

MSc in Informatics Engineering
Internship
Final Report

Platform of Advertising and Push Notifications for Mobile Apps

Francisco José da Cruz Gameiro

francisco.gameiro@wit-software.com

fjcg@student.dei.uc.pt

WIT Software Supervisor:

Eng. Filipe Cardeal Patrão Freitas Dos Santos

DEI Supervisor:

Prof. Dr. Luís Miguel Machado Lopes Macedo



FCTUC DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

This page was intentionally left in blank

Abstract

Since the beginnings of mankind, communication has been a fundamental component in our interpersonal relationships. Initially with rudimentary characteristics, it evolved becoming more complex, and leading to the creation of new ways of communication in response to the needs of the modern society. The space and time were overcome by creativity and technology. The Global Village experiences a real input with the creation/introduction of the mobile devices.

Nowadays, these devices are part of our quotidian and represent an indispensable element of our *modus vivendi*. With the introduction of Internet use on these gadgets, infinite possibilities arose, potentiating the availability and the access to a variety of services.

Communication services emerged, bringing people together and strengthening their relations. Among the various possibilities of communication, push notifications are the phenomenon of mobile technology. Advertisers, marketers and mobile developers see it as a key way to reach audiences and deliver information.

The solution here presented is a gateway that delivers push notifications to iOS and Android users through the integration of Apple Push Notification Service and Google Cloud Messaging. It is possible to create notification campaigns and target different types of audiences, like a specific segment of users, based on a set of defined rules. There is also a statistics and report module where the client can acknowledge, for example, if the push notifications sent were opened. To interact with these different features it was developed an API for expert clients, and a web page for a user-friendlier synergy.

Desde os primórdios da humanidade que a comunicação tem sido uma componente fundamental nas relações interpessoais. Inicialmente com características rudimentares, evoluiu, tornando-se cada vez mais complexa levando à criação de novas formas de comunicação como resposta às necessidades da sociedade moderna. O espaço e o tempo foram superados pela criatividade e pela tecnologia. A Aldeia Global experienciou um verdadeiro *input* com a criação / introdução dos dispositivos móveis.

Hoje em dia, esses dispositivos fazem parte do nosso cotidiano e simbolizam um elemento indispensável do nosso *modus vivendi*. Com a introdução do uso da Internet nestes gadgets, inúmeras possibilidades surgiram, potencializando a disponibilidade e o acesso a uma panóplia de serviços.

Serviços de comunicação surgiram, aproximando as pessoas e fortalecendo as suas relações. Entre as diferentes formas de comunicação existentes, as *push notifications* são o fenômeno da tecnologia móvel. Anunciantes, comerciantes e programadores de dispositivos móveis vêm-nas como uma forma essencial e estratégica para atingir o público e distribuir informações.

A solução aqui apresentada é um *gateway* que entrega *push notifications* a utilizadores de iOS e Android através da integração da Apple Push Notification Service e Google Cloud Messaging. Possibilita a criação de campanhas publicitárias e alcançar diferentes tipos de público, como um segmento específico de utilizadores, através do estabelecimento de um conjunto de regras pré-definidas. Também possui um módulo de estatísticas e relatórios, através do qual o cliente pode ter acesso a informações como, por exemplo, se as *push notifications* enviadas foram abertas. Para interagir com as diferentes funcionalidades disponibilizadas, foi desenvolvida uma API para clientes mais tecnologicamente diferenciados, e uma página web promotora de uma amigável sinergia com o utilizador.

This page was intentionally left in blank

Keywords

“Push Notifications”, “Push Notification Services”, “Push Messaging”, “Rich Communication Services”, “Over the Top applications”, “Application Programming Interface”, “Push Notifications Gateway”.

This page was intentionally left in blank

Table of Contents

1. Introduction	15
1.1. The company	15
1.2. Context.....	15
1.3. Motivation.....	16
1.4. Goals	16
1.5. Document Structure	17
2. State of the art.....	19
2.1. Push Messaging Solutions	19
2.1.1. Proprietary solutions	19
2.1.2. 3 rd Party solutions	21
2.1.2.1. Brief description.....	23
2.1.2.2. Features comparison.....	26
3. Requirements.....	29
3.1. Priorities.....	29
3.2. Types	29
3.3. Functional Requirements.....	30
3.3.1. Server.....	30
3.3.2. Device.....	32
3.3.3. Web UI.....	32
3.4. Non-Functional Requirements.....	34
3.4.1. Web UI.....	34
3.4.2. Devices	34
3.4.3. Server.....	35
4. Architecture	37
4.1. REST API Definition	38
4.1.1. Generic API.....	38
4.1.2. Telco API.....	38
4.1.3. Client/Network Operator API.....	39
4.2. Device Libraries	40
4.2.1. iOS Generic Framework & Android Library.....	40
4.2.2. iOS Telco Framework.....	41
4.3. Server	43
4.3.1. Components	43
4.3.1.1. REST API	44
4.3.1.2. Jobs.....	44
4.3.1.3. RabbitMQ.....	46
4.3.1.4. Consumer Threads.....	48
4.3.1.5. Database	49
4.3.1.5.1. ER Model.....	49
4.3.1.5.2. PostGIS.....	50
4.3.1.5.3. Dataset of countries and cities	51
4.3.1.5.4. Security	51
4.3.1.6. Quartz Scheduler.....	52
4.3.1.6.1. PushToRabbitMQJob.....	52
4.3.1.6.2. LoadScheduledJobs	53
4.3.1.6.3. AppleFeedbackService.....	55
4.3.1.7. Push Engine.....	56
4.3.1.7.1. Apple Engine	57
4.3.1.7.1.1. Multithreaded Engine.....	57
4.3.1.7.1.2. Binary Interface compliant message composition and delivery	58
4.3.1.7.1.3. Failed notifications processing.....	59
4.3.1.7.2. Google Engine	60
4.3.1.8. Web Application.....	61
4.3.2. Container.....	64
4.3.2.1. Initialization	64
4.3.2.2. Configuration.....	65

5. Software Development Methodology	69
5.1. Agile Principles	69
5.2. Scrum Roles	69
5.3. How it works	70
5.4. Planning	71
5.5. Risk Management	73
6. Software Quality	77
6.1. Functional Tests	77
6.1.1. Acceptance tests	77
6.1.2. API Tests	77
6.1.3. Regression tests	78
6.2. Quality tests	79
6.2.1. Software Performance	79
6.2.1.1. Web Application	79
6.2.1.1.1. YSlow Rules	79
6.2.1.1.2. HTTP Requests	80
6.2.1.1.3. Page Load Times	81
6.2.1.1.4. Benchmarking	82
6.2.1.2. RabbitMQ Benchmarking	84
6.2.1.2.1. Publishing	84
6.2.1.2.2. Consuming (delivering)	85
6.2.1.2.3. Publishing and delivering simultaneously	85
6.2.1.3. Push Notifications Benchmarking	87
6.2.1.3.1. Apple Engine	88
6.2.1.3.2. Google Engine	89
6.2.2. Usability tests	90
6.3. Static program analysis	93
6.3.1. Javadoc	93
6.3.2. CodePro Analytix	93
6.3.2.1. Code Audit	93
6.3.2.2. Code Metrics	93
6.3.2.2.1. Server	94
6.3.2.2.2. Android SDK	94
6.3.2.2.3. Java API SDK	94
6.3.2.3. Code Dependency Analysis	95
6.3.2.3.1. Server	95
6.3.2.3.2. Android SDK	96
6.3.2.3.3. Java API SDK	96
6.3.3. Xcode Statistician	97
7. Conclusion	99
7.1. Work done	99
7.2. Contributions	101
7.3. Deviations from the initial plan	101
7.4. Future work	102
7.5. Final Thoughts	103

Index of Tables

Table 1 - Push.io available Plans [8]	23
Table 2 - Parse available plans [10]	24
Table 3 - Urban Airship available plans [12][13]	24
Table 4 - Xtify available plans [15][16]	25
Table 5 - Features comparison of the Push Messaging Services	27
Table 6 - Requirements' priorities	29
Table 7 - Requirements' types	29
Table 8 - Generic API Definition	38
Table 9 - Telco API Definition	38
Table 10 - Client API operations	39
Table 11 - Job Status and related actions	45
Table 12 - JSPs and description	62
Table 13 - Global JavaScript files and description	63
Table 14 - Definition of Done	70
Table 15 - Risks Impact Classification	73
Table 16 - Acceptance tests result	77
Table 17 - API Test results resume	78
Table 18 - YSlow rules evaluation	80
Table 19 - Pages empty vs primed cache load times	81
Table 20 - Usability test reformulated task result	92
Table 21 - Code Audit Results	93
Table 22 - Server code metrics	94
Table 23 - Android SDK code metrics	94
Table 24 - Java API SDK code metrics	95
Table 25 - iOS generic framework statistics by Xcode Statistician	97

Index of Figures

Figure 1 - Push Notification flow in iOS [3]	19
Figure 2 - Android Push Notification Flow [4]	20
Figure 3 - Flow of a Push Notification on Windows Phone [5]	21
Figure 4 - High-level design	37
Figure 5 - Android Library Register Request	40
Figure 6 - iOS Telco Framework Push Notification Use Case	41
Figure 7 - Global Architecture Components	43
Figure 8 - Job Status Lifecycle	45
Figure 9 - RabbitMQ Asynchronous Job Architecture	47
Figure 10 - RabbitMQ Synchronous Job Architecture	47
Figure 11 - Database ER Model	49
Figure 12 - LoadScheduledJobs algorithm overview	53
Figure 13 - Binary format of a feedback tuple	55
Figure 14 - APIPushJob decisions	56
Figure 15 - Optimized Thread and Connection Number	58
Figure 16 - Enhanced Notification Format	59
Figure 17 - Error-response packet	60
Figure 18 - Web App Project Structure	62
Figure 19 - Scrum Overview [27]	71
Figure 20 - First Semester Gantt diagram	73
Figure 21 - Second Semester Gantt diagram	73
Figure 22 - YSlow Index page score	80
Figure 23 - YSlow dashboard page score	80
Figure 24 - Index page empty vs primed cache	81
Figure 25 - Dashboard page empty vs primed cache	81
Figure 26 - RabbitMQ' Flow Control with prefetch count of 30	86
Figure 27 - Server dependency analysis	95
Figure 28 - Android SDK dependency analysis	96
Figure 29 - Java API SDK dependency analysis	97

This page was intentionally left in blank

Index of Charts

Chart 1 – Index website page relation of throughput with request concurrency	82
Chart 2 – Index website page relation of failed requests with concurrency	83
Chart 3 – RabbitMQ Job publishing throughput.....	84
Chart 4 - RabbitMQ Job delivery throughput according to number of consumers	85
Chart 5 - RabbitMQ Job publishing + delivering throughput according to QoS Value	86
Chart 6 – Throughput of push requests to iOS devices	88
Chart 7 - Throughput of push requests to Android devices	89

This page was intentionally left in blank

Acronyms

API	Application Programming Interface
APNS	Apple Push Notifications Service
FR	Functional Requirement
GCM	Google Cloud Messaging
GSMA	GSM Association
HTTPS	Hypertext Transfer Protocol Secure
IPTV	Internet Protocol television
JSON	JavaScript Object Notation
MNO	Mobile Network Operator
MPNS	Microsoft Push Notifications Service
NFR	Non-Functional Requirement
OTT	Over-the-top
PN	Push Notification
PNG	Push Notifications Gateway
RCS	Rich Communications Suite
REST	Representational State Transfer
SDK	Software Development Kit
SSL	Secure Sockets Layer
TLS	Transport Layer Security
UI	User Interface
URI	Uniform Resource Identifier
WCS	WIT Communications Suite
XML	Extensible Markup Language
XMPP	Extensible Messaging and Presence Protocol

Glossary

Java	Object-oriented programming language.
joyn	A certification trademark of GSMA
Product Backlog	Artifact of SCRUM that represents a list of requirements for the product.
SCRUM	Agile software development framework
Store & Forward	Station that keeps a message when its delivery was not possible.
Wi-Fi	Radio wave technology that allows wireless internet access.

This page was intentionally left in blank

1. Introduction

The internship is part of the Thesis/Internship of the MSc in Informatics Engineer of the Faculty of Science and Technology of the University of Coimbra. This document has the purpose of presenting the work developed during the internship as well as the knowledge acquired along the way. It was entirely done in WIT Software's premises in Coimbra, from October 2013 to June 2014, having as mentors, Dr. Luís Macedo (DEI) and Eng. Filipe Santos (WIT Software).

1.1. The company

WIT Software is a company, which started as a spinoff of the University of Coimbra (UC) back in March 2001. Its headquarters were initially in Pedro Nunes Institute until 2004 when the capacity maxed out, and the team had to move to another place. From then until December 2013, the company grew substantially and had to find a new home again, this time in the building *Centro de Empresas de Taveiro*. Today the company has offices in Coimbra, Porto, Lisboa, Leiria, Silicon Valley (California, USA) and Reading (UK), which sums more than 160 employees.

WIT Software has business in more than 30 countries and is organized in three different business units:

- Telco

This unit develops software for mobile network operators and has clients in Europe, South Africa and South America. One of the main products, which have been boosting the exportation to Europe in a large percentage, is called WIT Communications Suite (WCS), which enables the convergence of mobile, landline and Internet communications.

- Mobile

This unit develops software for mobile devices, mainly: iPhone/iPad, Android, Blackberry and Windows Phone. It has clients such as: Millennium BCP, ControlInveste, Vodafone Group, EDP, among others.

- TV

This unit has a main product called "WIT Connected TV Suite" which allows IPTV operators to offer new contents to their users such as: social platforms integration, Video-on-demand, content share from the smartphone to the TV and smartphone apps for TV remote control.

1.2. Context

From day to day, the world of telecommunications becomes more and more competitive, which forces network operators to offer new and innovative services with lower prices than their competitors.

As a result of the rising penetration of smartphones and application stores in our quotidian, network operators have the need to not lose the pace and to be able to keep up with different free alternatives offers.

Regarding the most recent threats, OTT (Over-the-Top) applications arise. They are being developed by third-party companies, not related to mobile operators, and can be installed on our smartphone devices, delivering ways to communicate with people. Since they are in general free and make use of the internet data plan subscribed by the users, its effect is very negative in the revenue of mobile networks, letting the OTT companies gain ground to proliferate.

In 2012, the number of SMS sent globally reached 8600 billions [1], while the number of messages exchanged between OTT applications reached 5846 billions [1]. This difference has been declining with time, being expected in 2016 the number of messages between OTTs to double the number of SMS sent.

Therefore, to overcome this threat, GSM Association (GSMA) created an initiative called Rich Communications Ecosystem which has the main goal to strengthen mobile operators in the global market, offering an innovative way for its clients to access multimedia communications which are able to expand and improve their mobile interactions. This initiative has a base specification nominated RCS (Rich Communication Services), which is currently in version 5.1 and defines the services being standardized now.

These services are contained in RCS clients who after being accredited by GSMA become a product with the joyn brand. Belonging to this brand, there's a guarantee that they are in conformity with the guidelines defined as well as its interoperability between different RCS clients/devices and mobile operators, which represents an advantage towards OTT services in the market.

Thus, joyn assumes itself as a global telecommunications initiative which has the goal to integrate on a unique interface, voice and video communication on top of the IP protocol, instant messaging, file transfer and much more, depending on the ambition of the product developers.

This is where the context of my internship started, focusing the final product to be used by RCS apps. This product is called "Push Notifications Gateway", and aims to provide advertising to these applications through push notifications. However, the potential of the internship proved to be much higher, leading to set more ambitious goals.

1.3. Motivation

The delivery of advertising to RCS applications via push notifications is still underexplored. WIT Software saw an opportunity to offer MNOs the possibility to support the costs of offering these applications for free by the delivery of advertising. Since this strategy is already in adoption by some indirect competitors of WIT Software (the OTTs), after an analysis of its advantages, it was concluded that this would be a very nice feature to enrich the suite of RCS applications available for MNOs. Thus, this feature is not only a way to increase profits for our existing customers but also seeks to attract new ones by offering them something that other RCS development companies don't have.

1.4. Goals

This project is inserted in the Telco business unit and came initially from the need to solve a big limitation coming from the technology used in RCS applications, specifically in iOS, which is the way that Apple manages background applications. When a client doesn't have the application opened or in background, it never receives messages coming from other users from the IP Multimedia Subsystem Network, which is a great flaw and needs to be handled.

In order to solve this problem, WIT Software proposed the development of a Push Notifications Gateway (PNG), which has the main goal to deliver push notifications (PNs) to RCS applications developed by WIT Software for network operators, and also for them to use with other applications they have. Extending its usage, PNG will also need to be able to communicate with Android Clients and not only iOS ones, which expands the supported mobile platforms. In order to achieve this goal, there's also the need to develop a library for iOS and Android that allows the reception and trigger of those pushes.

Allied to the PNs, came the idea to use them to send advertising to the RCS applications. This explores a model that makes possible to support the costs of mobile messaging through advertising. In this model, the messages between smartphones will be free but in return, the consumers receive advertising from companies with consumer products. This way, mobile networks neutralize the loss of revenue for offering the service for free.

There's also the need to offer a great user experience for network operators, the internship also includes the creation of a Web application which makes possible for the administrators to manage their application users, send PNs and visualize reports and statistics.

Since this project is very flexible and offers the possibility to be used in different scenarios, it needs to be implemented not only for network operators but also for any application owner who wants to deliver PNs to his clients. Having this in mind, the idea of using notifications to deliver advertising becomes just a subset of what is possible to achieve, so the project may have a much broader market and a possibility to collect a wider range of potential clients.

