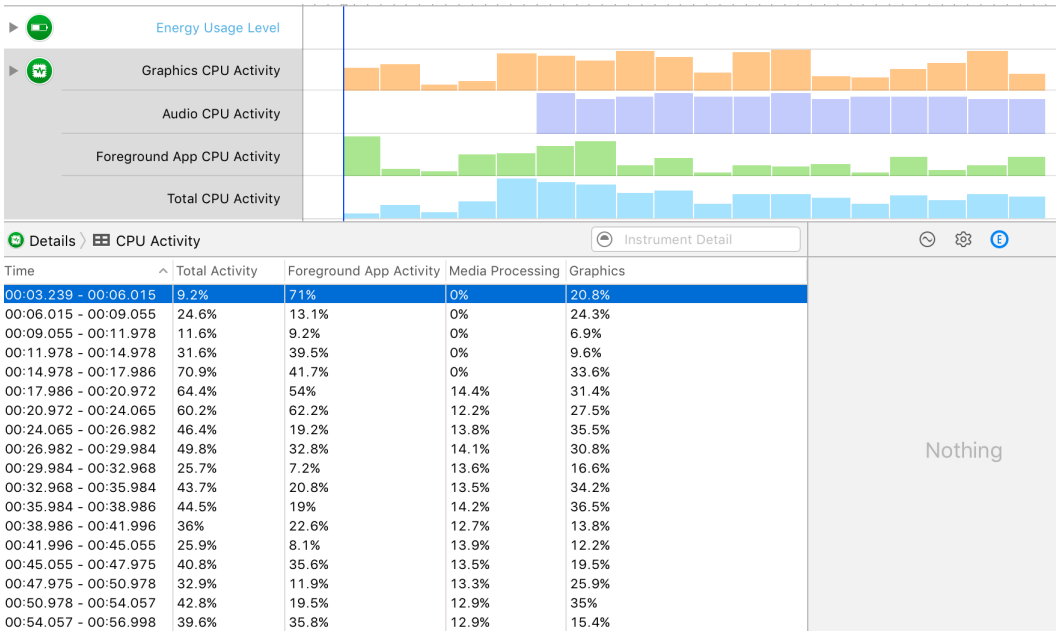


Figure 6.10: Network requests in the Debug Navigator

6.2.4 Energy consumption

Another important aspect after analyzing CPU, memory and network statistics is their impact on energy consumption (i.e., battery life). We used the "Energy Log" tool for this purpose and ran the app for a minute each time. The results displayed in figure 6.11 show that the app's foreground activity spiked when it was opened because it was fetching photos of items and decoding them. For other use cases such as creating and editing items and wish lists, activity reported significantly lower values.

Our app showed constant amounts of graphics activity because it was in the foreground and constantly refreshing its views as response to user interaction.



Time	Total Activity	Foreground App Activity	Media Processing	Graphics
00:03.239 - 00:06.015	9.2%	71%	0%	20.8%
00:06.015 - 00:09.055	24.6%	13.1%	0%	24.3%
00:09.055 - 00:11.978	11.6%	9.2%	0%	6.9%
00:11.978 - 00:14.978	31.6%	39.5%	0%	9.6%
00:14.978 - 00:17.986	70.9%	41.7%	0%	33.6%
00:17.986 - 00:20.972	64.4%	54%	14.4%	31.4%
00:20.972 - 00:24.065	60.2%	62.2%	12.2%	27.5%
00:24.065 - 00:26.982	46.4%	19.2%	13.8%	35.5%
00:26.982 - 00:29.984	49.8%	32.8%	14.1%	30.8%
00:29.984 - 00:32.968	25.7%	7.2%	13.6%	16.6%
00:32.968 - 00:35.984	43.7%	20.8%	13.5%	34.2%
00:35.984 - 00:38.986	44.5%	19%	14.2%	36.5%
00:38.986 - 00:41.996	36%	22.6%	12.7%	13.8%
00:41.996 - 00:45.055	25.9%	8.1%	13.9%	12.2%
00:45.055 - 00:47.975	40.8%	35.6%	13.5%	19.5%
00:47.975 - 00:50.978	32.9%	11.9%	13.3%	25.9%
00:50.978 - 00:54.057	42.8%	19.5%	12.9%	35%
00:54.057 - 00:56.998	39.6%	35.8%	12.9%	15.4%

Figure 6.11: Energy Log profiling test

Unfortunately, we weren't able to obtain the energy usage level, which ranks an app's consumption on a scale of 0 to 20 at each measured interval (one millisecond by default)[8] because we had to profile our device wirelessly. Apple states: "Bonjour and multicast must also be enabled on your wireless network access point" [148]; alas, we lacked Bonjour support on our wireless router, so the only diagnostics we were able to run were tethered.

6.3 Deployment

We plan to release this app on the App Store after the beta-testing process is complete. To deploy apps and use advanced app capabilities (i.e, iCloud), one must have a valid Apple Developer membership (€99/year) [149].

It is important to make sure we follow Apple's App Store review guidelines to maximize the chances of first-round approval. Apple rewrote these guidelines as of June 13, 2016 "to be more helpful, provide more context (...)"[49]. A comic book version is also available for more enjoyable reading [150].

6.3.1 Marketing

Increasing visibility in the App Store is a complex task that can involve thousands of euros spent on marketing. That being said, there are some ways to gain exposure without spending a lot including, but not limited to:

- Using Search Ads to promote the app. Apple provides developers with $100 credit for the first campaign [151]. It should be used to target functionally similar yet more

[8] Nonetheless, Apple says this metric is "subjective" because a high level of consumption may not translate directly to an inefficient app [148].

popular apps so when users search for them, the first app they see is the one we're promoting;

- Having great marketing materials (e.g. banners, screenshots) that increase the chance of an Apple feature[9]. A feature would result in a large influx of new users (depending on the app);
- Giving blogs free copies of an app in exchange for a brief review[10];
- Having a couple of friends or family post honest App Store reviews will slightly increase chances of new users signing up, as these help sway undecided people into downloading;
- Retaining current users: the key to success. It is crucial to keep users interested and answer their concerns and feedback in a prompt manner. Word of mouth travels fast; this is equally true for good and bad apps (but essentially the latter).

6.4 Summary

This chapter focused on the various testing mechanisms in iOS app development, how we integrated them into our development process and the deployment stage after that.

For unit testing, Xcode provides acceptable "out-of-the-box" tools for testing using the XCTest framework, but third-party ones like Quick and Nimble made writing unit tests clearer and with less verbosity. Couple this with Xcode's UI testing capabilities in Xcode 7, which make it possible to record and quickly play back simulations of user interface flows.

The beta-testing process included five testers who contributed valuable feedback to improve the app's user experience and fix bugs continuously. With their help, we the beta period has had seven betas (at the time of writing).

We also tested a series of metrics using Instruments (e.g. energy consumption, memory leaks, CPU usage) to ensure the app operates within reasonable levels. This profiling proved to be a crucial stage of asserting the quality our implementation and diagnosing potential issues which otherwise would not be encountered by unit or user interface tests alone.

Finally, we gave an overview of the importance of the App Store review guidelines, the necessity of an Apple Developer membership to submit apps to the Store, and how to boost an app's visibility via marketing and Search Ads.

[9] This is when Apple features an app in their "Noteworthy" section, and may include a banner at the top of its respective category.

[10] Note that sponsors can very expensive, depending on the website's reputation.

Chapter 7

Conclusion

This chapter summarizes our accomplished project goals after its six-month timespan. Conversely, we also present our shortcomings and planned future work. We conclude with a final appreciation statement.

7.1 Completion of objectives and requirements

To reiterate the first objective listed in the introduction, we wanted to understand the issues that cause (larger) companies during their apps' maintenance phase[1]. We covered this in Chapter 3 and concluded that a common pattern existed — out of the ten apps we analyzed, only six of them provided user onboarding, just under half (40%) exceeded energy consumptions, and over half (60%) explicitly disabled ATS. While 90% complied with Apple iOS HIG (YouTube was the single offender), we suspect this is due to Apple's strict enforcement.

The results do not bode well for users. With these popular apps having a diverse spectrum of issues, it is not surprising to see a majority ranked poorly on the App Store. An app with no user onboarding essentially leaves users to fend for themselves in an unknown environment. As one might deduce, does no wonders to improve a company or developer's reputation — and in turn, the user's experience.

7.1.1 Requirements completion

As for our app's functional requirements listed in Chapter 4, we managed to implement them all, even if some are still in an initial stage (e.g., sharing wish lists among users of the app which is limited to users of the app). Equally as important are our non-functional requirements, and while these are much harder to gauge, our decision to use Firebase helped meet some of the goals.

Google's platform provides scalability, reliability, resilience, rapid response times, and security. This leaves us to worry about platform compatibility, usability, privacy and pricing — which are linked to the client side of development and can be considered and improved upon independently.

[1] This is the phase where the app has shipped and typically receives regular updates to fix bugs, improve performance, and occasionally add new features. In some cases, it can last for decades

7.2 Limitations & future work

Snapwish's development from its initial idea to an App Store release to take six months (the timeframe of this project). However, as mentioned above, the app remains in beta. One factor that led to this was the amount of time invested in writing this dissertation, but the biggest was the lengthy testing process.

This process a significant amount of time — more so than the implementation phase — because it is a process constantly repeated with every iteration of the app (in our case, every beta seed). We must assert that all unit and user interface tests pass and that the profiling phase reports no abnormal results.

We believe that Snapwish could have made the App Store in early November of 2016 (provided it passed the App Store review process). Nonetheless, we will continue to develop our app and have updated its (new) target release date for Q1 of 2017. As for the roadmap, we plan on introducing the following new features (as well as improving existing ones):

- Adding items from the web (e.g., Amazon) in addition to manual input;
- Adding an activity tab to show recent activity for friends;
- Adding more information to profiles such as items and wish lists;
- Ability to share wish lists with people who do not have the Snapwish installed;
- More sorting and filtering options.