1.5. Document Structure

In the next chapter, the state of the art will be presented. There will be a description of the services I will use in order to handle PNs and the already existing platforms in the market.

Then follows a presentation of the requirements identified for the project development, and also an overview of the architecture implemented.

In the next chapter, the software methodology used will be explained, how the work plan was defined and the identified risks and their associated mitigation plan.

Posteriorly, a chapter describing the tests made is presented. It is divided in three main groups, functional tests, quality tests and static program analysis.

Finally, the last chapter will describe all the implementation done, the contributions made, an explanation about the deviations from the original plan and suggestions for future work.

Attached to this document are the following appendixes:

- Appendix_A_-_Push_Notification_Services_Technical_Specifications

This document contains all the technical details regarding push notification providers, which I will use in order to deliver PNs to devices.

- Appendix_B_-_Requirements

This document contains all the requirements established for the project. They are divided into functional and non-functional requirements and are properly categorized and prioritized.

- Appendix_C_-_ Architecture

This document explains all the technical decisions made during the development process and how all the components work.

- Appendix_D_-_ Product_Backlog

This document has a comprehensive list of all the users stories that integrate the Product Backlog.

- Appendix_E_-_ Software_Quality

This document presents all the tests made in the application. It also helps to clarify some of the decisions made by analyzing some benchmark values.

- Appendix_F_-_ Competitor Analysis

This document gives a detailed analysis about WTT RCS Apps competitors.

2. State of the art

The state of the art aims to provide an overview of the current market where the internship fits. There will be several comparative analyses, about the similar solutions to my final product (Push Messaging). It is important to give the reader a glance about the context and the thematic of the work that took place.

This chapter also seeks to show the research that has been carried out. This research helped on making decisions for the implementation done through the analysis of what exists and what needed to be improved in the current market.

2.1. Push Messaging Solutions

In this section I will address the main Push Messaging solutions available in the market that aim to reach smartphones with content. This technology enables the delivery of PNs to one or more users according to a set of rules established by them or by the server.

This analysis will be divided in two main areas, one relative to the proprietary solutions of the companies developing smartphone operating systems, and the other about 3rd party solutions which integrates all the proprietaries into one unique platform.

2.1.1. Proprietary solutions

- Apple 

Apple has a PN system denominated Apple Push Notifications Service (APNS), which is available since iOS 3 [2]. The PN flow is presented in Figure 1.

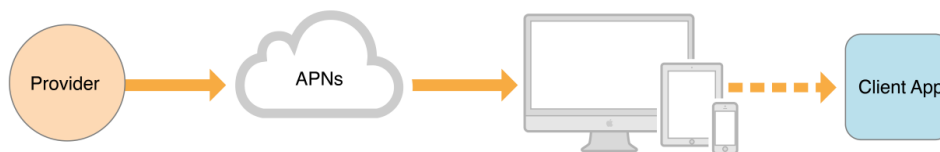


Figure 1 - Push Notification flow in iOS [3]

The source server (Provider) generates initially a notification, sending it to APNS through a secure sockets layer (SSL) connection. The service takes care of distributing the notification to the target device(s). If these are not available in the moment, APNS stores the notification for a certain period of time (Store & Forward) in order to assure the delivery of the data later.

Initially, in order for the application to receive notifications, it needs to send a request to the APNS, which in turn will generate a unique token that identifies the requesting application in that specific device. This identifier will be redirected by the smartphone in background to the Application Server (Provider), which will store it in the database. Therefore, when a notification needs to be delivered to that device, it will be associated with the token which will be delivered to the APNS in order for it to know which is the target device.

In order to guarantee the security model, two certificates are necessary in order to enable the establishment of a secure connection either between the provider and the APNS or the APNS and the device. First the Provider needs to obtain a certificate generated from Apple along with a private key

that serves as its identification. The device in turn uses the public server certificate passed by the APNS in order to authenticate the service to which it is connected. Lastly, the client with its certificate and private key authenticates itself to the service and establishes an SSL connection in order to receive the notification.

The notification payload format accepted by APNs is JSON and its maximum size is 256 bytes. Beyond the custom payload data that can be sent, Apple defines specific key-value pairs that perform certain operations. For example, the notification text must be defined inside the “alert” key and the sound inside “sound”. On iOS, when the push notification arrives in the device it is immediately displayed on the notification center, with no intervention needed to be made by the application. It is possible however to pre-process the notification that arrives in the device, and decide what to do with it. For example, if a notification serves the purpose of waking the application to do some background operations, the payload must specify “content-available” : 1 and the application must support capabilities for remote notifications in order to wake the application when a notification is received.

APNs has two systems to report problems with the notifications sent or device tokens. They are the feedback service and the error-response packets. The first one needs to be parsed regularly and gives information about invalid tokens while the latter provides immediate error reports when pushing notifications.

- Google 

In order to send notifications to their Android devices, Google created a service called Google Cloud Messaging (GCM). This service requires that the device have, at least, the version 2.2 installed. Conceptually, the architecture presented by Google is pretty similar to Apple’s. Figure 2 expresses the flow of an Android Notification.

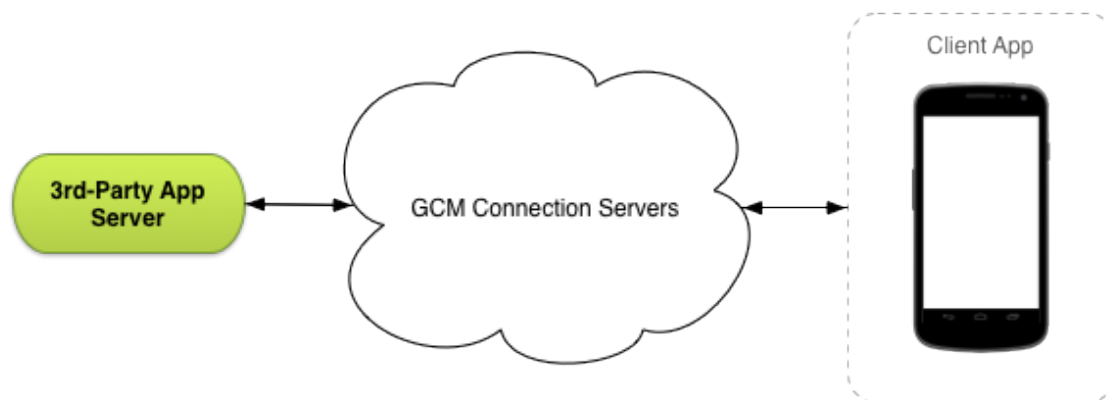


Figure 2 - Android Push Notification Flow [4]

On an initial phase, the application needs to be registered by the developer in the Google API Console in order to obtain an API Key and a Project ID. With this Project ID, also known as “Sender ID”, a register request will be made to the GCM servers in order to obtain the Registration ID, which will identify the app running on that specific device. The Registration ID will be stored on both the device and the database in the Application Server. The device can either register itself in the XMPP server or the HTTP (both offered by Google), depending on whether respectively it wants to send upstream messages or not.

The Application Server is responsible for binding to the same GCM server that the devices are connected to. For this, it must use the API Key generated in the API Google Console in order to identify the application in use and be able to establish a secure connection with transport layer security (TLS) or HTTPS, depending on the chosen server.

Every sent message is in JavaScript Object Notation format (JSON) and includes the Registration IDs of the target devices, being dispatched via HTTP POST or encapsulated in XMPP messages (once again depending on the chosen server). Unlike what happens on iOS, the notifications received on the Android device do not trigger an automatic alert to the notification bar. It is the application that decides what to do with the notification content received.

Since the communication with Google Cloud Messaging is done via HTTP POST requests, the feedback possible to obtain in regard to errors occurred comes in the request's responses. Response codes different than 200 represent a variety of possible errors that makes possible to understand what went wrong and possibly retry the operation successfully.

- Microsoft 

Microsoft offers a service to send notifications called Microsoft Push Notification Service (MPNS). This works asynchronously and according to a best-effort model, meaning that it doesn't guarantee the delivery of the notifications to the client. In fact all services work this way. Figure 3 represents how a PN is sent.

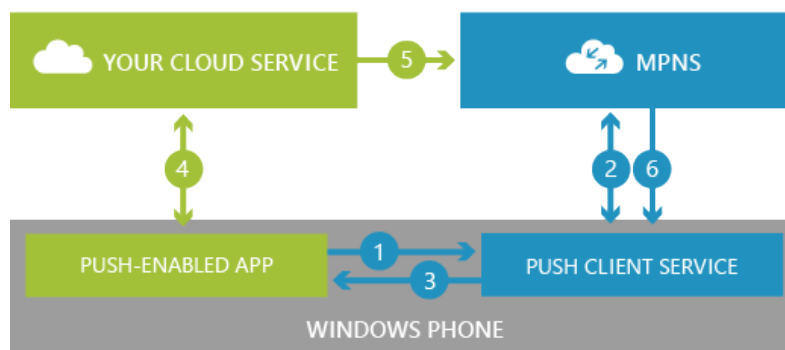


Figure 3 - Flow of a Push Notification on Windows Phone [5]

In the first step, the application needs to ask for a PN uniform resource identifier (URI) to the Push Client Service, which in turn will redirect the request to the MPNS. The reader should interpret the Push Client Service as a module available on the Phone that has the responsibility to serve all PNs for that device. After that, MPNS will reply with an URI that identifies the application running on that device, which will then be sent in background to the Cloud Service (Provider) to be stored. Henceforward, every time the Provider needs to send a PN it just needs to communicate directly with MPNS, which in turn will redirect it to the URI attached. All communication is done via HTTP.

For a thorough understand about the inners of APNs and GCM, please refer to the technical specifications document that can be found on Appendix A

2.1.2. 3rd Party solutions

The 3rd party solutions available in the market describe themselves as a viable and consistent alternative to the proprietary solutions. Nevertheless, we have to keep in mind that all of this solutions use the proprietaries defined above, with the difference that they enable the use of all platforms in a transparent way, allied to a set of features that enables a high level of personalization and parameterization. They offer APIs to communicate and perform operations in order to transform the implementation in something less painful and heavy. At the time of the architectural decision to implement, one must have in mind two factors:

- Bandwidth and Cost

We should question ourselves if our infrastructure is able to handle possibly millions of notifications for millions of devices. In this aspect, the inherent costs to maintain a server that offers sufficient bandwidth will always increase as our clients arrive. Also in the bandwidth topic, this also applies to the capacity in terms of human resources to develop such a huge and modular server.

- Multi-Platform

In the case of our application be present on multiple platforms, not only it will become complex to implement each solution of the proprietary but also entails implementation costs (working hours) by the employer towards its workers.

Having in mind these components, it's important to consider these aspects when it comes to make a decision in order to bring together the best of both worlds.

2.1.2.1. Brief description

A comparative analysis between the best Push Messaging solutions available in the market will follow.


- Push.io 

Push.io is one of the Push Messaging competing solutions in the market. Founded in April 2009 by ex-Apple workers and experienced entrepreneurs [6], it assumes itself as a viable alternative to the remaining companies in the same business. It was acquired in January 2014 by Responsys [7], which is Oracle's Marketing Cloud.

Depending on the usage, Push.io offers two types of plans. In the first one there's a limit of 10.000 users, although the number of PNs that can be sent be infinite. The second one is advantageous in case of having a higher number of users compared to what the first offers. Thus, the price depends on the number of PNs that the client sends.

Monthly Per User	Per Push
10.000 Users	\$99.00 / 25.000 Pushes
\$99.00 / Month	\$199.00 / 250.000 Pushes
	\$399.00 / 1.000.000 Pushes

Table 1 - Push.io available Plans [8]

- Parse 

Parse offers a variety of services to its clients, although I'm only going to focus on the notification service, known as "Parse Push". Acquired recently by Facebook [9], it offers one of the best free plans available in the market.

A comparative analysis of the features offered according to the monthly plan chosen.

	Free	Paid
Price	FREE	\$100 - \$1700 per month
API Requests per month	30 req/s	40 - 200+ req/s
Pushes per month	1,000,000 unique recipients 5¢ for each 1,000 above	1,000,000 unique recipients 5¢ for each 1,000 above
Background Jobs	1 Concurrent Job	2 – 10 Concurrent Jobs
App open metrics	Y	Y
Compare metrics	Y	Y
Global Push opens metrics	Y	Y

	Free	Paid
Advanced Push Targeting	Y	Y
Push Scheduling	Y	Y
Individual Push opens metrics	Y	Y

Table 2 - Parse available plans [10]

- Urban Airship 

Founded in 2009 with the aim to provide a service that simplified the delivery of PNs for iOS, Urban Airship has grown to the point of becoming the most deployed service globally [11]. Regarding the plans they offer, their information is very limited being the client obliged to contact them via email. Anyway, the data that could be gathered is shown in Table 3.

	Developer Edition	Small Business Edition	Enterprise Edition
Price	FREE for 45 days	Depends on the number of users	Depends on the number of users
Pushes per month	1,000,000 \$0.001 for each push above \$0.0025 for each Rich Push above	Unlimited	Unlimited
Number of users	Unlimited	Unlimited	Unlimited
Documentation	Y	Y	Y
Rich media landing pages	N	Y	Y
Core push activity reports	N	Y	Y
Full reports	N	N	Y
Location Targeting	N	Add-on	Add-on
Target audience segmentation	N	N	Y

Table 3 - Urban Airship available plans [12][13]

- Xtify **X**

Xtify was founded in 2009 when the world of PNs began to appear on the mobile applications. Due to its success, IBM acquired it in October 2013 with the objective to combine the Push Messaging potential with their analysis power and cloud infrastructure [14]. With these components, Xtify pretends to offer innovative content, which allows them to influence and inform their consumers.

	Developer	Just Push	Enterprise Complete
Price	\$199/month	Starting at \$1000/month	Starting at \$2500/month
Push Notifications	When available Up to 30,000 devices	Unlimited	Unlimited
Push Scheduling	Y	Unlimited	Unlimited
Time-zone based delivery	Y	Y	Y
Segments	Limited	Unlimited	Unlimited
Analytics	Y	Y	Y
Full reports	N	Y	Y
Location Targeting	N	N	Y
Target audience segmentation	N	N	Y

Table 4 - Xtify available plans [15][16]

2.1.2.2. Features comparison

A comparative table with the different Push Messaging solutions in study will follow according to this set of features:

- **Dashboard:** Allows to configure all the platform available features on the online panel.
- **Audience Segmentation:** Allows the creation of groups, combining users preferences and interests, in order to send notifications, not for everyone but for specific groups.
- **Advanced Targeting:** Allows the creation of user segments based on certain parameters such as: age, location, language, preferences and behaviors inside the application.
- **Advanced Scheduling:** Allows to schedule notifications according to a specific time zone in order to reach all users at a specific hour. There's also the possibility to define the frequency and regularity with which the notification is sent.
- **Statistics/Analytics:** Allows to visualize statistical information with the aid of graphics, about the notifications receptivity by the users, to evaluate their campaigns effectiveness. In this way, the administrators can see the number of times that their application were launched either through the notifications or not, number of API requests, types of devices and a lot more. There's also the possibility to filter this results based on date, type of device and also to generate reports with a variety of information.
- **API Integration:** Through the API offered, it's possible to integrate all the platform features in the Application Server and thus, avoid the use of the Dashboard to do some tasks which have the need to be dynamically defined.
- **Message Center:** Offers the possibility to create within the application a message center which offers to the users, deals, promotions and videos. Thus, this is a way to reach users who didn't allow the application to send pushes and force them to see the contents of the message center in order to eliminate the notifications counter coupled to the application icon.
- **Client SDKs:** Software Development Kits (SDK) allow the integration of features such as receiving PNs on the devices for which they are available.
- **Rich Messaging:** Allows to send notifications with a variety of content such as videos, images and coupons. It's also possible to integrate HTML inside the notification in order to take advantage of features present in the application. So, the push just needs to be sent and when opened by the user it will show its content inside the application.
- **Newsstand Integration:** Supports Apple's Newsstand applications.
- **Passbook Integration:** Supports Apple's Passbook applications. With this feature, developers can personalize the templates for their tickets, which will be sent after their purchase.
- **Google Wallet Integration:** Supports Android's Google Wallet.

	Push.io	Parse	Urban Airship	Xtify
Dashboard	Y	Y	Y	Y
Audience Segmentation	Y	Y	Y	Y
Advanced Targeting				
Age	N	Y	N	N
Location	Y	Y	Y	Y
Language	N	Y	N	N
In-app behaviors	N	N	Y	N
Preferences	N	N	Y	N
Advanced Scheduling				
Time-zone	Y	Y	Y	Y
Frequency & Pace Management	N	Y	N	Y
Client SDKs				
iOS	Y	Y	Y	Y
Android	Y	Y	Y	Y
Blackberry 10	N	N	Y	Y
Windows Phone 8	Y	Y	Y	Y
Phone Gap	N	N	Y	N
Windows 8	N	Y	Y	N
OS X	N	Y	N	N
JavaScript	N	Y	N	Y
Statistics/Analytics	Y	Y	Y	Y
API Integration	Y	Y	Y	Y
Message Center	N	N	Y	Y
Rich Messaging	N	N	Y	Y
Newsstand Integration	Y	N	Y	N
Passbook Integration	N	N	Y	Y
Google Wallet Integration	N	N	Y	Y

Table 5 - Features comparison of the Push Messaging Services

Urban Airship is the push messaging solution which provides the higher number of features. It is also the only platform which allows targeting users based on their in-app behaviors and preferences.

There are features which are already considered a standard in this industry, once all these solutions have them. Audience segmentation, targeting, scheduling and statistics are a good example of them and will all be a part of my solution.