7.3 Final appreciation

Developing for iOS is not just about creating content and experiences for a platform or market, but also for its users. Through the App Store, one's work has the potential to influence the daily lives of millions. While the top 1% of apps continue to dominate almost 95% of the revenue (curiously, a reflection of the economies of most countries) [152], it is still possible to develop and enjoy moderate success for this platform. Finding a niche, defining the app's value proposition, and creating a business plan to test its financial viability is one way to approach this.

Knowing how to develop for iOS adds an advantage in a developer's skill set. Swift is still a language undergoing significant transformations as it progresses towards a stable ABI, so at this time there are no seasoned developers with decades of experience in this language (unlike Objective-C, which is over thirty years old). An eager developer with a reasonable object-oriented background who is new to iOS can learn Swift and become competitive in the workforce after a few years developing apps [153].

One of the major benefits of Apple's platforms — especially iOS — is the sheer amount of tools and resources on offer through which one can learn and employ in app development. Apple offers many excellent free resources to get started: the Swift Playgrounds app, WWDC videos, comprehensive documentation, and the Xcode IDE (with Instruments for profiling). Additionally, third-party support is extensive with libraries, frameworks, and components available for almost anything [39]. This ample support makes iOS a platform to consider developing for, personally and professionally.

That said, more important than the resources on offer is the development process *per se*[2]. One must consider good engineering practices from the initial idea to the time the app is seen as ready to ship (to the App Store). These practices include aspects such as answering engineering questions concerning project feasibility before even thinking about functional and non-functional requirements (i.e., the app's features and benefits).

To add to this, two factors at always in play: **time** and **money**. Developing an app takes a significant investment of both resources, so the lack of either can severely impact or even invalidate an end result. While ample time will likely result in a higher quality app, this proves to be untrue if the app's requirements are suited for a large team instead of a single individual (e.g., complex multiplayer games). Conversely, hiring more developers to complete an app's development in less time will often backfire[3].

Overall, this project gave us the opportunity to look into why some popular apps receive below average reviews and negative press from a user and developer standpoint. We were frequently frustrated with negative aspects such as excessive battery consumption or lack of user onboarding; judging by the reviews, we were not alone.

We lack insight on how the companies whose apps we analyzed work. The size of their development team or their methodology is also unknown to us, but we still believe they should increase efforts into making their apps better[4]. Furthermore, these companies whose apps represent the top 1% should not be given a "free pass" by Apple simply due to their popularity and installed user base. As long as this continues to happen, a vast majority of users will make do with "good enough".

7.4 Summary

This chapter drew conclusions on our work, from the app analysis to the entire development process of our app, Snapwish.

As far as the former, we managed to find offenses in every app we analyzed. We can almost certainly state that was it not for the apps' popularity and vast user base, many versions of these apps would not pass the App Store's review process. Unfortunately, user feedback and bad reviews by a small minority of their users are not yet enough to mitigate this issue.

The app analysis led to an identification of the criteria for Snapwish's tests: measure CPU performance, memory allocation, and energy consumption, just to name a few, to verify that our app was solely performing what it promised to do without abusing the device's resources.

In regards to the latter, we managed to develop a wish list management app that remains in the late beta stages. This may seem worrisome, but development requires a significant amount of time (if not more resources). It was tough to juggle beta tester feedback, and implementing and testing new features and bug fixes while having a set deadline for the submission of the complementary dissertation.

[2] Latin for "by itself" or "in itself".

[3] Productivity is not directly proportionate to human resources because developers first need to be acquainted with the inner workings of an app — and this takes time.

[4] One way is to listen to user feedback or implement public beta testing, as users will frequently find more bugs, provide suggestions and voice discontent.

Despite some shortcomings, we can answer this dissertation's main question that inquires if it is possible to develop an app with a reasonable timeframe that focuses on the user experience and respects good engineering practices. The answer is **yes**; it is possible.

A project's timeframe should be clearly defined. However, if developers are committed to creating apps that have their target users best interests in mind, they may experience that deadlines are constraints, forcing them into rushing things[5]. Regardless, the quality of the outcome will depend on a mix of good design and engineering practices, copious amounts of testing, and an unparalleled attention to detail. While it may not translate into guaranteed success on the App Store, it represents a surprisingly uncommon effort in the name of user experience; or, as Apple put it, in the name of "courage" [154].

[5]Naturally, this is easier when there aren't hard deadlines and stakeholders to answer to, so corporate environments remain outside the scope of this dissertation.

Bibliography

[1] Apple, Inc. *App Store - Support - Apple Developer*. Aug. 2016. url: https://developer.apple.com/support/app-store/.

[2] Apple, Inc. *WWDC 2016*. url: https://developer.apple.com/wwdc/.

[3] Jordan Golson. "Apple's App Store now has over 2 million apps". In: (June 2016). url: http://www.theverge.com/2016/6/13/11922926/apple-apps-2-million-wwdc-2016.

[4] Natasha Lomas. *App Monetization To Get Tougher Still, With Gartner Predicting 94.5% Of Downloads Will Be Free By 2017*. Jan. 2014. url: https://techcrunch.com/2014/01/13/making-apps-pay-gets-harder/.

[5] Frank Bi, James Bareham, and Michael Zelenko. *Life and death in the App Store*. Mar. 2016. url: http://www.theverge.com/2016/3/2/11140928/app-store-economy-apple-android-pixite-bankruptcy.

[6] S P Jobs et al. "Touch screen device, method, and graphical user interface for determining commands by applying heuristics". Jan. 2009. url: https://www.google.com/patents/US7479949.

[7] Statista. *Number of apps available in leading app stores*. June 2016. url: http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/.

[8] App Annie. *App Annie 2015 Retrospective*. Tech. rep. Jan. 2016. url: http://blog.appannie.com/app-annie-2015-retrospective/.

[9] Whitney Rhodes. *Why It Makes Sense to Update Your App by Launch Day*. 2016. url: http://savvyapps.com/blog/why-it-makes-sense-to-update-your-app-by-launch-day.

[10] Monkop. *9 Causes of Bad App Reviews*. June 2015. url: http://blog.monkop.com/post/120657007496/9-causes-of-bad-app-reviews.

[11] David Bolton. *The Average App Loses More Than 75% Of Its Users After One Day*. May 2016. url: https://arc.applause.com/2016/05/20/app-retention-rates-2016/.

[12] Romain Dillet. *Apple is going to remove abandoned apps from the App Store*. Sept. 2016. url: https://techcrunch.com/2016/09/01/apple-is-going-to-remove-abandoned-apps-from-the-app-store/.

[13] Kristina Bjoran. *User Expectations with Mobile Apps - Catching up with Effective UI*. Dec. 2010. url: http://www.uxbooth.com/articles/12207/.

[14] Matt Brian. *Mobile Apps: A look at what makes an app popular - TNW Mobile*. July 2011. url: http://thenextweb.com/mobile/2011/07/16/mobile-apps-a-look-at-what-makes-a-good-app-great/.

[15] Richard Paul, Robert Niewoehner, and Linda Elder. *The Thinker's Guide to Engineering Reasoning*. 2006. isbn: 978-0944583333. url: http://www.worldcat.org/title/thinkers-guide-library/oclc/729894075.

[16] *Value Proposition Definition*. url: http://www.investopedia.com/terms/v/valueproposition.asp.

[17] R S Kaplan and D P Norton. *Strategy Maps: Converting Intangible Assets Into Tangible Outcomes*. Harvard Business School Press. Harvard Business School Press, 2004. isbn: 978-1591391340. url: http://www.worldcat.org/title/strategy-maps-converting-intangible-assets-into-tangible-outcomes/oclc/53356641.

[18] Wolfgang Ulaga and Andreas Eggert. "Value-Based Differentiation in Business Relationships: Gaining and Sustaining Key Supplier Status". English. In: *Journal of Marketing* 70.1 (2006), pp. 119–136. doi: 10.1509/jmkg.2006.70.1.119. url: http://dx.doi.org/10.1509/jmkg.2006.70.1.119.

[19] Jelena Djurkic. *Three questions every value proposition must answer*. Nov. 2013. url: https://www.marsdd.com/news-and-insights/three-questions-every-value-proposition-must-answer/.

[20] T Woodall. "Conceptualising 'value for the customer': an attributional, structural and dispositional analysis". In: *Academy of marketing science review* (2003). url: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.195.8733&rep=rep1&type=pdf.

[21] Anna Mar. *7 Types of Negotiation And 1 Big Myth*. Feb. 2013. url: http://training.simplicable.com/training/new/7-types-of-negotiation-and-1-big-myth.

[22] Strategizer. *Business Model Canvas Explained*. Sept. 2011. url: https://www.youtube.com/watch?v=QoAOzMTLP5s.

[23] Michael A Trick. *Analytic Hierarchy Process*. Nov. 1996. url: http://mat.gsia.cmu.edu/classes/mstc/multiple/node4.html.

[24] Sam Rutherford and Alex H Cranz. *The iPhone 6s Is The World's Fastest Smartphone*. Oct. 2015. url: http://www.tomsguide.com/us/fastest-smartphone,review-2881.html.

[25] Oliver Haslam. *Detailed Benchmarks Show iPhone 6s Is The Fastest Phone In The World | Redmond Pie*. Sept. 2015. url: http://www.redmondpie.com/detailed-benchmarks-show-iphone-6s-is-the-fastest-phone-in-the-world/.

[26] Apple, Inc. *iPad Air 2 - Performance - Apple*. url: http://www.apple.com/ipad-air-2/performance/.

[27] Apple, Inc. *iPad Pro - Technology - Apple*. url: http://www.apple.com/ipad-pro/technology/.

[28] Nino S Vitale. *Apple's Touch ID fingerprint scanner: A Study of Functionality, Usability & Security Concerns*. Tech. rep. Nov. 2013.

[29] Federico Viticci. *iPhone 5s: Our Complete Overview*. Sept. 2013. url: https://www.macstories.net/stories/iphone-5s-our-complete-overview/.

[30] Josh Tyrangiel. *Inside the Design Labs Where the iPhone's Coolest New Feature Was Built*. Sept. 2015. url: http://www.bloomberg.com/features/2015-how-apple-built-3d-touch-iphone-6s/.

[31] Apple, Inc. *iPhone 6s - 3D Touch - Apple*. url: http://www.apple.com/iphone-6s/3d-touch/.