3. Requirements

This section will focus on the requirements of the project. It represents the scope of the internship and the goals established in the beginning of the project that needed to be reached in the final solution. It's important to clarify that the requirements specified in here won't be as detailed as the document that serves this purpose, thus, the reader should complete his reading with the full specification requirements document (Appendix B).

The requirements will be divided in two main groups: functionals and non-functionals. Each one of them will have an identifier, title, priority, type and description. For a better understanding of the priorities and types of requirements, I'll explain in the following topics.

3.1. Priorities

All the requirements are related to a specific priority, which establishes its importance for the final product. Thus, a scale was created in order to classify different requirements.

Type of priority	Description
Must	Must be met and are of great importance for the project's success.
Should	Represent a medium priority. They are important, but if they are not implemented the project won't be compromised.
Could	Represent a low priority. Should only be implemented if there is enough budget (time).
Won't	They are future recommendations and won't be implemented.

Table 6 - Requirements' priorities

3.2. Types

In order to make the requirements more consistent and complete, an extra layer of specification will be added, the type. In this way, it is possible to understand the context where the requirements belong and to make easier to distinguish the different types of existing requirements.

Type of requirement	Description
Functionality	Describe system activities. The security is included here.
Usability	Based on the users, interfaces and accessibility.
Reliability	Fault tolerance, system availability.
Performance	System performance, use of resources.
Supportability	Scalability, compatibility, configuration, tests, maintainability.
Security	Protection of the system and communications.
Technical Constraint	Pre-established technical constraints that must be met.

Table 7 – Requirements' types

3.3. Functional Requirements

Functional requirements (FR) describe what is intended to be done by the software, in general, its features. In order to define these requirements, there are 3 subtopics which intend to divide into categories each functional requirements: Server, Device and Web UI.

3.3.1. Server

FR_01 - Send Notifications to Apple and Android Devices	
Priority: Must	Type: Functionality
Description	The server must be able to send PNs to Apple and Google devices.

FR_02 – Get Scheduled Notifications	
Priority: Must	Type: Functionality
Description	The server must be able to get all scheduled notifications.

FR_03 – Edit Scheduled Notifications	
Priority: Must	Type: Functionality
Description	The server must be able to edit scheduled notifications.

FR_04 – Delete Scheduled Notifications	
Priority: Must	Type: Functionality
Description	The server must be able to delete scheduled notifications.

FR_05 – Information about PNs Sent	
Priority: Must	Type: Functionality
Description	The server must be able, upon request, to return information about PNs Sent.

FR_06 – Information about PNs opened	
Priority: Must	Type: Functionality
Description	The server must be able, upon request, to return information about PNs opened.

FR_07 – Information about App Opens	
Priority: Must	Type: Functionality
Description	The server must be able, upon request, to return information about app opens.

FR_08 – Get information about device(s)	
Priority: Must	Type: Functionality
Description	The server must be able, upon request, to return information about all or a specific device. This includes information about the last time the application was launched, the last user location, segments to which the user belong, the device token, brand and version owned, time zone, and number of push notifications received and opened.

FR_09 – Create Segments of users	
Priority: Must	Type: Functionality
Description	The server must be able, upon request, to create segments of users based on a set of rules like, country, city, coordinates with a radius or platform.

FR_10 – Get segment(s)	
Priority: Must	Type: Functionality
Description	The server must be able, upon request, to return all the existing segments or one specified. In the latter, the name, number of devices and criteria are returned. If the user decides to not specify any segment, for each one, its id, name and number of devices are returned.

FR_11 – Edit segment	
Priority: Must	Type: Functionality
Description	The server must be able, upon request, to edit a segment.

FR_12 – Delete segment.	
Priority: Must	Type: Functionality
Description	The server must be able, upon request, to delete a segment.

FR_13 – List countries.	
Priority: Must	Type: Functionality
Description	The server must be able, upon request, to list all countries available.

FR_14 – List cities.	
Priority: Must	Type: Functionality
Description	The server must be able, upon request, to list all cities belonging to a country specified.

FR_15 – Receive device information's	
Priority: Must	Type: Functionality
Description	The server must be able to receive from the devices a registration request which includes: device ID, device token, device module, device version, display name, app version, user ID, time of registration and time zone. It must be able to receive their location, represented by a set of coordinates along with a date and time and also information about when the user opens the application. Finally, when a PN is opened, the client must send that info to the server.

FR_16 – Persist client information	
Priority: Must	Type: Functionality
Description	The server must be able to store, update and delete on the database, information received from the devices.

FR_17 – Create an application	
Priority: Must	Type: Functionality
Description	The server must be able to receive requests to create new applications to send PNs to. These applications are created on the Web UI.

FR_18 – Generate an APP_ID	
Priority: Must	Type: Functionality
Description	The server must be able upon receiving an app creation request to generate an APP_ID, which will be used on every request made to the API, for the purpose of authentication.

FR_19 – Persist Applications	
Priority: Must	Type: Functionality
Description	The server must store on the database the applications created by the clients.

3.3.2. Device

FR_20 – Register for PNs	
Priority: Must	Type: Functionality
Description	The device must be able to register itself with the corresponding Push Provider.

FR_21 – Send Push Registration and Device Information to PNG	
Priority: Must	Type: Functionality
Description	The device must be able after registering with the Push Provider, to send the device token along with other device information.

FR_22 – Send Location information	
Priority: Must	Type: Functionality
Description	The device must be able to send its current location to the PNG.

FR_23 – Send Information upon opening the application	
Priority: Must	Type: Functionality
Description	The device must be able to send the date and time when the application was opened.

FR_24 – Send Information upon opening a PN	
Priority: Must	Type: Functionality
Description	When the client opens a PN, an acknowledge must be sent to the PNG with the ID of the PN opened and the time.

FR_25 – Request PNG to send a PN to an Apple device	
Priority: Must	Type: Functionality
Description	The client application must be able to request the PNG to send a PN to an Apple device in order to deliver any information.

FR_26 – Request PNG to send a silent PN to an Apple device	
Priority: Must	Type: Functionality
Description	The application must be able to request the PNG to send a silent PN to an Apple device in order to wake the application.

3.3.3. Web UI

FR_27 – Login	
Priority: Must	Type: Functionality
Description	The client should be able to login in the Web UI and have access to his applications and associated functionalities.

FR_28 – Multi-session	
Priority: Must	Type: Functionality
Description	Must support multiple login sessions concurrently for the same user.

FR_29 – Logout	
Priority: Must	Type: Functionality
Description	The client should be able to logout from his account.

FR_30 –User with Multiple Applications	
Priority: Must	Type: Functionality
Description	The client should be able to register multiple applications inside his account.

FR_31 – Create an Application	
Priority: Must	Type: Functionality
Description	The Web UI should provide an interface to create an Application project in order to start using the Notifications Gateway. Upon creation it's mandatory to define: the Apple Certificate and password and/or the Google API_KEY (the application can be used for both platforms), and a Name for the application. It is also possible for the user to upload the application icon.

All the remaining features present on the Web UI are related with the ones described on the server topic and thus, will not be part of this topic. If the reader wants to view the remaining web requirements he should refer to *Appendix B 4.3 Web UI*.

3.4. Non-Functional Requirements

Non-Functional Requirements specify how a system should behave rather than the features it must implement. Although they must be related to functional requirements, non-functionals aim to define different types of aspects of the general system such as: usability, security, performance, supportability and technical constraints.

This topic will be divided in three different categories: Web UI, Devices and Server.

3.4.1. Web UI

NFR_01 – Web UI Offers a Good Interface	
Priority: Must	Type: Usability
Description	The Web UI should provide an interface design according to WIT Software's standards, which means: easy to understand, intuitive, and with a great User Experience.

NFR_02 – Web UI Alphanumeric Login Password	
Priority: Must	Type: Security
Description	Every registered user must have an alphanumeric password in order to increase the security of his account.

NFR_03 – Web UI Responsiveness	
Priority: Must	Type: Performance
Description	Every response in the Web UI should happen in less than 3 seconds, since it is a reasonable time to wait. Users expect a page to load in 2 seconds and when it takes more than 3 they start to abandon the site. [17]

NFR_04 – Multi-language	
Priority: Won't	Type: Usability
Description	The Web UI supports multiple languages.

3.4.2. Devices

NFR_05 – Available for iOS	
Priority: Must	Type: Supportability
Description	PNs must be available for iOS.

NFR_06 – Available for Android	
Priority: Must	Type: Supportability
Description	PNs must be available for Android.

NFR_07 – Available for Windows Phone	
Priority: Could	Type: Supportability
Description	PNs must be available for Windows Phone.

3.4.3. Server

NFR_08 – Comprehensive Server logs	
Priority: Must	Type: Security
Description	Every action made by the server must be logged on files.

NFR_09 – Multi-Instances Load Balanced	
Priority: Could	Type: Supportability
Description	The server could be deployed in multiple instances with a load balancer distributing the work.

NFR_10 – Requests to the API Authenticated by an APP KEY	
Priority: Must	Type: Security
Description	An App ID must authenticate all requests to either of the APIs in order to ensure authenticity.

NFR_11 – Encrypted Requests	
Priority: Must	Type: Security
Description	Every request to either of the APIs should be encrypted in order to ensure confidentiality. This encryption will be done using HTTPS.

NFR_12 – Server implementation in Java	
Priority: Must	Type: Technical Constraint
Description	The server-side implementation must be in the Java language.

NFR_13 – Conformance to coding standards	
Priority: Must	Type: Supportability
Description	All software code must conform to WIT Software's standards.

NFR_14 – API Communication in JSON	
Priority: Must	Type: Technical Constraint
Description	Unless it violates a specification made by 3 rd parties, all data transmitted to and from the APIs should be in JSON format.

NFR_15 – Deployment in Apache Tomcat	
Priority: Must	Type: Technical Constraint
Description	All the server-side code must be deployed in Apache's Tomcat HTTP Server.

NFR_16 – Throughput Handling	
Priority: Must	Type: Performance
Description	The server must be able to handle an average of 3 Million PN's per day. Please see <i>Appendix C 5.1 Throughput study</i> to understand the study made to achieve such value.

NFR_17 – Apple certificates, passwords, Google API Key and User passwords encrypted	
Priority: Should	Type: Security
Description	The server should be able to store these values in the database encrypted

This page was intentionally left in blank

4. Architecture

The architecture will be explained in this chapter, though for a comprehensive understanding of all the aspects regarding this topic, the reader should see the architecture document (Appendix C), which has a full detailed explanation of all the underlying mechanisms and decisions made.

To start this chapter, Figure 4 sums the whole environment of the project's development.

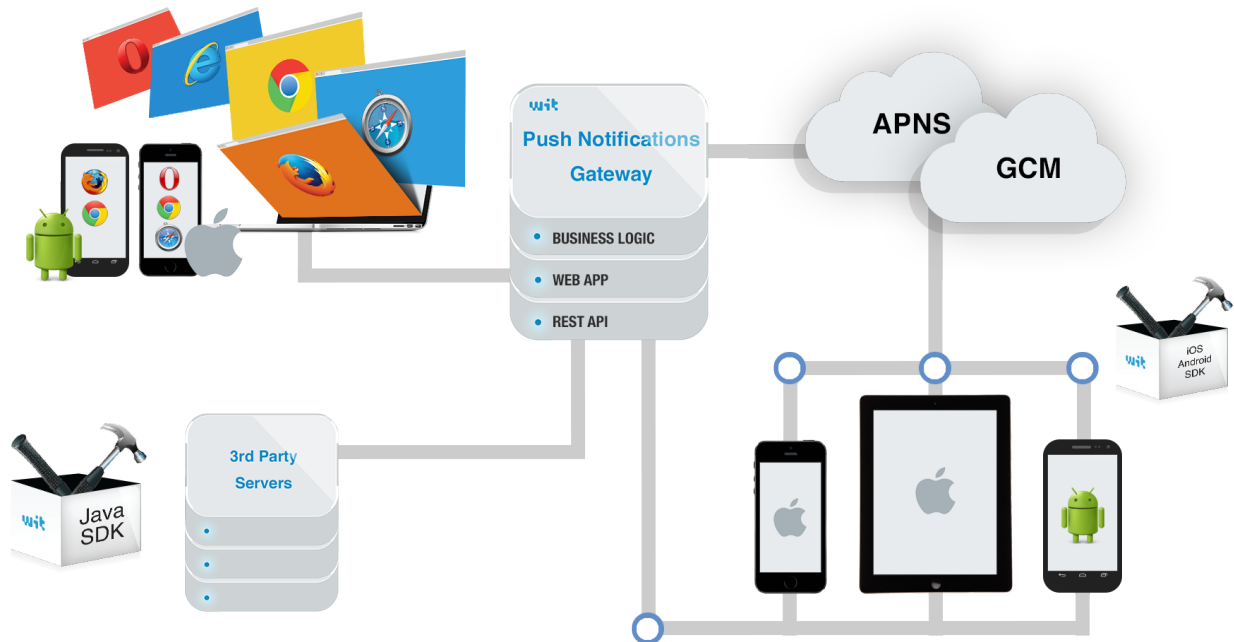


Figure 4 – High-level design

The solution is composed of four main components:

Web Application: The web application has a responsive layout which makes the mobile's users experience easier and improved, by offering them a version for this devices.

REST API: It is divided in two main interfaces: one for device-gateway communication and the other for 3rd party servers-gateway communication. The first one is in turn also divided in two interfaces: the Generic which exposes endpoints for any generic device integration, and the Telco which is a subset of the first, offering only a subset of the functionalities (more information in 4.1.2). The 3rd party servers API enables all the features available in the web to be integrated with the client's code.

SDKs: For the device-gateway communication there are two SDKs, one for iOS and another for Android, which simplify all the work on the client's application. Its use is mandatory since the API Definition is not available for the public.

For the 3rd party servers API there is a Java SDK built for clients who don't want to handle the payload construction and the HTTP requests.

Business Logic: The core of the gateway, responsible to handle all the requests and process all the data. It composes all the components defined in 4.3.1 Components. It is fully written in Java.

4.1. REST API Definition

The APIs are an important piece of the architecture since they allow the communication of external software, hiding the details of the internal implementation.

For the purpose of this project, three APIs needed to be built in order for the devices and the clients make HTTP requests. All the responses have a status code and also a JSON or XML body giving more feedback to the developer about what happened. Following there's a description of each one of the interfaces.

4.1.1. Generic API

The generic API has the purpose to enable application owners to deliver PNs to their clients. It accepts only JSON requests and all the logic behind the requests is hidden behind a Library or a Framework, that was made in order to ease the integration with the PNG. These Libraries will be explained further in this topic.

Regarding the possibilities of this API, the following features were implemented, and although they are already described on the Requirements topic, they will be reminded here.

Verb	Endpoint	Requirement	Description
POST	/generic/client/register	FR_21	Registers the device on PNG
POST	/generic/client/location	FR_22	Submits device location
POST	/generic/client/launched	FR_23	Acknowledges that the application was launched
POST	/generic/client/conversion	FR_24	Acknowledges that a PN was opened

Table 8 - Generic API Definition

Internally, the register endpoint can distinguish a creation of a new client from an update, since updates are made with the verb POST, instead of PUT.

4.1.2. Telco API

The Telco API is a subset of the Generic, and is aimed for network operators to use with their apps. Thus, one can consider it as being a lighter version of the Generic API, which only allows three types of requests:

Verb	Endpoint	Requirement	Description
POST	/telco/client/register	FR_21	Registers the device
POST	/telco/client/sendAlertPushRequest	FR_25	Sends PN to another device
POST	/telco/client/sendSilentPushRequest	FR_26	Sends Silent PN to an Apple device

Table 9 - Telco API Definition

Since this API followed specifications previously made for a European Network Operator, there were some rules that needed to be followed. First, all the communications need to be made in XML, contrary to JSON that is used in the remaining APIs, and secondly, all the requests need to be validated against an XML schema in order to meet the expected format for them.

4.1.3. Client/Network Operator API

The Client/Network Operator API allows developers of applications to integrate PNG features into their server code, and thus, focused on more technical users.

Table 10 has all the requests that are possible to make to this interface:

Verb	Endpoint	Requirement	Description
POST	/api/push	FR_01	Sends immediate or scheduled PN to one or multiple devices, possibly according to a set of filters.
GET	/api/push/scheduled	FR_02	Returns all scheduled PNs.
PUT	/api/push/scheduled	FR_03	Edits a scheduled notification.
DELETE	/api/push/scheduled	FR_04	Deletes a scheduled notification.
GET	/api/reports/pushsent	FR_05	Returns information about pushes sent according to a set of filters.
GET	/api/reports/pushopens	FR_06	Returns the number of opened PNs according to a set of filters.
GET	/api/reports/appopens	FR_07	Returns the number of applications opens according to a set of filters.
GET	/api/reports/device	FR_08	Returns all device Ids.
GET	/api/reports/device/{deviceId}	FR_08	Returns all information about a specific device
GET	/api/reports/device/count	FR_08	Returns the number of registered devices.
POST	/api/segments	FR_09	Creates a new segment of users according to a set of filters.
GET	/api/segments	FR_10	Returns information about all segments.
GET	/api/segments/{segmentId}	FR_10	Returns information about a specific segment
PUT	/api/segments	FR_11	Edits a specific segment.
DELETE	/api/segments	FR_12	Deletes a specific segment.
GET	/api/list/countries	FR_13	List all countries
GET	/api/list/cities	FR_14	List all cities belonging to a country

Table 10 – Client API operations

FR_08 and FR_10 are duplicated, and each one of them relates to the same functional requirement. This was done for the sake of simplicity. In the requirements document (Appendix B) the reader can acknowledge that these endpoints have separate specific requirements according to each one of the features. Although in order to simplify and compact the extensive list of requirements, since these are all related, they were included in the same requisite.

4.2. Device Libraries

In order to simplify the integration with the client applications, three libraries were developed, one for Android and two for iOS. Next there's a description of what the libraries do and how they do it.

4.2.1. iOS Generic Framework & Android Library

Given the similarities between the iOS framework and the Android library, both will be addressed in this topic. The first is a ".a" Cocoa Touch Static Library, while the Android library is a ".jar" which is included in the build path of the client app. In order to send requests, the client only needs to call the appropriate method and the library will take care of everything in the background and in a transparent way. Following there's an example of a register request made by the application.

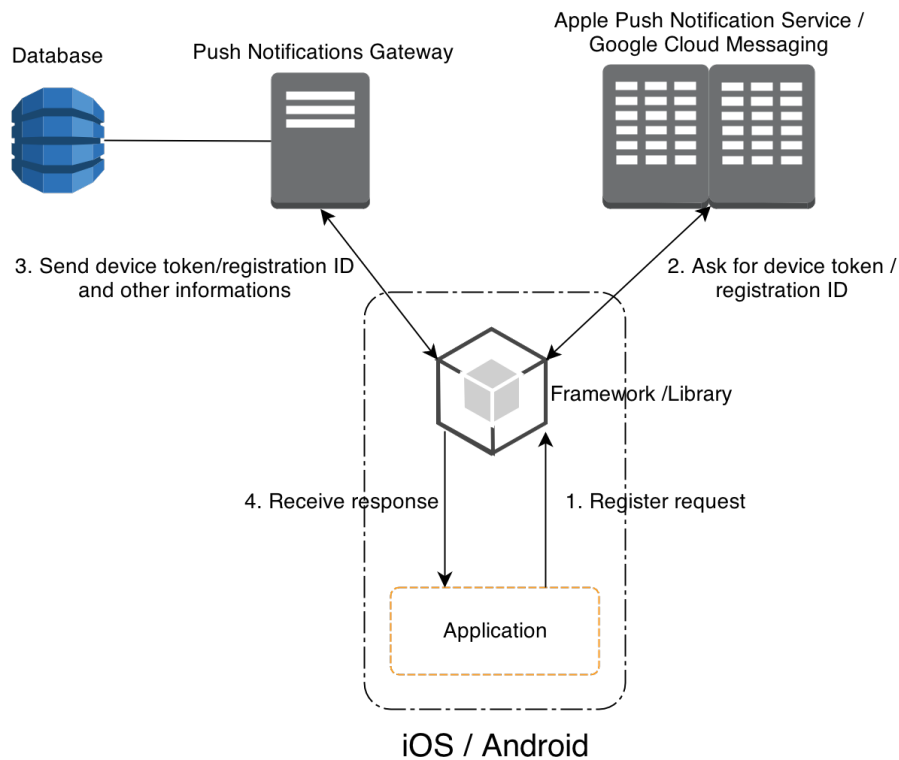


Figure 5 – Android Library Register Request

On the first step, the application invokes the *init()* method from the framework/library in order to start the registration process. Then, a Device Token/Registration ID is requested from APNs/GCM Servers (Step 2), and forwarded along with other information about the device to PNG (Step 3). Finally, the framework/library receives the response and redirects it to the application (Step 4).

In the android case, for the client to receive the responses from the server, he must implement the corresponding interface in the class where he wants to receive the data. By doing this, he is obliged to add the unimplemented methods. The Library is responsible to send the response data to the callback method implemented by the user. In iOS, the client needs to implement the delegate methods where he will receive the responses from the server.

The example showed above is the most complete use case that is handled by the framework/library. All other action available such as send location or launch information follows the same principles described, except the using of APNs/GCM Servers.

Since the information gathered from the devices are a crucial point for the functioning and success of the segment creation algorithms, this information should never be lost and thus, a reliability mechanism must exist. In order to guarantee that, in both libraries all the requests that fail to be delivered to the gateway are stored in the device's storage, with the exception of the register requests (there is no need to send no longer valid tokens to the gateway). The next time the library is initialized, pending requests will be sent.

4.2.2. iOS Telco Framework

The need to solve a particular use case regarding RCS apps motivated the creation of this specific framework. This one aims to be used by network operators with their apps in order to communicate and re-engage users. This framework is a subset, a lighter version of the iOS Generic Framework once it only implements the register and the push notifications use case.

The decision of not using the Generic framework for this use case is because it was not designed to follow the set of specifications that needed to be adopted. These specifications were made for a network operator and define, for example, a totally different data format for communication, which is XML. Since the implementation needs to be much different, it was decided to create this new framework.

The use case talked in the beginning of this topic is represented in Figure 6.

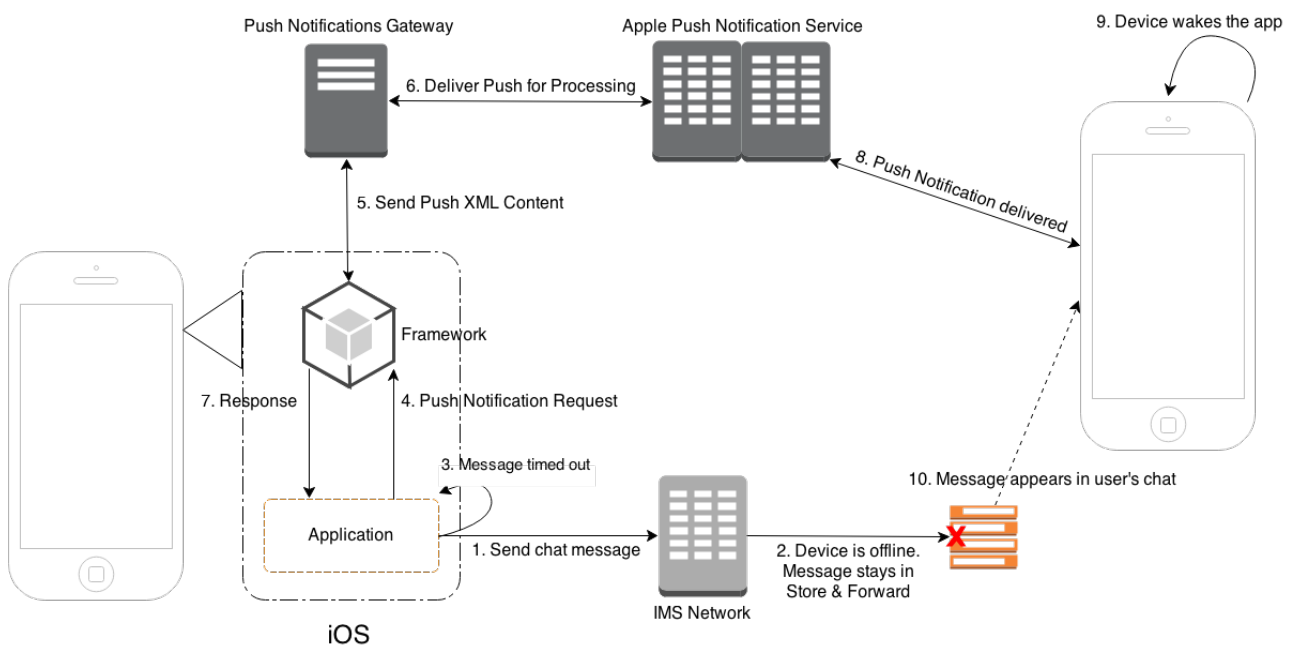


Figure 6 – iOS Telco Framework Push Notification Use Case

One of the major problems in RCS iOS Apps is that when the B party doesn't have the application opened or in background he can't be warned of an incoming chat message or call. In order to circumvent this limitation, the above architecture was designed.

When a chat message is sent to another party, it gets delivered to an IMS Network, which is responsible to pass it to the other device (Step 1). In this case, if the destination user has the application closed, this message is sent to an intermediate station called Store & Forward (Step 2), which keeps it and sends it later when the application registers itself again in the IMS Network. The A party application after waiting for a delivery notification and not receiving it, triggers a timeout (Step 3). When this happens,

the application requests the iOS Framework (Step 4) to contact the Gateway (Step 5) to send a Push Notification to the other party in order to warn him and invite him to open the app (Step 6 and 8). When the Push Notification arrives at the destination and the user opens it, the application will register itself in the IMS network and the pending message in the Store & Forward will be delivered right away (Step 10).

This architecture solves the chat use case and all other possible in the app, such as group chat messages, location share and many others that couldn't be delivered to the destination. Every time a delivery notification is not received before the timeout occurs, the application will request the iOS Framework to contact the PNG to deliver a Push Notification to the other device.

In order to the PNG know the Push Notification content (the message to deliver to the other party), a "Type" is provided upon requesting the Gateway. To clarify this last aspect let's have a look at the following chat message example:

1. The iOS Framework creates a request to send to PNG with the following content:

```
<?xml version="1.0" encoding="UTF-8"?>
<SendAlertPushRequest>
  <SIP-URI>sip:+351910000001@ims.example.com</SIP-URI>
  <ToURI>sip:+351910000002@ims.example.com</ToURI>
  <Type>APN_CHAT</Type>
</SendAlertPushRequest>
```

2. PNG sends a Push Notification with the following format:

```
{
  "aps": {
    "alert": {
      "loc-key": "APN_CHAT",
      "loc-args": ["351910000001"]
    },
    "sound": "default.aiff"
  },
  "eventType": "chat"
  "from": "351910000001"
}
```

This notification Payload is send to the device token of the destination user previously obtained from the Database.

3. The Push Notification arrives in the device. (The Push Notification Images are in Appendix C).

4.3. Server

The server represents a very important part of the architecture, since most of the work performed depends on it. The throughput number that was established in NFR_16 depends heavily on the architecture. To know such value, a study was made and is carefully explained in the architecture document (Appendix C).

Another important part of the server is the database model, which will be responsible to store and give responses to developer's requests.

Both topics will be discussed in this chapter, in order for a better understanding of the server's architecture.

4.3.1. Components

In order to reach the throughput defined on NFR_16 and to achieve requirements such as performance and reliability, the following architecture modules were designed:

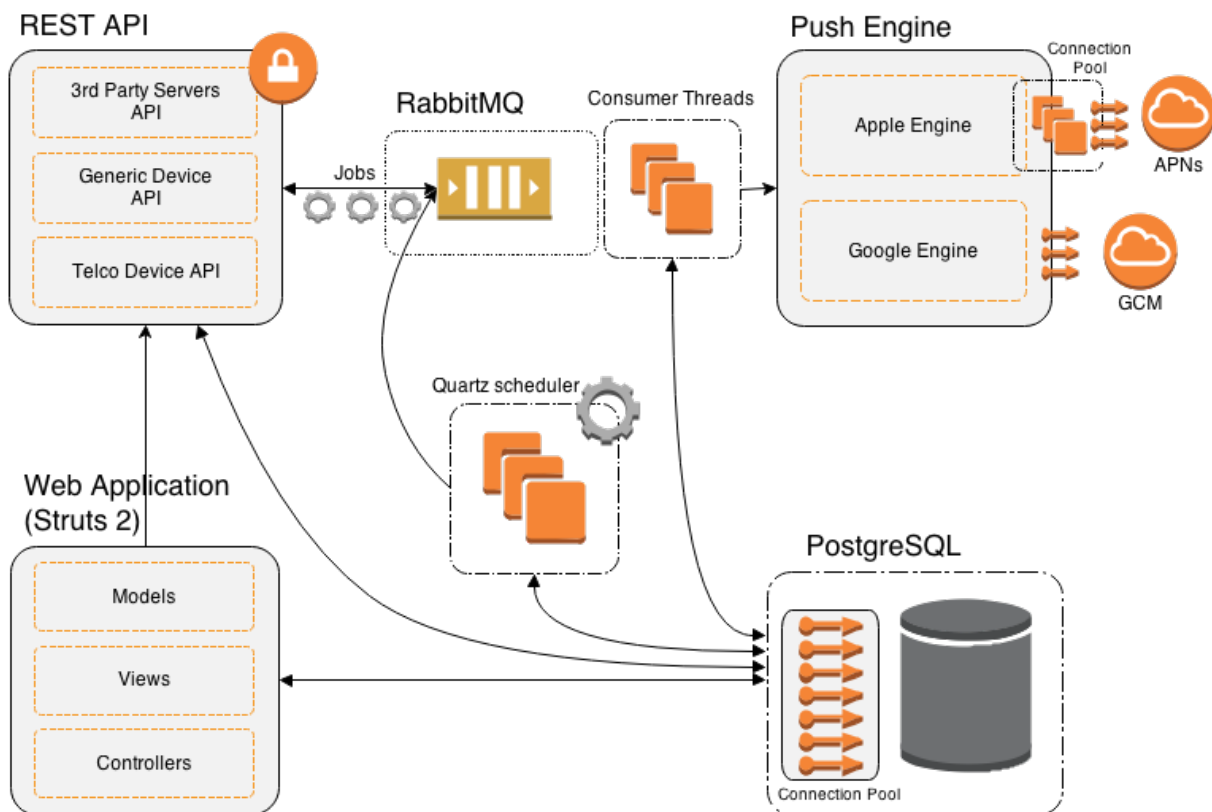


Figure 7 – Global Architecture Components

The above architecture (Figure 7) represents how the global system will work, and how all the pieces interact. These various components deserve special attention since each one of them decomposes in its own fine-grained architecture. For this purpose, each component will have its own sub-chapter where a deeper analysis will be made.

4.3.1.1. REST API

The REST API serves as an entrance point for the gateway. It exposes a series of endpoints to external clients as described in 4.1 REST API Definition. These endpoints are created with Jersey RESTful Web Services Framework, which is an open-source toolkit for Java developers who want to expose their data in a variety of representation media types [18].

Every request arriving at one of these endpoints needs to be authenticated first, which means that the HTTP request needs to have a valid “Authorization” header with an existing *APP_ID*. If the request is authenticated, the number of API or device requests (depending on the request’s source) stored in the database is incremented. In addition to this validation there are also other verifications performed depending on the endpoint triggered (see chapter 3.4 and 3.5 of Appendix C). After this process, the request is converted into a Job and stored in the database. In order to send this object to the RabbitMQ broker and henceforward to the queue, it needs to be serialized into a byte array, but only after changing its status to *QUEUED* and updating it in the database.

To clarify this concept, the reader should see the example in the Appendix C Figure 10 along with the description.

4.3.1.2. Jobs

One important concept in PNG is how the requests are processed and handled by the server. As already mentioned in 4.3.1.1, when a request arrives, in order to be forwarded to the queue for later execution, it must meet certain pre-conditions specific for that type of request. Pre-conditions can range from XML validation to JSON validation, among other types. If any condition fails, the response will be immediately generated to the requester with the associated error, and no Job will enter the queue.

If all pre-conditions are met, the request is ready to be processed and a Job is created, which is no more than a Java Object that encapsulates the following attributes:

- *id* (int): Unique identifier of that specific Job.
- *applicationId* (int): The application to which this job is associated.
- *creationTime* (Timestamp): The timestamp identifying when the Job was created.
- *executionTime* (long): Time in milliseconds since the Job is pulled out of the queue for processing until it has completed or failed.
- *queuedTime* (long): The time that the job was queued.
- *status* (Status): There are five types of status, as described in the Status ENUM, according to their execution stage: *CREATED*, *SCHEDULED*, *QUEUED*, *PROCESSING*, *COMPLETED*, *FAILED*.
- *type* (Type): The execution steps that must be made distinguish Job types. There are several, and can range from a simple *GENERIC_REGISTER* to a more complex *API_PUSH*.
- *request* (String): The request content, typically an XML or JSON converted to a String.
- *error* (String): If the Job fails, this attribute will store the complete Stack Trace in order to subsequently detect the originating problem.

Each different Job type is a subclass of Job. All subclasses, extending from Job, are specific for each type of request received. They all have an execute method which is in charge of performing all operations needed to successfully achieve the intended goal. This is the method called when a thread, belonging to the thread pool, acquires a job from the queue. In this stage, the execution flow can either end here, or being the job synchronous, the thread will reply the result to a reply queue, as will be

explained in the next chapter. In Appendix C Table 8 there is compilation of all the endpoints associated with the corresponding job types and also their execution mode (Synchronous or Asynchronous).

Another important aspect of the Job objects is their Status. Throughout the lifecycle of a Job it changes state several times, until its execution terminates. Figure 8 is a machine state diagram that intends to describe the different Status of Jobs.

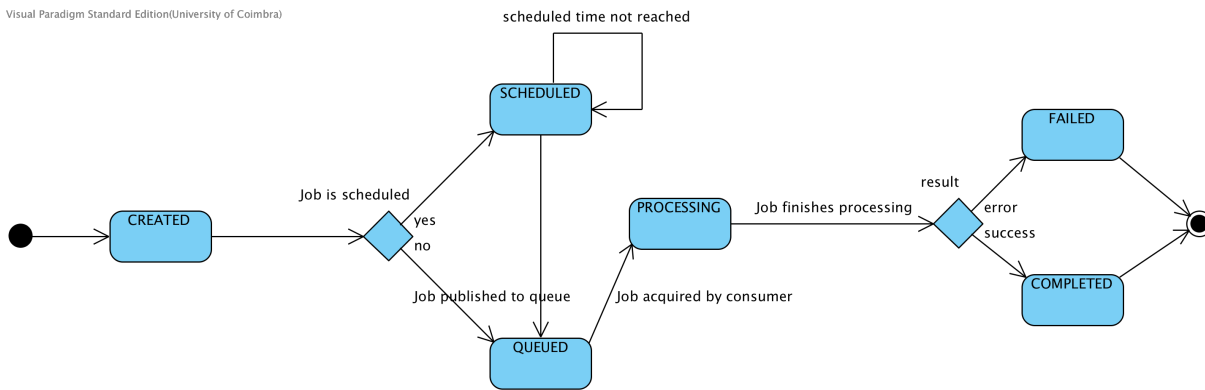


Figure 8 – Job Status Lifecycle

When the job is first created it automatically acquires the *CREATED* Status. Prior to its instantiation there’s a condition which defines the next Status of this object, the scheduled time. If the job is intended for immediate execution, it is published to the queue, acquiring the *QUEUED* status. If not, it first changes to the *SCHEDULED* status until the time for its execution arrives and only then it gets published to the queue and becomes *QUEUED*. The *PROCESSING* status occurs since a consumer acquires the Job and remains for the whole Job execution until it reaches a verdict, if the execution was successful the final status is *COMPLETED*, otherwise it is *FAILED*.