[32] James Vincent. *Apple says iOS 9 adoption is 'fastest ever,' with 50 percent of devices upgraded*. Sept. 2015. url: http://www.theverge.com/2015/9/21/9364169/ios-9-adoption-fastest-ever.

[33] Apple, Inc. *iOS 10*. url: http://www.apple.com/ios/ios10-preview/.

[34] Apple, Inc. *Swift - Apple Developer*. url: https://developer.apple.com/swift/.

[35] Apple, Inc. *Swift is Open Source - Swift Blog*. Dec. 2015. url: https://developer.apple.com/swift/blog/?id=34.

[36] Apple, Inc. *Swift.org*. 2015. url: https://swift.org/.

[37] Apple, Inc. *Apple Github Repositories*. Dec. 2015. url: https://github.com/apple.

[38] Apple, Inc. *Using Swift with Cocoa and Objective-C (Swift 2.1): Working with Cocoa Data Types*. url: https://developer.apple.com/library/ios/documentation/Swift/Conceptual/BuildingCocoaApps/WorkingWithCocoaDataTypes.html#//apple_ref/doc/uid/TP40014216-CH6-ID79.

[39] Vinicius Souza. *Awesome iOS | Github*. url: https://github.com/vsouza/awesome-ios#server.

[40] Ian Fette and Alexey Melnikov. *The WebSocket Protocol*. Dec. 2011. url: http://tools.ietf.org/html/rfc6455.

[41] Benedict Evans. *Market shares and ecosystem value*. June 2014. url: http://benevans.com/benedictevans/2014/6/25/market-shares.

[42] Keith Griffith and John Heggestuen. *Apple's Astronomical 800 Million iTunes Accounts Could Give It A Huge Advantage In Payments*. Apr. 2014. url: http://www.businessinsider.com/apples-astronomical-800-million-itunes-accounts-could-give-it-a-huge-advantage-in-payments-2014-4.

[43] *Apple App Store: global category revenue share per business model 2014*. Tech. rep. Apr. 2014. url: http://www.statista.com/statistics/283753/apple-app-store-global-category-revenue-share-per-business-model/.

[44] WRIGHTLABS, LLC. "WishMindr - Create & share wish lists for any occasion". In: *App Store* (). url: https://itunes.apple.com/us/app/wishmindr-create-share-wish/id972951070?mt=8.

[45] UserOnboard. *User Onboarding | A frequently-updated compendium of web app first-run experiences*. url: http://www.useronboard.com/.

[46] MyGiftster Corporation. "Giftster - wish list registry for holiday, birthday, baby". In: *App Store* (). url: https://itunes.apple.com/us/app/giftster-wish-list-registry/id478126039?mt=8.

[47] Giftry LLC. "Giftry – Universal Wish List and Gift Registry". In: *App Store* (). url: https://itunes.apple.com/us/app/giftry-universal-wish-list/id944005342?mt=8.

[48] Apple, Inc. *iOS Human Interface Guidelines*. url: https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/MobileHIG/.

[49] Apple, Inc. *App Store Review Guidelines*. url: https://developer.apple.com/app-store/review/guidelines/.

[50] Apple, Inc. *iOS Human Interface Guidelines: iOS App Anatomy*. url: https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/MobileHIG/Anatomy.html#//apple_ref/doc/uid/TP40006556-CH24-SW1.

[51] Apple, Inc. *iOS Human Interface Guidelines: Color and Typography*. url: https://developer.apple.com/library/ios/documentation/

UserExperience/Conceptual/MobileHIG/ColorImagesText.html#//apple_ref/doc/uid/TP40006556-CH58-SW1.

[52] Apple, Inc. *Cocoa Keys*. url: https://developer.apple.com/library/ios/documentation/General/Reference/InfoPlistKeyReference/Articles/CocoaKeys.html#//apple_ref/doc/uid/TP40009251-SW33.

[53] Sebastian Düvel. *Facebook app is draining your iPhone's battery*. Nov. 2013. url: https://blog.hagga.net/archives/iphone-ipod-touch/3805-facebook-app-is-draining-your-iphones-battery.

[54] Federico Viticci. *The Background Data and Battery Usage of Facebook's iOS App*. Oct. 2015. url: https://www.macstories.net/linked/the-background-data-and-battery-usage-of-facebooks-ios-app/.

[55] *Facebook on the App Store*. url: https://itunes.apple.com/us/app/facebook/id284882215?mt=8.

[56] Samuel Gibbs. "Uninstalling Facebook app saves up to 15% of iPhone battery life | Technology". In: *The Guardian* (Feb. 2016). url: https://www.theguardian.com/technology/2016/feb/08/uninstalling-facebook-app-saves-iphone-battery-life?CMP=Share_iOSApp_Other.

[57] Chris Dzombak. *Nobody is using App Transport Security; what's next?* Sept. 2015. url: https://www.dzombak.com/blog/2015/09/Nobody-is-using-App-Transport-Security--what-s-next-.html.

[58] Facebook. *Data Policy*. url: https://www.facebook.com/policy.php.