Status is related to execution and queued times. When a Job turns into the *QUEUED* status, current time in milliseconds is obtained from the system clock and stored. When the Job traverses the queue and is acquired by a thread, it changes to *PROCESSING* and the current time in milliseconds is subtracted to the previously stored in order to calculate the *queuedTime*. The same logic happens for the *executionTime*, when the Job turns into *FAILED* or *COMPLETED* status, the time is subtracted to the one stored when it started *PROCESSING*. These metrics are stored in the database once they are very important to understand the performance level of the gateway and possibly to detect bottlenecks.

Table 11 describes the actions taken by the gateway when a Job changes its Status.

Status	Action
CREATED	INSERT Job in database
SCHEDULED	UPDATE Job Status to <i>SCHEDULED</i> in database
QUEUED	Retrieve current time in milliseconds UPDATE Job Status to <i>QUEUED</i> in database
PROCESSING	Calculate <i>queuedTime</i> UPDATE Job Status to <i>PROCESSING</i> and <i>queuedTime</i> in database Retrieve current time in milliseconds
COMPLETED	Calculate <i>executionTime</i> UPDATE Job Status to <i>COMPLETED</i> and <i>executionTime</i> in database
FAILED	Calculate <i>executionTime</i> UPDATE Job Status to <i>FAILED</i> and <i>executionTime</i> in database

Table 11 - Job Status and related actions

4.3.1.3. RabbitMQ

In order to process requests received in the APIs, all the jobs are published to a queue in order to be processed later by the available threads. To assure a set of necessary features, RabbitMQ is the messaging broker chosen. It is based on AMQP, which is an advanced message queuing protocol created with message-oriented middleware in mind.

In RabbitMQ, there are five core definitions important to understand:

- **Producer:** The component that publishes jobs.
- **Connection:** TCP Connection that is used to connect to the RabbitMQ server.
- **Channel:** A channel is used to connect to the TCP Connection in order to deliver the job.
- **Queue:** It can be seen as a mailbox, stores all the messages in an infinite buffer.
- **Consumer:** The component that consumes the job and processes them.

RabbitMQ is one of the best open-source messaging protocols available. It is very stable, offering the possibility to cluster multiple servers, making them appear from the outside as one node, which is an added value in terms of scalability and future growth.

Reliability is one of the requirements covered by RabbitMQ. One of the features that offer reliable delivery is message acknowledges. An acknowledge is sent back from the consumer to tell RabbitMQ that a particular message has been received, processed and that it is free to delete it. This process is very useful in case a consumer dies. If it dies without sending an acknowledge, RabbitMQ will understand that the message wasn't processed successfully and will redeliver it to another consumer. Persistence is covered too, which is a feature of extreme importance to ensure that in case the queue or the broker dies, all messages being stored on disk will be later processed by integrating a new queue.

Availability is another feature in mind when it comes to assure high non-functional standards. It's possible to replicate queues to all or certain nodes in a cluster, so that if a node holding a queue dies, another one becomes active and takes over the work by activating the replicated queue. In this way, all clients can keep producing and consuming messages with no downtime.

In terms of performance, for small messages, which is the case since all jobs are serialized java objects with very few attributes, RabbitMQ outperforms the other main messaging queues by a factor of 3 [19], being capable of processing ~5000 messages per second with a single producer and consumer [20]. This was the main feature in consideration upon choosing the messaging broker. The other messaging solutions studied were ActiveMQ, HornetQ, QPID and Apollo, although neither conjugated all the features in such an efficient way. For example, ActiveMQ and QPID are very slow when using persistence [21] and HornetQ performs poorly with small and medium sized messages [19]. Apollo, which is the next generation of ActiveMQ with a totally different architecture built from scratch, is still in its early stages, not offering much features and still accounting with a small number of users in the community.

Since with RabbitMQ, in a very simple scenario it can handle way more messages per second than the target number we aim for (35 push notifications per second), it won't ever represent a bottleneck.

There are two architectures used depending on the type of job to be performed, whether it is a synchronous operation or asynchronous. Both of them have common components and principles, which are always used independently of the type of operation to execute.

- There are only two active RabbitMQ TCP connections, one to push jobs into the queue and another to pull. These connections are created when the server starts and are alive for the whole time.

- Each time a request arrives in the API a new channel is created using the already existing TCP connection.
- Jobs are only pulled into the queue if its pre-conditions are met. For example, if it's a Telco Push Notification request, the job will only be accepted if the XML is valid.
- Each thread consuming jobs has a dedicated channel to the TCP connection. These channels are instantiated when the thread pool is initialized (when the server starts).
- When a job is taken from the queue by a thread, an acknowledge is sent to the channel in order to signal RabbitMQ that the job was delivered successfully.
- The Queues are durable and all the messages going through it are persistent so that in case either RabbitMQ or the queue crashes, messages won't get lost since they are stored on disk.

Having enumerated the common principles for both execution modes of jobs, Figure 9 and Figure 10 demonstrate them and how they change the architecture implemented.

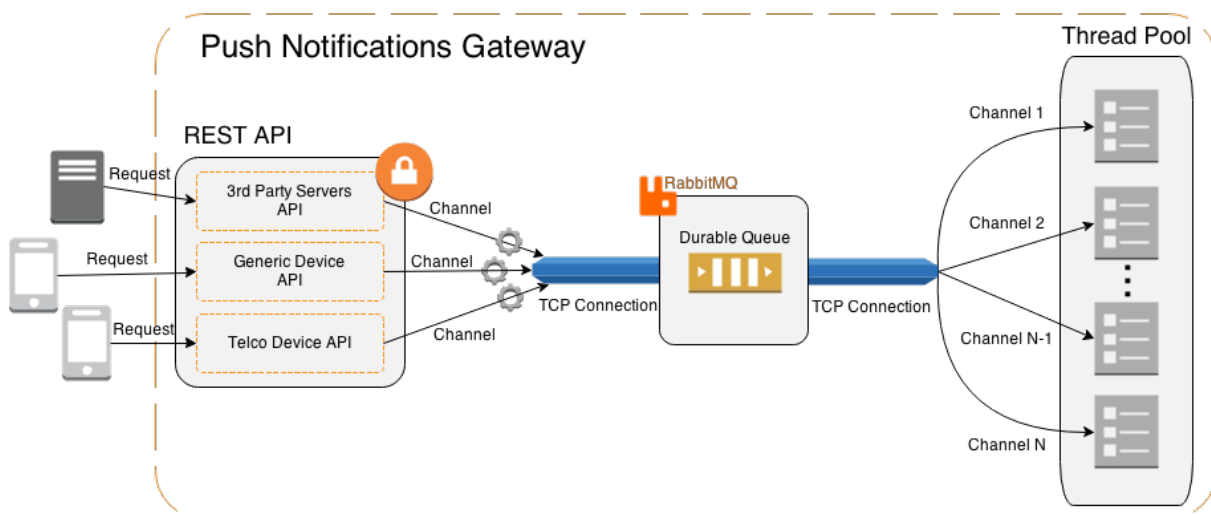


Figure 9 – RabbitMQ Asynchronous Job Architecture

In this scenario, the requests arrive at the corresponding API and a Job is created for each one of them. Further, a channel is allocated in order for the Job to be delivered to the queue and afterwards, picked by one of the available threads to be processed. Since this is the asynchronous use case, when a job is pulled into the queue, the response is immediately send to the requester with information that its request was accepted for processing.

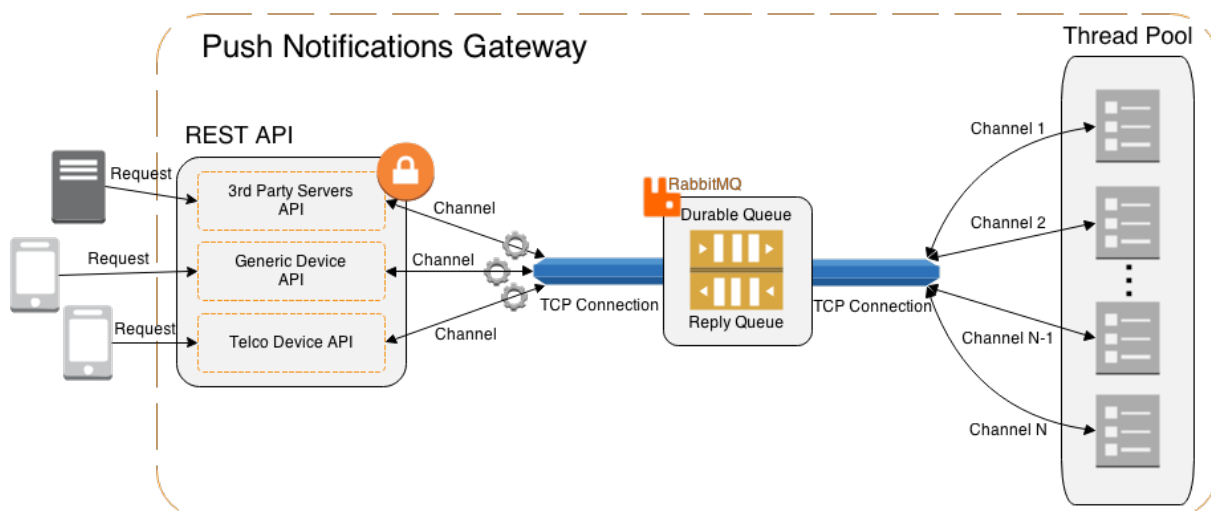


Figure 10 – RabbitMQ Synchronous Job Architecture

In the synchronous case there's a slight difference in the way jobs are executed. When a job is pulled into the queue the response is not sent to the client immediately, instead, the channel will block waiting for the job result and only then it will return the response. After the working thread in the thread pool ends the execution, it will write the result to the same channel that delivered the job. The channel will then pull the response to a reply queue, specifically allocated to that request, which will then unblock the channel waiting for the response in the API side. In this stage, the channel and the corresponding reply queue are closed and the result is returned to the requester. It is important to note that although a new queue is presented here (reply queue), there's no new connection being created. The same TCP connection is used to connect to RabbitMQ, so it can be seen as if it is shared between both queues. In this synchronous mode, the jobs have the highest priority, which means they are the first to be processed. This decision was taken in order to assure fast responses to the clients that are blocked waiting for them.

4.3.1.4. Consumer Threads

The whole processing phase occurs in the consumer thread that acquired the job. Each one has its own channel to the unique TCP Connection which connects to the RabbitMQ broker, which is established only one time, when the thread is created. This decision is explained by the fact that starting a new channel on an existing connection involves only one network round trip, but the inverse, starting a new connection, takes several. At the same time, each connection uses a file descriptor on the server, channels don't. Also, as recommended in [22], it is a good practice to separate publishing and consuming connections, which is the applied here since in the API side, the producers use another connection.

Threads run inside an Executor Service, which is a useful Java mechanism used for managing pools of threads. It's important to acknowledge that this is the central unit of the gateway when it comes to process jobs. Most jobs are entirely executed in the thread that acquired them, except push requests, which could take some time to process and would cause the thread to block for too much time.

Each consumer Thread starts to create a channel, in the already existing connection, to the queue to which it pretends to connect. The channel is also configured with a *prefetchCount* of 20, which is the maximum number of messages that can be delivered to this consumer before requiring acknowledgements. When a message arrives, it is immediately acknowledged to RabbitMQ and deserialized into a Job object again, being its status update to *PROCESSING*.

In this phase the execution depends on the job type, if the consumer is handling an asynchronous one it will just call its *execute()* method and set its status to *COMPLETED*, otherwise it will also receive a result with the data requested which will also need to be published back to the queue. This happens when the job executes successfully, which is not always the case. When it doesn't, the consumer will receive the exception thrown, store it in the Job and update it in the database along with the new *FAILED* status. When the processing ends, the thread will block waiting for another message to arrive.

Every error happening during the execution is thrown back to this central unit in order to ease the logging of problems.

For a deeper analysis of the actions taken by a consumer thread the reader should see the sequence diagram in Appendix C Figure 15.

4.3.1.5. Database

In order to store data in PNG, a database model had to be carefully thought in order to meet the goals and criteria established. The chosen database is relational and is implemented in postgresSQL.

4.3.1.5.1. ER Model

In Figure 11 is represented all database tables and relations.

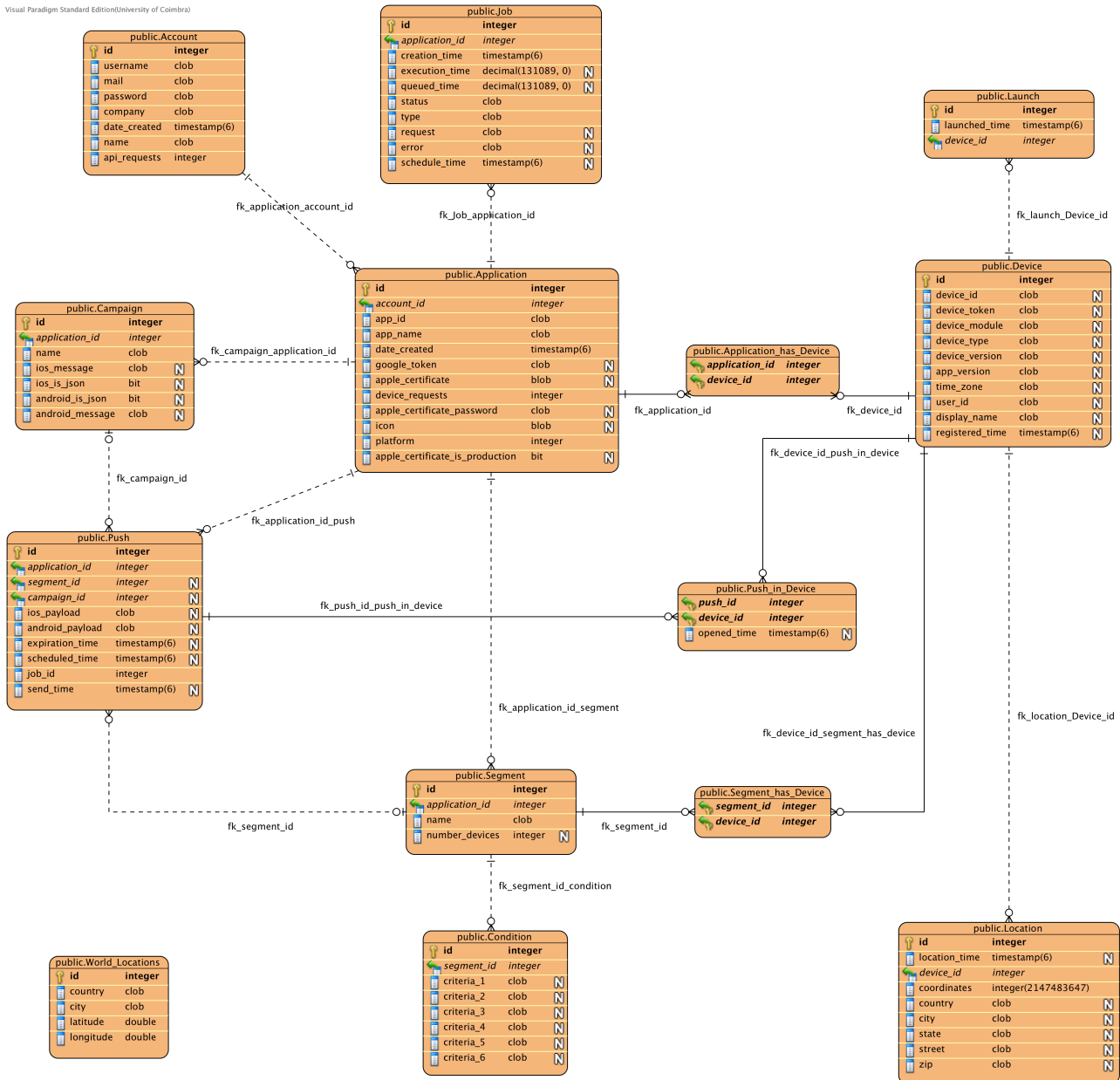


Figure 11 – Database ER Model

The “Account” table is responsible to store information about accounts created by application owners in the Web UI. It is mandatory that every application owner creates an account in order to use the gateway and its associated features.

Then there’s the “Application” which has a 1-to-n relationship relatively to “Account”. This table allows an Account to have multiple apps, which are identified by an *APP_ID*. It stores information

about Apple certificates, Google API Keys, the application icon represented in bytes among other application-related data.

As said before, every time a request arrives in the API, it creates a job, which will then be forwarded to a queue in order to be processed. This is the reason why every Application has a 1-to-n relationship with the table Job. In this table, every request that results in a job is stored along with its associated details as described in 4.3.1.2.