[59] Sarah Perez. *Users Pummel YouTube's iOS Update With One-Star Reviews*. Oct. 2015. url: http://techcrunch.com/2015/10/06/users-pummel-youtubes-ios-update-with-one-star-reviews/.

[60] *YouTube on the App Store*. url: https://itunes.apple.com/us/app/youtube/id544007664?mt=8&ign-mpt=uo%3D2.

[61] Google. *Privacy Policy – Privacy & Terms – Google*. url: https://www.google.com/intl/en/policies/privacy/.

[62] Kat. *Building Dropbox's New User Experience for Mobile, Part 1*. Aug. 2014. url: https://blogs.dropbox.com/tech/2014/08/building-dropboxs-new-user-experience-for-mobile-part-1/.

[63] *Dropbox on the App Store*. url: https://itunes.apple.com/us/app/dropbox/id327630330?mt=8&ign-mpt=uo%3D2.

[64] Anthony Ha. *Edward Snowden's Privacy Tips: "Get Rid Of Dropbox," Avoid Facebook And Google*. Oct. 2014. url: http://techcrunch.com/2014/10/11/edward-snowden-new-yorker-festival/.

[65] "How Snapchat Onboards New Users | User Onboarding". In: (). url: https://www.useronboard.com/how-snapchat-onboards-new-users/?slide=2.

[66] *Snapchat on the App Store*. url: https://itunes.apple.com/us/app/snapchat/id447188370?mt=8&ign-mpt=uo%3D2.

[67] Alex Heath. *Try this to keep Snapchat from destroying your battery life*. May 2016. url: http://www.techinsider.io/keep-snapchat-from-draining-battery-life-2016-5.

[68] *App Transport Security Talk*. Jan. 2016. url: https://github.com/cdzombak/app-transport-security-talk-jan-2016/blob/master/App%20Transport%20Security%20-%20What,%20Why,%20How.md.

[69] UX Archive. *Facebook Messenger*. url: http://uxarchive.com/apps/facebook_messenger.

[70] *Messenger on the App Store*. url: https://itunes.apple.com/us/app/messenger/id454638411?mt=8.

[71] *Amazon App: shop, scan, compare, and read reviews on the App Store*. url: https://itunes.apple.com/us/app/amazon-app-shop-scan-compare/id297606951?mt=8&ign-mpt=uo%3D2.

[72] Amazon, Inc. *Amazon.com Help: Amazon.com Privacy Notice*. url: http://www.amazon.com/gp/help/customer/display.html?nodeId=468496.

[73] UX Archive. *Twitter*. url: http://uxarchive.com/apps/twitter.

[74] *Twitter on the App Store*. url: https://itunes.apple.com/us/app/twitter/id333903271?mt=8&ign-mpt=uo%3D2.

[75] Philip Bates. *Avoid These iPhone Apps for Better Battery Life*. Feb. 2016. url: http://www.makeuseof.com/tag/avoid-iphone-apps-better-battery-life/.

[76] Twitter, Inc. *Privacy Policy*. url: https://twitter.com/privacy?lang=en.

[77] Statista. *WhatsApp: number of monthly active users 2013-2016*. Tech. rep. 2016. url: http://www.statista.com/statistics/260819/number-of-monthly-active-whatsapp-users/.

[78] *WhatsApp Messenger on the App Store*. url: https://itunes.apple.com/us/app/whatsapp-messenger/id310633997?mt=8&ign-mpt=uo%3D2.

[79] Cody Lee. *WhatsApp iOS 7 update is out with new design, broadcast lists and more*. Dec. 2013. url: http://www.idownloadblog.com/2013/12/02/whatsapp-ios-7-app-update-out/.

[80] Cody Lee. *WhatsApp updated with support for iPhone 6 and 6 Plus*. Nov. 2014. url: http://www.idownloadblog.com/2014/11/17/whatsapp-updated-with-support-for-iphone-6-and-6-plus/.

[81] WhatsApp. *WhatsApp Security White Paper*. Tech. rep. url: https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf.

[82] Nate Cardozo, Kurl Opsahl, and Rainey Reitman. *Online Service Providers' Privacy and Transparency Practices Regarding Government Access to User Data*. Tech. rep. June 2015. url: https://www.eff.org/who-has-your-back-government-data-requests-2015#whatsapp-report.

[83] Evelyn M Rusli, Hannah Karp, and Douglas Macmillan. "SoundCloud's Valuation Could Top $1.2 Billion With New Fundraising". In: *The Wall Street Journal* (Dec. 2014). url: http://blogs.wsj.com/digits/2014/12/09/soundclouds-valuation-could-top-1-2-billion-with-new-fundraising/.

[84] Benjamin Libor. *Soundcloud - Onboarding (iOS 3.6.0)*. url: https://www.youtube.com/watch?v=HhIVkN_X7Ac.

[85] *SoundCloud - Music & Audio on the App Store*. url: https://itunes.apple.com/us/app/soundcloud-music-audio/id336353151?mt=8&ign-mpt=uo%3D2.

[86] SoundCloud. *Privacy Policy on SoundCloud*. url: https://soundcloud.com/pages/privacy.

[87] SoundCloud. *Terms of Use on SoundCloud*. url: https://soundcloud.com/terms-of-use.

[88] *NFL Fantasy Football - Official NFL Fantasy App on the App Store*. url: https://itunes.apple.com/us/app/nfl-fantasy-football-official/id876054082?mt=8.

[89] NFL. *Privacy Policy*. url: http://www.nfl.com/help/privacy.

[90] Kate Conger. *Apple will require HTTPS connections for iOS apps by the end of 2016*. June 2016. url: http://techcrunch.com/2016/06/14/apple-will-require-https-connections-for-ios-apps-by-the-end-of-2016/.

[91] Apple, Inc. *The Swift Programming Language (Swift 2.2): About Swift*. url: https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/index.html#//apple_ref/doc/uid/TP40014097-CH3-XID_0.

[92] *Is Swift a dynamic or static language?* Apr. 2015. url: http://stackoverflow.com/questions/29924477/is-swift-a-dynamic-or-static-language.

[93] Ben Sandofsky. *Why big apps aren't moving to Swift (Yet)*. May 2016. url: https://medium.com/@sandofsky/why-big-apps-arent-moving-to-swift-yet-f8e9a89ef661#.mk7n455he.

[94] Ryan Olson. *Are the Top Apps Using Swift?* Jan. 2016. url: https://medium.com/@ryanolsonk/are-the-top-apps-using-swift-42e880e7727f#.f3jhsohzs.

[95] Sergio De Simone. *Swift 3 Will Not Have a Stable ABI*. May 2016. url: https://www.infoq.com/news/2016/05/swift-3-no-stable-abi.

[96] Maria Kuz. *7 Advantages of Using Swift Over Objective-C*. Mar. 2016. url: http://mlsdev.com/en/blog/51-7-advantages-of-using-swift-over-objective-c.

[97] Kevin Lacker. *Moving On*. Jan. 2016. url: http://blog.parse.com/announcements/moving-on/.

[98] Pauli Olavi Ojala. *Facebook's Parse shutdown has a lesson to all tech customers*. Jan. 2016. url: https://medium.com/swlh/facebook-s-parse-shutdown-has-a-lesson-to-all-tech-customers-ecc43a83e36b.

[99] Related Code. *relatedcode/ParseAlternatives*. url: https://github.com/relatedcode/ParseAlternatives.

[100] Apple, Inc. *Model-View-Controller*. url: https://developer.apple.com/library/ios/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html.

[101] Stanford University. *Developing iOS 8 Apps with Swift*. 2015. url: https://itunes.apple.com/gb/course/developing-ios-8-apps-swift/id961180099.

[102] Github. *Quick/Quick*. url: https://github.com/Quick/Quick.

[103] Marco Pierre White et al. *MasterChef Australia (Season 6, Episode 29)*. June 2014. url: http://www.imdb.com/title/tt3774896/?ref_=nm_flmg_slf_1.

[104] Apple, Inc. *Swift API Design Guidelines*. 2016. url: https://swift.org/documentation/api-design-guidelines/.

[105] Apple, Inc. *Swift API Design Guidelines - Apple Developer*. June 2016. url: https://developer.apple.com/videos/play/wwdc2016/403/.

[106] Nick Babich. *Little Big Details For Your Mobile App*. June 2016. url: http://babich.biz/little-big-details-for-your-mobile-app/?utm_campaign=iOS%2BDev%2BWeekly&utm_medium=web&utm_source=iOS_Dev_Weekly_Issue_266.

[107] Apple, Inc. *UI Design Do's and Don'ts*. url: https://developer.apple.com/design/tips/.

[108] LINITIX. *Empty States : Serist App*. url: http://emptystat.es/image/149836721051.