In addition to the Job table, an Application can also have one or more segments associated with it. Each segment is created by the application owner and results in a set of devices for which the transmission of a single push notification is facilitated. As expected, there's also a relation of n-to-m from Segment to Device, since a device can be associated with one or more segments and vice-versa. Still in the segments subject, it is possible in the web app to re-evaluate the segment creation criteria. For example if I create a segment and define it to include all devices in a certain city, this segment will get outdated as soon as new devices from that city get registered in the gateway and are not yet associated with the segment. In order to re-evaluate these conditions they must be stored, and that is the purpose of the table Condition.

An application can also have multiple devices associated to it, to whom push notifications can be sent. All the contents of the Device table are filled with information received from devices upon their registration. Also, information like location and launch history can be associated with a device, which is why there's a relation of 1-to-n to these tables. They store all the history of the user's locations and every time the user opened the application.

Table "Campaign" stores information about pre-defined push messages which can be reused anytime by the application owner. As so, it is also related to the "Application" table since one of these can have multiple campaigns created.

Push notifications sent are stored in table "Push". This table has 1-to-n relations to Campaign, Segment and Application. The latter is the only one that is mandatory since a push notification is always sent to a specific application. Campaign and Segment relations can exist or not, depending if the Push sent used a previously created Campaign or targeted a specific Segment. Since a Push notification can target multiple devices, it is important to keep track of those who receive them. As such, there is a last m-to-n relation to the table Device. As part of this last relation, the intermediate table, which decomposes the m-to-n relation, has an additional attribute, the "opened_time". This variable stores the date and time (if applicable), in which that specific device opened that push notification.

"World_Locations" table has data about all the main cities of the world countries.

4.3.1.5.2. PostGIS

When it comes to create new segments, there are different criteria that can be applied to create a resulting set of devices covered. One of these criteria allows defining a set of coordinates and a radius. In order to be able to determine which coordinates are inside that set, PostGIS is used. It is a spatial database extender for PostgreSQL, which adds support for geographic objects, allowing location queries to be run in SQL.

Table "Location" has a column named "coordinates" which type is *geography(Point, 4326)*. The geography type stores spatial features represented in "geographic" coordinates, sometimes called "geodetic" which are latitude/longitude pairs. Geography was chosen in order to get the best approximation to real world calculations since it is based on a sphere where the shortest path between two points is an arc and not a straight line. 4326 is the identifier for the WGS-84 standard coordinate

system for the earth. In short, this object stores latitude/longitude pairs in geographic coordinates in relation to the coordinate system WGS-84.

PostGIS also provides functions to calculate if a given coordinate is within a certain radius and always in regard to the spherical shape of the earth.

4.3.1.5.3. Dataset of countries and cities

In order to provide a list of available countries and cities from which users can choose to create segments, this information had to be obtained and stored in the database. For this purpose, a dataset available in [23] was used which contains all the main countries and cities of the world. This information was inserted in the “World_Locations” table.

When devices send their location, besides sending their coordinates, they also reverse geocode them in order to send the country and city on where they are.

When users create segments based on a country and a city, the algorithm will parse all devices whose country and city match the one chosen, by comparing the “World_Locations” country or city name with the reverse geocoded on the device.

4.3.1.5.4. Security

Since there is confidential information on the database, it is important to assure that it is well secured. For this purpose, the default access credentials to the database were changed to guarantee a higher-level of security. However, this is not sufficient as the credentials can be discovered and the database file can be stolen. As such, the following values are stored encrypted: apple certificate and password, Google API Key and accounts’ passwords. To achieve this, the Jasypt encryption library was used.

Although this valuable information is already safe, it was also decided to protect the application from SQL injection attacks. For that purpose, the gateway’s code uses *PreparedStatement* to build the queries it needs to perform. In this way, if the user inputs an executable SQL command, it won’t be executed since it is converted to a data type first, for example, a string or a number.

4.3.1.6. Quartz Scheduler

Before beginning this topic, it is important to clarify both Job definitions. There are Jobs which belong to the gateway architecture as described in 4.3.1.2 Jobs, and there are Quartz Scheduler Jobs which are classes that implement Quartz's own Job class.

Job scheduling is essential when a task needs to be performed on a specific time. For this purpose, a scheduling engine was created and is based on Quartz, the Job scheduling library. Quartz scheduler can be seen as a separate container where jobs defined for later execution are put and wait for firing.

Among the features that this library offers are:

- Job Persistence: Quartz can store jobs either in a relational database via JDBC or in RAM.
- Transactions: The execution of a Job can be wrapped in a JTA transaction, assuring that a Job happens in its totality or not.
- Clustering: Quartz can run in a load-balanced environment and also provides fail-over mechanisms.

The entire configuration is defined on a *quartz.properties* file that has the following content:

- org.quartz.scheduler.instanceName = Scheduler
- org.quartz.threadPool.threadCount = 3
- org.quartz.jobStore.class = org.quartz.simpl.RAMJobStore

Quartz instance is a singleton, being only instantiated once and living for the whole execution time. Whenever this instance needs to be obtained, *instanceName* needs to be provided. The thread pool managed internally has 3 threads which means that a maximum of 3 jobs can be run simultaneously. As a last configuration, all scheduled jobs residing in Quartz are stored in RAM.

There are two main definitions used when creating a new Quartz Job:

- JobDetail: The object that declares the Job class that will execute.
- JobKey: Identifier of the Quartz object being created, which is a name and group pair.
- JobDataMap: An HashMap to store data to pass with the Quartz Job. Needs to be attached to JobDetail.
- Trigger: This is the mechanism by which Jobs are scheduled. It defines when the Job will be executed, and has multiple scheduling options such as one-time or regularly executions.

For example, when a scheduled push request arrives at the server, a Job object is created with a *jobId*. To begin the scheduling process, a *JobKey* is initiated with the *jobId* and as part of the "push-notifications" group, and the Job object is stored inside a *JobDataMap*. Finally a *JobDetail* is created, setting the execution class (*PushToRabbitMQJob*), the *jobDataMap* and its *jobKey*. To fire this *jobDetail* in the right time, a *Trigger* is instantiated with the *scheduledTime*. Both are then scheduled to Quartz, and the Job will update its status to *SCHEDULED*.

The following three sub-chapters describe the goal of each Quartz Job.

4.3.1.6.1. PushToRabbitMQJob

Although this Quartz Job is generic and can be used to publish every type of job to the RabbitMQ queue, it was created with scheduled pushes in mind, since they are the only type of gateway jobs that can be scheduled. This class's goal is to publish jobs to RabbitMQ in order for them to be processed.

Following the example given in 4.3.1.6 Quartz Scheduler, when the time for the push job to execute comes, this Quartz Job gets fired. The push job represented in bytes is obtained from the *jobDataMap* and de-serialized back to the Job Object, in order to update its status to *QUEUED*. After performing the update in the database, the Job is serialized again to its byte representation and inserted into the RabbitMQ queue to be executed.

So, this can be seen as a bridge between job scheduling and execution. Everything that is inserted into the queue is immediately processed which is why scheduled Jobs must be transformed into Quartz Jobs and live in its container until they are ready to be processed. When they are, *PushToRabbitMQJob* is responsible to ship them into the queue which will then be consumed by one the threads available.

To understand the flow of execution of this Quartz Job with a detailed sequence diagram, please refer to Appendix C Figure 18.

4.3.1.6.2. LoadScheduledJobs

Since Quartz jobs are stored in RAM it is important to control the amount of memory being occupied whenever too much push jobs reside in this component. Once all scheduled jobs are immediately sent to Quartz, it is perceptible that memory starts to get saturated quickly. In order to solve this problem this Quartz Job is executed when the server starts and from then on, in an hourly-basis. It has two main goals:

1. Remove from Quartz every scheduled push job (belonging to the *push-notifications* group).
2. Get from the database all jobs with status *SCHEDULED* that should have been already executed (this handles the case where the server was down for a long time and jobs that should have been executed during that time wasn't), and all jobs that are *SCHEDULED* to execute in the next hour.

With this algorithm, we assure that even if a lot of push jobs get stored in Quartz, an hour after they will get all cleaned out and only the ones that need to be executed in the next hour will be loaded.

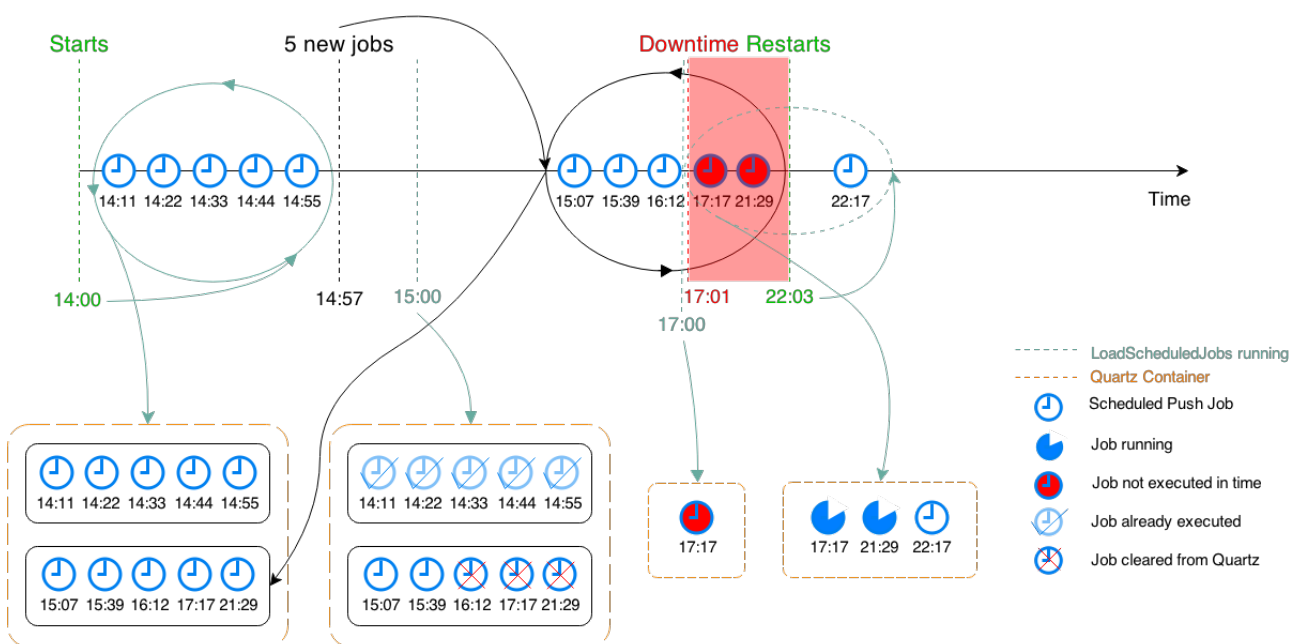


Figure 12 – LoadScheduledJobs algorithm overview

14:00H

Server starts, LoadScheduledJobs is fired and scheduled to execute every hour from that minute on. It parses the database and obtains the push jobs scheduled to execute in the next hour. These 5 are then stored inside a PushToRabbitMQJob, which is loaded into the Quartz Scheduler waiting to be fired.

14:57H

At this time, 5 new push jobs are created and, since none of them are of immediate execution, they are all put inside the Quartz container. By adding all new schedule jobs to Quartz disregarding what their scheduled time is, we avoid the need to maintain information about when the next LoadScheduledJobs will happen in order to only insert jobs into quartz that are to be fired before that time. In this diagram, although none of the jobs will execute before the next LoadScheduledJobs check (at 15:00H), if there were jobs to be fired in the next 3 minutes they would successfully be executed.

15:00H

LoadScheduledJobs is executed again. By this time, the 5 jobs between 14:00H and 15:00H already executed so they are no longer in Quartz. All Quartz Jobs belonging to *push-notifications* group are cleared, and the ones scheduled to run in the next hour are loaded, which in this case is the 15:07H and 15:39H jobs.

16:00H

LoadScheduledJobs runs and the 16:12H job will be loaded to Quartz and will execute in that time.

17:00H

Another hour passed since the last loading of scheduled jobs from the database, which means it is fired again. There is one job to be executed in the next hour, which is loaded into the container, although it won't execute.

17:01H

The server crashes and the only Job living in Quartz is lost (Quartz Jobs are stored in RAM).

22:03H

The server is back online and will run LoadScheduledJobs as part of its initialization protocol. In this time, since there are jobs scheduled for the past, which are still in the *SCHEDULED* state, they are loaded into Quartz for immediate execution, and the only job scheduled to fire within the next hour is loaded too for later execution.

Figure 12 represents a global overview of what is done by this Quartz Job during the server uptime. To give the reader a better understanding about the inners of this task, please see the sequence diagram in Appendix C Figure 20.

4.3.1.6.3. AppleFeedbackService

Apple's feedback service allows fetching information about device tokens which are no longer valid. For example, when a push notification failed to be delivered because the intended app was uninstalled from the device, feedback service adds that device's token to its list. In this way, we can remove these identifiers from our database and henceforward, improve system performance by reducing unnecessary message overhead.

Apple has two feedback services, one for sandbox and another for production servers. Initially all applications that have apple certificates are obtain from the database along with their passwords. For each one, a connection is made to the corresponding feedback service server and the inactive devices are pushed from the stream. With this list of devices, the database is updated by removing all the invalid tokens. The Device entry in the database table remains with all its associated information such as location and launches, being only the token removed.

The format of the data received from the feedback service is the following:

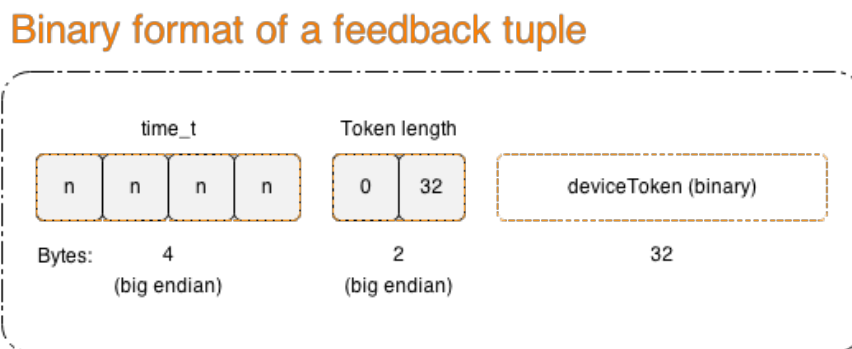


Figure 13 – Binary format of a feedback tuple

First 4 bytes define the timestamp when APNs determined that the application no longer existed on the device. Value is in network order and represents seconds passed since 00:00 of January 1, 1970 UTC.

The next two bytes are an integer value in network order representing the device token length.

The device token is in binary format and has a maximum of 32 bytes.

Each tuple, which in turn represents a device, has a total size of 38 bytes. If the feedback service has more than one device no longer valid, we will downstream the complete 38 bytes tuple times the number of devices. When we finish reading the stream, the feedback service's list is cleared and the next time we connect to it we will only have information about devices found invalid since this time.

For a sequence diagram of this Quartz Job please visit Appendix C Figure 21.

4.3.1.7. Push Engine

PushEngine is responsible essentially for delivering each target device to its specific engine, Apple's or Google's. When a notification request arrives at the server, after being parsed and analyzed, it often results in a big set of target devices. Suppose we choose to send a notification to a segment previously created which has 5000 devices associated, *PushEngine* will divide them into two groups according to their operating system and send them to the specific engine.

PushEngine has two methods, with equal *modus operandi*. The only difference is that one sends the same payload to both engines and the other has different ones for each platform (Apple and Google).

The *PushEngine* serves as a bridge between the *APIPushJob*, which is where the Job is acquired from the queue, and each platform-specific engine. Not every request passes in this bridge since it is possible for them to specify the target platform. When there's guarantee that the resulting devices belong only to one platform, the *APIPushJob* can immediately invoke the corresponding platform engine. Figure 14 presents four examples that explain this statement.

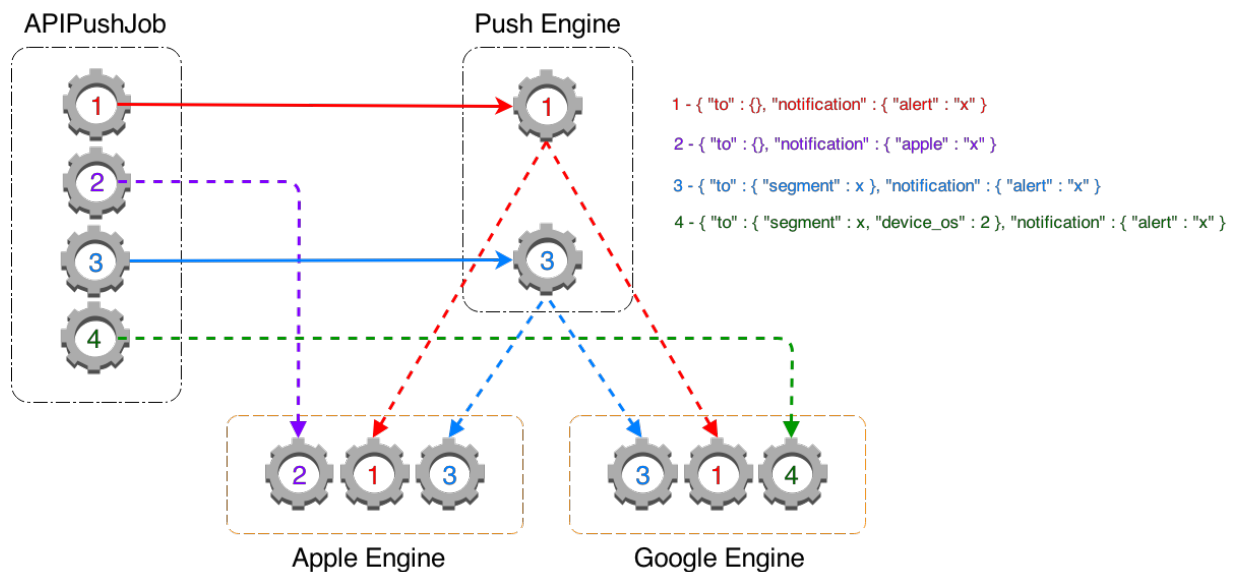


Figure 14 – APIPushJob decisions

These four push request examples, among the various possible, seeks to show common cases where *PushEngine* needs or not to be used.

The first request does not specify any audience (all devices are assumed) and declares a common simple notification for both platforms. In this case, since *APIPushJob* doesn't know if this application has devices, it calls *PushEngine* which will group them according to their platform. From then, if there are devices of any platform it sends their tokens to the specific engine. Remember that it is always possible that only one engine is called, suppose that there are only Google or Apple devices registered for the application in question.

The second request also does not set any audience, but only specifies a payload for apple devices. With this, even if the application has Google devices they are ignored since only Apple payload is specified. As such, Apple Engine can be immediately called with the application's iOS device tokens.

Third request declares a notification common for both platforms and targets a specific segment. Once *APIPushJob*, after retrieving the segment's devices from the database, doesn't know if this group contains

both or one of the device's platform, it sends them to *PushEngine*. At this moment devices form platform-specific groups and will be sent to the according engine.

In request 4, although the notification is common for both platforms and the segment filter can have also both operating system devices, this doubt is erased because of the "device_os" platform filter. This indicates only Android devices, as such, we can extract these from the segment and send them immediately to Google Engine for delivery.

To comprehend all the steps taken by this component, please see the sequence diagram available in Appendix C Figure 23 along with the description provided.

4.3.1.7.1. Apple Engine

Apple Push Engine is the logic module responsible for handling APNs-related tasks, from Payload construction and communication to exception handling. This engine performs three main steps along its execution:

1. Multithreaded engine set up
2. Compose and deliver binary interface compatible notification
3. Process failed notifications

Once they all have much implementation details to be explained, they will be divided in specific sub-chapters.

4.3.1.7.1.1. Multithreaded Engine

Apple Push Notification service uses a streaming TCP socket design for asynchronous communication. With this architecture, Apple allows to establish multiple connections to the same or to multiple gateway instances. This comes in handy especially when there's a need to deliver a large number of notifications, which can be achieved by spreading them over several connections. By doing this, performance gets improved in comparison to a single connection usage.

Thought it seems simple to open several connections and send batches of notifications through them, the scenario gets much more complicated. In a perfect world the gateway would have a fixed number of connections always open, and applications wanting to deliver notifications would use them.

In the first place, connections are directly related to a specific application because in order to establish them, the app's certificate and password must be provided. This does not seem to be an obstacle since we can open a couple of persistent connections for each application registered on the server, and whenever each of them needs to deliver notifications it just makes use of them. Unfortunately Apple doesn't allow to establish an infinite number of simultaneous connections. Although they don't specify a number, after some Internet research it can be found that one should not have more than 15-20 connections opened [24][25] because APNs will treat it as a denial of service attack and will block the connections for a period of time.

Due to these limitations, connections are established on a per-request basis and up to a maximum of 10 simultaneous per-application. For example, if a push results in 5000 devices to target, a maximum of 10 connections will be established each of them delivering 500 notifications. By limiting this number of connections per-application, we grant the possibility of other applications to send notifications at the same time since there is still room to open more connections until Apple starts to close them.

But how do we know the ideal number of connections to open (each thread holds a connection) in relation to the number of devices to deliver notifications to? There is another limitation which helps to define this number and that was found after much trial and error, which is the maximum number of messages that an SSL connection can deliver before starting to fail randomly. Two hundred notifications were found to be the magical number of notifications that can be streamed over a continuous connection to an Apple server without incurring in random socket errors. When that maximum is reached, the thread automatically closes and reopens a fresh new connection to the server and continues streaming notifications in order to assure reliable delivery. Since reopening a connection takes time, the goal is to prevent that from happening.

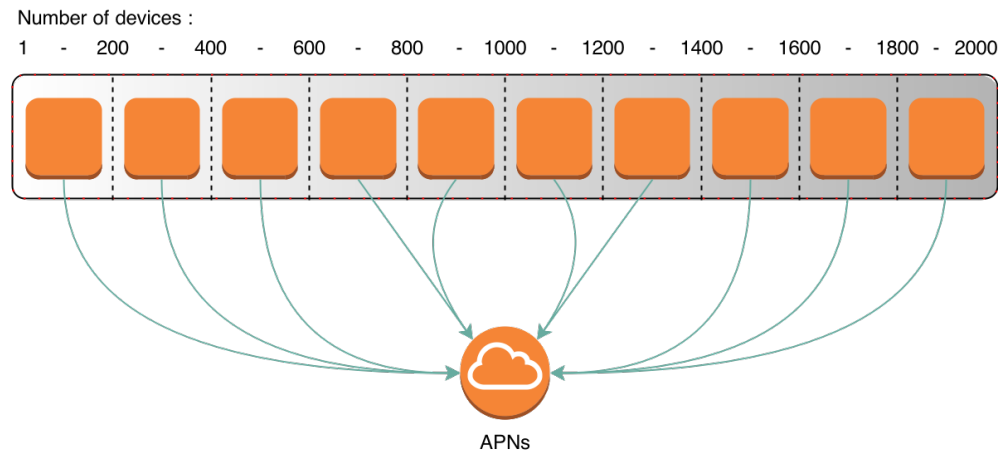


Figure 15 – Optimized Thread and Connection Number

As Figure 15 suggests, for each 200 devices a thread and connection is created until a maximum of 10. When there are more than 2000 devices, each connection will need to stream more than 200 notifications and thus, reaching this number, will have to reopen the socket in order to avoid the previously mentioned errors. This algorithm ensures that up to 2000 devices, it is possible to allocate a number of threads (10 is the maximum) that allows the gateway to not close and reopen sockets.

In short, these are the steps taken by Apple Engine before starting to stream the messages:

1. For a given *applicationId*, the apple certificate, password and type (sandbox or production) are obtained from the database.
2. Following the defined in the request, the push payload is created.
3. An ID is generated for this push and inserted in the payload. It will be used to acknowledge the opening of the push notification on the device in the server (push conversion).
4. The optimized thread number is calculated following the logic associated with Figure 15.
5. According to the number calculated, the threads are created and the devices are equally distributed between them.

These 5 steps can be further analyzed in the sequence diagram presented in Appendix C Figure 26.

4.3.1.7.1.2. Binary Interface compliant message composition and delivery

This second step involves the execution of each individual thread to process the notifications. For each one of them a socket connection to Apple Servers is established. Then, for each target device is defined a *messageId*, which will be useful to identify the error response packets as it will be described forward. Finally the construction of the binary interface compliant stream is created and sent through the socket for each target device. If any socket-related error occurs, we create another one and retry the deliver up to a maximum of three times. Between each notification streamed is checked if the device just

processed is multiple of 200 and thus if the connection needs to be restarted (as explained in 4.3.1.7.1.1). If it is, step 4.3.1.7.1.3 needs to be performed first. It also needs to be executed before closing the connection, which is after all the notifications are sent.

It is important to explain how the binary interface works and thus, what is really delivered to the socket. There are two different notification formats, which will henceforward influence the binary stream composition. We will only focus on the Enhanced Notification Format which is the most recent and implements error response packets and notification expiration.

Unlike the simple format, the enhanced gives per-message feedback if an error occurs. It allows tagging a specific notification with an ID (the *messageId*). If there is an error, APNs returns a packet with an error code associated with the provided *messageId*. With this association it is possible to detect, and possibly act upon the failed notification.

Regarding the notification expiration, with this format it is possible to define an expiration time and thus allow the APNs to store the message and deliver it when the device becomes available, in case it isn't by the time of the request. With the simple format the notification would be delivered regardless of the device status and as such it could get lost.

Enhanced Notification Format

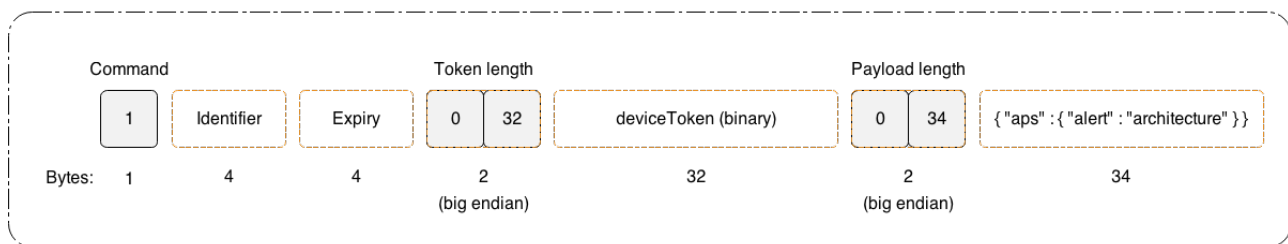


Figure 16 – Enhanced Notification Format

The identifier is the *messageId*, which will be returned in case APNs cannot interpret a notification, and responds with an error. Expiry identifies when the notification is no longer valid and must be discarded. It is a fixed UNIX epoch date expressed in seconds. The next two blocks define the token length and its binary form. To finish the notification construction, the payload length, which cannot exceed 256 bytes and should not be null-terminated, needs to be specified along the actual payload data.

In Appendix C Figure 27 is a sequence diagram describing all the steps performed by the gateway when executing each one of the threads.

4.3.1.7.1.3. Failed notifications processing

When a thread finishes streaming all notifications, before closing the socket connection, it needs to check for the existence of errors.

With the enhanced notification format it is possible to obtain feedback regarding per-notification errors. Whenever a malformed notification is sent to APNs, it returns an error-response packet before silently closing the connection on its side. The problem is that it can take a while for the Apple's dropped connection to be perceptible by our gateway because of normal latency. Between this period, several notifications can be sent through the socket although they never reach their servers, being left in a limbo with no error-response packets.

The easiest way to solve this problem would be to asynchronously read the socket for the existence of error-response packets. Although, waiting between 100 and 500ms for a possible error after sending each notification would kill the throughput performance intended.

As such, the solution implemented waits for a write on the socket to fail or for all the notifications to finish streaming before checking for error packets. If the socket fails to write, and after reading it there is error packets, every message following the one associated with the error (can be found by looking at the *messageId* assigned) will be sent through a new socket connection. In case there are no errors when reading the socket, a new connection is created too and the batch of notification continues to be streamed normally. Since even if there is a notification sent with an error, it is possible in the meanwhile that all the remaining are written through the socket without acknowledging its closing on Apple's side, and therefore knowing immediately that they will be discarded, the error-response packets are verified in the end of the transmission too. Thus, it is guaranteed that in both scenarios, no notifications are left forgotten for processing.

The socket is only closed on our side after reading it for error-response packets.



Figure 17 – Error-response packet

This packet has a command value of 8 followed by a one-byte status code, which corresponds to a specific type of error. The last four bytes specify the identifier of the malformed notification, which was assigned to it when it was sent to APNs (*messageId*).

4.3.1.7.2. Google Engine

Google Cloud Messaging integration is much more simple than Apple's. Communication with this platform is done with HTTP POST Requests and the deliver of notifications is made in batches of 1000 devices maximum.

First step is to get from the database the application's Google token. Then, the payload is created with the content of the notification to be sent. Google pre-defines the "data" key for its JSON payload, which means that all data send to the device needs to be put inside it. When the notification content set by the user is just simple text, it will be associated with a "content" key, which in turn will be a value of the mandatory "data". Otherwise, by specifying the JSON payload, it will be sent as is inside the "data" key. Following the same principles of Apple Push Engine, regardless of the payload specified, every notification carries an ID put inside "data", which will be used to acknowledge the opening of the push notification on the device in the server (push conversion).

Since the maximum number of devices to which a notification can be delivered in the same HTTP request is 1000, there's a mechanism that calculates the amount of requests needed to dispatch all the notifications. For each set of 1000 devices, a thread is created and the devices are spread equally for each one of these.

The HTTP request body is composed by all the JSON inside the mandatory “data” key, the “registration_ids” (target devices) inside an array with comma-separated strings and, if pretended, a “time_to_live” which specifies how long the message should be kept in GCM store if the device is offline. After performing the request, the response is read and if the status code returned is a non-200 we retry the sending process until a maximum of 3 attempts, always honoring the exponential backoff times. If the status is a 200 OK we will parse the response body “results” key in search for individual messages that failed with 500 (Internal Server Error) and 503 (Unavailable) codes, which will then be grouped and send again. For this group of notifications retried, the process will repeat itself in search for new failed messages that need to be sent, up to three attempts.

In the end of the available retry attempts, if used, an aggregated result is created with the responses received from the GCM server. With this, it is possible to analyze each notification sent and perhaps update *registration_ids* with new *canonical_ids* provided (the new id for that device), or even delete no longer valid ones. This last step is the only way to clean invalid *registration_ids* since Google doesn’t provide any feedback service like Apple does.

Figure 30 in Appendix C represents in a sequence diagram what has been described.

4.3.1.8. Web Application

The entire website interface was done using the well-known bootstrap front-end framework. As so it is also available with a responsive layout, auto-adjusting the content weather the user is on a desktop, tablet or phone. I designed the Web UI and UX.

The web application was made with Struts 2, a web development framework. It was designed with the MVC model in mind where views present the content, controllers process them and models create object representations.

In this framework there is a central file where all the URL mappings are declared, which is called *struts.xml*. All the *.jsp* files are mapped to an action, which means that when a user enters the site URL followed by */register*, Struts is actually mapping this endpoint to *register.jsp*. The same is done for forms and Ajax requests where, for example, a *POST* to */SavePushCampaign* is mapped by Struts2 to be handled by *GetMyPushCampaignsAction* class *execute()* method.

Another really handy feature of this framework is the use of interceptors. When a resource is requested, it is mapped to an action in order to be processed. Between these two actions, an interceptor can execute code in order to validate or perform intermediate tasks. In the gateway this serves the purpose of checking for a user’s valid session. When a URL that needs login credentials to be accessed is invoked, the *LoginInterceptor* checks to see if the user is logged, granting him permission to access that URL, or denying it.

Figure 18 was created in order for the reader to have a global overview of the web application project’s structure.

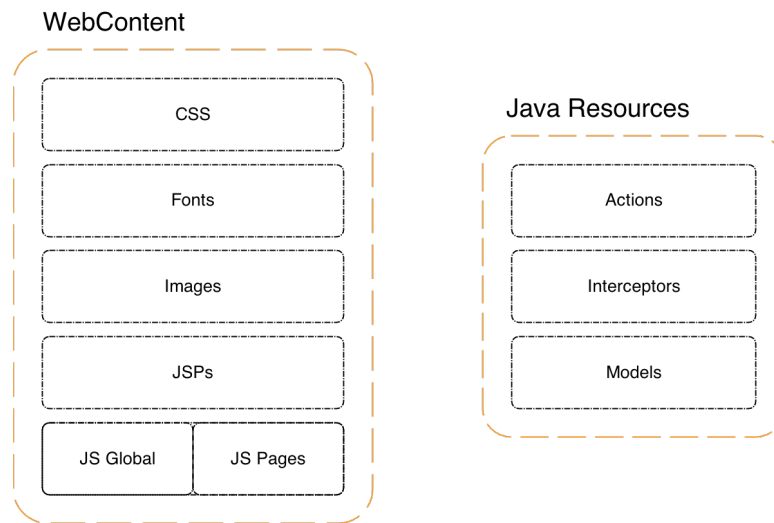


Figure 18 – Web App Project Structure

The *WebContent* part holds several types of content. The CSS files are responsible for describing the look and format of the JSPs. Fonts have a set of icons that are used in the back office, and Images have a few static banners used in the home page. In the JSPs folder resides the entire HTML. Finally, the JavaScript folder has two sub-directories, the global which have the files used in various JSPs and the pages that have individual JavaScript files for each JSP.

In the Java Resources are all the *Actions* mapped by Struts, the Interceptors and the Models.

Table 12 and Table 13 enumerate all the existing JSPs and JavaScript files along with a description.

JSP	Description
account	View and edit account details
api-requests	API Requests chart overview with filtering
app-opens	App Opens chart overview with filtering
dashboard	Central website page showed after login
devices	Device details and requests overview with filtering
index	Start page
login	Login to access back office
new-app	Create a new application
push-advanced	Create and remove push campaigns
push-notifications	Push notifications chart overview with filtering
push	Send push notifications
register	Register new account
segments	Create, delete and update segments
settings	View and edit application’s settings

Table 12 – JSPs and description

JavaScript	Description
bootstrap-datetimepicker	A date time picker used on schedule components.
bootstrap-switch	A switch used to turn on or off the JSON option
bootstrap	Custom jQuery plugins to animate the UI
daterangepicker	Date range picker for chart filtering
json2	Validates JSON input
jsonlint	Auto-indents JSON input

JavaScript	Description
jQuery	Fast, small, and feature-rich JavaScript library
minify.json	Minifies JSON input
moment	Parse, validate, manipulate, and format dates
morris	Chart Library
raphael	Vector graphics library
typeahead	Auto-completer

Table 13 – Global JavaScript files and description

Almost all the requests made to the server are done asynchronously with Ajax. This is done to leverage the user experience by retrieving data without refreshing the entire page. A big part of these requests are done without the user being aware of it, and to pre-load content that will be displayed to him if a certain action occurs. For example, there are various dropdown lists whose contents needs to be parsed from the server, but instead of loading them when the user clicks on the list, it is asynchronously loaded when the page is opened and thus, giving a fluid and responsive experience.