[109] Azendoo. *Empty States : Azendoo App*. url: http://emptystat.es/image/134919332373.
[110] Apple, Inc. *Building Better Apps with Value Types in Swift*. June 2015. url: https://developer.apple.com/videos/play/wwdc2015/414/.
[111] Apple, Inc. *Value and Reference Types*. Aug. 2014. url: https://developer.apple.com/swift/blog/?id=10.
[112] Firebase. *Firebase Realtime Database*. url: https://firebase.google.com/docs/database/.
[113] Firebase. *Structure Data*. url: https://firebase.google.com/docs/database/ios/structure-data.
[114] Firebase. *Save Data on iOS*. url: https://firebase.google.com/docs/database/ios/save-data.
[115] Firebase. *Retrieve Data on iOS*. url: https://firebase.google.com/docs/database/ios/retrieve-data.
[116] David East. *Best practices for the iOS UIViewController and Firebase*. Oct. 2015. url: https://firebase.googleblog.com/2015/10/best-practices-for-ios-uiviewcontroller_6.html.
[117] Firebase. *Understand Firebase Realtime Database Rules*. url: https://firebase.google.com/docs/database/security/.
[118] Wendy Lu. *Immutable models and data consistency in our iOS App*. Aug. 2016. url: https://engineering.pinterest.com/blog/immutable-models-and-data-consistency-our-ios-app.
[119] Apple, Inc. *View Controller Programming Guide for iOS: The Role of View Controllers*. url: https://developer.apple.com/library/content/featuredarticles/ViewControllerPGforiPhoneOS/.
[120] Apple, Inc. *UICollectionView - UIKit*. url: https://developer.apple.com/reference/uikit/uicollectionview.
[121] Firebase. *Users in Firebase Projects*. url: https://firebase.google.com/docs/auth/users.
[122] Apple, Inc. *UIApplicationDelegate - UIKit | Apple Developer Documentation*. url: https://developer.apple.com/reference/uikit/uiapplicationdelegate.
[123] Daniel Langh et al. *Skyscanner/SkyFloatingLabelTextField*. Swift. url: https://github.com/Skyscanner/SkyFloatingLabelTextField.
[124] Dario Pellegrini. *dariopellegrini/DPRadialMenu*. url: https://github.com/dariopellegrini/DPRadialMenu.
[125] Mazyod. *telly/TLYShyNavBar*. url: https://github.com/telly/TLYShyNavBar.
[126] Joakim Gyllström. *mikaoj/BSImagePicker*. url: https://github.com/mikaoj/BSImagePicker.
[127] Apple, Inc. *Photos | Apple Developer Documentation*. url: https://developer.apple.com/reference/photos.
[128] Dalton Cherry. *Shaving Our Image Size*. Apr. 2016. url: http://engineering.dollarshaveclub.com/shaving-our-image-size/.
[129] Yaoyuan. *ibireme/YYImage*. url: https://github.com/ibireme/YYImage.
[130] Apple, Inc. *UIKit Functions - UIKit*. url: https://developer.apple.com/reference/uikit/2009858-uikit_functions.
[131] Firebase. *Create a Storage Reference on iOS*. url: https://firebase.google.com/docs/storage/ios/create-reference.

[132] Apple, Inc. *NSData - Foundation*. url: https://developer.apple.com/reference/foundation/nsdata.
[133] Dean Brindley. *Create The Twitter iOS App User Interface*. Apr. 2015. url: http://developerdean.com/create-twitter-ios-app-user-interface/.
[134] Nick Lockwood. *nicklockwood/FXBlurView*. url: https://github.com/nicklockwood/FXBlurView.
[135] Vicc Alexander. *ViccAlexander/Chameleon*. url: https://github.com/ViccAlexander/Chameleon.
[136] FlatUIColors. *Flat UI Colors*. url: https://flatuicolors.com/.
[137] Tim Oliver. *TimOliver/TOCropViewController*. url: https://github.com/TimOliver/TOCropViewController.
[138] Apple, Inc. *Dispatch*. url: https://developer.apple.com/reference/dispatch.
[139] Apple, Inc. *Testing in Xcode 6*. url: https://developer.apple.com/videos/play/wwdc2014/414/.
[140] Apple, Inc. *XCTest*. url: https://developer.apple.com/reference/xctest.
[141] *kiwi-bdd/Kiwi*. url: https://github.com/kiwi-bdd/Kiwi.
[142] *Quick/Nimble*. url: https://github.com/Quick/Nimble.
[143] Apple, Inc. *UI Testing in Xcode*. url: https://developer.apple.com/videos/play/wwdc2015/406/.
[144] Vadym Markov. *vadymmarkov/Fakery*. url: https://github.com/vadymmarkov/Fakery/.
[145] Apple, Inc. *TestFlight Beta Testing*. url: https://developer.apple.com/testflight/.
[146] James Frost and Matt Galloway. *Instruments Tutorial with Swift: Getting Started*. May 2015. url: https://www.raywenderlich.com/97886/instruments-tutorial-with-swift-getting-started.
[147] Kevin Kazmierczak. *What every iOS Developer Should Be Doing with Instruments - Universal Mind*. Feb. 2016. url: http://www.universalmind.com/blog/what-every-ios-developer-should-be-doing-with-instruments/.
[148] Apple, Inc. *Instruments User Guide: Target Devices and Processes*. url: https://developer.apple.com/library/content/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/WorkingwithTargets.html.
[149] Apple, Inc. *Advanced App Capabilities*. url: https://developer.apple.com/support/app-capabilities/.
[150] Matt Simmons. *App Review Guidelines: A Comic Book*. 2016. url: https://devimages.apple.com.edgekey.net/app-store/review/guidelines/App-Review-Guidelines-The-Comic-Book.pdf.
[151] Apple, Inc. *Search Ads*. url: https://developer.apple.com/app-store/search-ads/.
[152] AppleInsider. *Occupy the App Store? Top 1% of monetized apps dominate 94% of US App Store revenue*. url: http://appleinsider.com/articles/16/05/11/occupy-the-app-store-top-1-of-monetized-apps-dominate-94-of-us-app-store-revenue.
[153] Rosie Allabarton. *Why Should I Learn Swift?* July 2016. url: http://blog.careerfoundry.com/ios-development/why-should-i-learn-swift.

[154] Nick Statt. *Apple says it took 'courage' to remove the headphone jack on the iPhone 7*. Sept. 2016. url: http://www.theverge.com/2016/9/7/12838024/apple-iphone-7-plus-headphone-jack-removal-courage.

Appendix A

Business Model Canvas

A.1 Business Model Canvas