In order to prevent CSRF Attacks, the server only accepts Ajax requests that were triggered by a real request of this type. For example, if a user performs a request to /GetAppSettings, which should normally be done by the page's JavaScript code but instead is made from URL direct access or any other way, it will be discarded. When a browser makes an Ajax request, a special header is set, which is *X-Requested-With*. This header is non-standard but is used by the jQuery plugin that is being used. When the request arrives at the server, it just needs to check the existence of this header and that its value is *XMLHttpRequest*. Although this header and value can be added, due to same-origin restrictions, *XMLHttpRequest* doesn't allow to make Ajax requests to a third party domain by default and as such it is not possible to forge a request with a spoofed *X-Requested-With* header.

Data stored in the HTTP Session is very important for the website's security. For every request made that involves manipulating data on the back-end, it is verified if that data is within the boundaries of that user's permissions. For example, if a user requests to delete segment 19, before doing that in the back-end it is verified if that segment belongs to the *applicationId* in use and in turn if it belongs to the *accountId* logged in. These two variables are stored in the HTTP Session and thus cannot be overridden. This prevents attacks where the DOM is manipulated to submit values outside the application's scope.

Another important aspect worth mentioning is the attention given to the website performance. For this purpose, Steve Souder's rules were taken into account. There is one rule although that was not followed, which is "Gzip Components". Since connections are encrypted with HTTPS, due to vulnerabilities such as CRIME and BREACH, compression is not used.

4.3.2. Container

The whole environment of the gateway runs inside a Tomcat 7.0.39 instance. In addition to the web application, every other component resides inside this Java HTTP Server. The main reason why this was the chosen container is because, in addition to being free of use, it is updated regularly, which is a plus in terms of ensuring security in this layer. Another important aspect of Tomcat is that it can be set up in a load-balancing environment with very simple changes.

4.3.2.1. Initialization

There are seven steps that must be taken to ensure correctly initialization of the instance, which are:

1. Jersey REST Service
2. Quartz Initializer
3. PostgreSQL definition
4. Initialize RabbitMQ Consumer Threads
5. Initialize RabbitMQ Publisher Connection
6. Struts 2 initialization
7. Bootstrap configurations

Jersey REST Service is a servlet which has all the features needed for the REST APIs to work. It scans the package *com.witsoftware.png* upon startup, in search for classes and methods with Jersey Annotations. It also defines the common *url-pattern* for which all resources can be accessed, which is */api/**.

The initialization of the Quartz Scheduler singleton is also done upon the tomcat startup. It also runs as a servlet and is configured based on the *quartz.properties* file configurations. In general it handles the creation of the 3 threads within the Thread Pool and sets the storage mechanism to the RAM. It also has instructions to auto-unload all of its instances when the tomcat is stopped or restarted, which prevent memory-leaks from happening.

The PostgreSQL JDBC drive is loaded and initialized in order to perform database operations. This is done by referencing the resource which details all the configurations needed for the database connection, which can be found on Tomcat's *context.xml*.

The *ExecutorService* with the 20 RabbitMQ Consumer Threads is initialized. This process includes the creation of the TCP Connection to the RabbitMQ broker, which is shared by all threads, and also the individual channels created by each thread to the queue.

The TCP Connection to the RabbitMQ broker, which is shared by all producers, is also created on the startup. When an API endpoint, in response to a request, needs to publish a Job to the queue, it needs to obtain the Singleton Publisher TCP Connection.

Struts 2 is configured as a Tomcat filter, which maps every request made to */** to this framework's actions (inside its configuration it ignores requests to */api/** since they are handled by Jersey). Tomcat is also instructed to lookup for Struts 2 action classes inside the package *com.witsoftware.png.struts.controller*.

These six steps are executed in the same way disregarding the application's environment. There are two environments where the application can live, development and production, and these influences the way Bootstrap is run.

First step performed by Bootstrap is the initialization of Paths needed by the application. Here are declared the paths according to the environment in execution for: tomcat deploy path, XML schemas

needed to validate requests made to the Telco API, log4j properties file and output log file and finally the default application icon which is used in the web dashboard when none is provided.

Second step, which only happens if the application is set for production, is the execution of the Grunt.js script. Grunt is a JavaScript task runner that is built on top of node.js. There are many available plugins that can be installed to be used by Grunt which ease the execution of tasks that once had to be done manually. In this case Grunt is used to concatenate all individual .CSS files and .JS files into one, minify these two, and finally process all the html to replace all the individual import declarations into the global .CSS and .JS file created. The execution of this script is made by invoking a terminal command outside the application's scope.

Third step performed by Bootstrap is the initialization of the Log4j logging framework. Log4j.properties file is read and injected as it is into the *PropertyConfigurator* class only if the environment is set for development. If not, after reading the file all declarations for *stdout* printing are stripped out since we just need to log to the file. After doing this, the overall request process rate increased 13%.

As a final step *LoadScheduledJobs* and *AppleFeedbackService* are executed and scheduled to run every hour inside the Quartz Container.

4.3.2.2. Configuration

Tomcat has been configured to boost performance and security.

The web application manager has restricted access outside WIT Software's premises, which means it can only be accessed inside WIT's network or via VPN. In addition to this layer of security, the default access username has been changed along with the password.

Every communication with the server is done via HTTPS, although this is not configured in the Tomcat Connector. WIT Software has a reverse proxy server which maps demo.wit-software.com/PNG/ into the tomcat <server-ip>:8080. Since the reverse proxy server has HTTPS configured, the SSL Handshake is done with this server since it is the first to be contacted and as such, any configuration regarding SSL in tomcat is ignored. Configurations regarding the tomcat connector are then all made in the port 8080 and can be configured to be redirected to 8443 with SSL support, which will only be of any use if the access to the server is done directly with its IP address. Connectors are configured in the following way:

```
<Connector port="8080"
  protocol="HTTP/1.1"
  connectionTimeout="20000"
  URIEncoding="UTF-8"
  maxThreads="200"
  minSpareThreads="50"
  redirectPort="8443"/>
```

```
<Connector port="8443"
  protocol="HTTP/1.1"
  URIEncoding="UTF-8"
  maxThreads="200"
  minSpareThreads="50"
  scheme="https"
  secure="true"
  SSLEnabled="true"
  keystoreFile="/path/to/keystore.store"
  keystorePass="xxxxxxx"
  clientAuth="false"
  keyAlias="tomcat"
  sslProtocol="TLS"/>
```

Port 8080 connector is the one used when accesses are done via the mapped URL. It employs HTTP/1.1 and decodes URI bytes in UTF-8 (after %xxx). The minimum number of threads always running is 50, which means that tomcat can always receive simultaneously a minimum of 50 requests. This value takes into account the 35 push requests that the server should handle per second (as defined in NFR_16) and also leaves 15 threads to handle other types of requests such as web page accesses. Foreseeing this 50-request/second rate, the 50 threads are always up and running in order to, in normal workloads, being sufficient to handle requests without needing to instantiate new threads and create additional overhead. The maximum number of threads that can be running simultaneously are 200 which is the maximum number of requests that can be received simultaneously without compromising the server behavior. If the load is found to be greater than this number, a load-balanced environment must be considered.

When requests are made directly to the server URL, for example in a development environment, they are all redirected to port 8443, which has HTTPS support. In fact this configuration is only suitable for development environments since the certificate configured is self-signed and as such does not warrant great confidence for users. The remaining configurations of this connector are the same as the other one.

HTTP sessions are configured to expire after 30 seconds by setting the *session-timeout* in the web.xml file. After this time, any page or resource requested in the web application is forwarded to the login page.

One optimization made regarding web pages, that needs to be done in the server configuration (in web.xml) is the *ExpiresFilter*. This filter controls the setting of the Expires HTTP Header and the max-age directive of the Cache-Control HTTP header in server responses. Images, CSS and JavaScript files are the types subject to caching, and its expiration date is set to be relative to the time of the client's access to the document plus 10 days.

The last optimization is related to the way tomcat connects to PostgreSQL. The database is configured via a JNDI Datasource (as a resource). There are two database connection pool implementations that can be used, which are commons-dbcp and tomcat-jdbc-pool. The latter is the chosen one for the various performance reasons stated in [26].

```
<Resource auth="Container"
    type="javax.sql.DataSource"
    maxActive="200"
    maxIdle="80"
    minIdle="50"
    maxWait="-1"
    initialSize="50"
    removeAbandoned="true"
    removeAbandonedTimeout="60"
    logAbandoned="true"
    validationQuery="SELECT 1"
    validationInterval="30000"
    testOnBorrow="true"
    name="jdbc/postgres"
    username="xxxxxxx"
    password="xxxxxxx"
    driverClassName="org.postgresql.Driver"
    url="jdbc:postgresql://<database-ip>:5433/<database-name>"
    factory="org.apache.tomcat.jdbc.pool.DataSourceFactory"/>
```

This resource is configured to have always 50 active connections to the database, which can be used by the 50 threads always available by tomcat, to serve requests. Since there are some requests that perform more than one database access, the maximum number of idle connections available is 80. This value allows to not being necessary to create new connections whenever the gateway receives request bursts that need to make more than 1 database access. As such, performance is gained in these cases because there's no need to invest time on creating new connections. Regarding the 200 *maxActive* connections, this value coincides with the maximum thread number available to serve requests simultaneously and is set as a prevention value for abnormal workload instants.

removeAbandoned allows the Datasource to automatically return connections to the pool when one of them was abandoned for the *removeAbandonedTimeout* of 60 seconds. This only takes effect when a connection acquired from the pool was not closed and thus to prevent connection leaking. *validationQuery* is the query used to validate connections from the pool before returning them to the caller, it is done each 30 seconds and every time a resource is trying to borrow it (*testOnBorrow*). The *factory* is what defines the use of tomcat-jdbc-pool.

This page was intentionally left in blank

5. Software Development Methodology

For the purpose of this internship, the methodology chosen is agile and based on Scrum, which is a framework for iterative and incremental software development. Even not following Scrum to the letter, the overall concepts of this framework were applied.

5.1. Agile Principles

An agile methodology has a set of principles which distinguishes it from others:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

With this set of principles, Scrum solves a group of risks that often arise in software development. The main risks and their associated mitigation are the following:

- Risk of not pleasing the customer → Mitigated through frequent delivery and feedback
- Risk of poor predictability → Mitigated through constant prioritization and estimation
- Risk of not resolving issues promptly → Mitigated through daily progress meetings
- Risk of not being able to ship → Mitigated through frequent releases

5.2. Scrum Roles

In Scrum there is a set of roles, which represent those who are committed to the project:

- Product Owner

The Product Owner is the person responsible to translate the customer needs in work. He is the one ensuring that his team brings value to the project. He writes a set of users stories that represent what a user does or needs to do, assigns a rank and prioritizes them according to what the client needs more urgently. The team, or the developer (in case there's only one), can also write the user stories, which in this project was exactly what happened. This group of stories is then added to a Product Backlog, which represents a comprehensive list of actions to take in order to implement features.

The Product Owner of my project is Eng. Rui Gil, who also is a Project Manager. This combination of roles makes very sense since he is the one who has the widest vision either regarding the scope of the work and the means to achieve the solution, in other words the technical side.

- Team

The team (in this case myself), is responsible to deliver the project in time and to fulfill the work plan. Most of the tasks inherent to the project development, such as documentation and testing are my responsible and are also an integral part of the product backlog.

- Scrum Master

The Scrum Master is also known as the facilitator whose main responsibility is to help the team to succeed in their objectives and to clear any impediments to the work progress. He can also be considered the coach, since he helps the team to define the work plan and facilitates meetings in order to monitor the project's progress. The Scrum Master of my project is Eng. Filipe Santos who is also my supervisor.

5.3. How it works

The methodology followed used work plans to organize the iterations of the development process. Each work plan has a variable duration depending on the chosen features from the Product Backlog by the Scrum Master and Product Owner, and is based on the Product Backlog previously defined. In the beginning, a set of users stories is chosen to be part of the work plan according to their priority and complexity, making them possible to be performed in the next iteration. They represent the work to be done during this time.

Everyday there's a short meeting with the Scrum Master, which aims to answer the following questions:

- What did you work on since the last meeting?
- What will you work on today?
- What impediments/blocking issues do you have?

In the end of each iteration, the results are presented and their development evolution is analyzed, readjusting the plan if necessary. In order to close a work plan, not only the features composing the work plan need to be concluded but they also must obey to a definition of done previously defined. This definition is the answer to the question, "What is done?" and represents the additional tasks, beyond the code done, that need to be performed in order to state that a task is done. The definition of done agreed with the Scrum Master is the following:

Level	Description
Story	UI Conformity according to guidelines
	Code follows WIT Standards
	Unit & Integration tests pass
	Builds with no errors
	Story implemented
	Code commented
Work Plan	All bugs fixed.
Release	Summary of changes/additions
	Ok from Product Owner
	Code committed on SVN
	Acceptance tests pass

Table 14 - Definition of Done

To conclude, this is shortly explained with Figure 19:

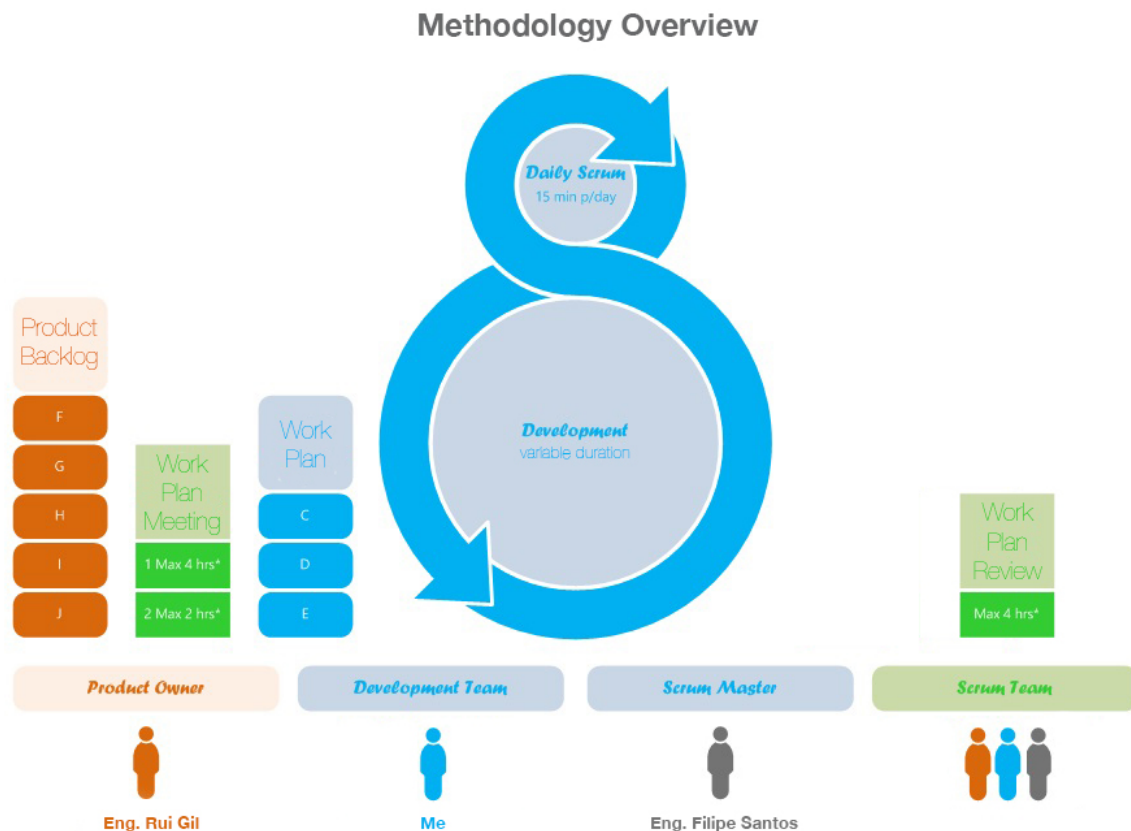


Figure 19 – Scrum Overview [27]

In short, Eng. Rui Gil and Eng. Filipe Santos are responsible to prioritize my backlog according to the features that are most urgent to be implemented. This prioritization results in a smaller set of user stories that make part of the work plan, which is defined when the previous ends. From then, the development cycle starts for the time defined, and everyday Eng. Filipe Santos does a short meeting where he tracks the progress of the work and realizes if there's any issue on the way. Finally when this cycle ends, there's a meeting with the whole team, and a review about what was done. The team also tries to understand what went wrong and what could be changed in the next work plan.

When this period ends, the whole cycle resumes, being defined a new work plan for the next iteration.

5.4. Planning

The project planning is directly interconnected with the methodology followed, a Scrum based approach. In the initial phase, a set of requirements was made in order to understand the project's goals. From these well-established and understood requirements, the next phase was to define the Product Backlog. This Scrum's artifact is a list of ideas for the product, which the team (me) expects to do. It is the single source from which all requirements flow, meaning that all work to be done by the team comes from this artifact.

The product backlog has a set of user stories that define the features from the perspective of the person who desires the new capability. Each one of them aims to capture the ‘who’, ‘what’ and ‘why’ of the requirement in a simple way. They are written with the following structure:

As a developer I want to add support for APNS in order to communicate with Apple Devices

type of user
(who)
some goal
(what)
some reason
(why)

Each user story belongs to a sub-category, which is inside of a main one and has an associated deadline.

Category	Sub-category	User story	Deadline
↓	↓	↓	↓
Implementation	Server-side	<Above>	December

By defining these three components of the user story, the definition remains very simple and easy to understand. The deadlines are set based on the prioritization done by the product owner and also on the difficulty of each user story.

The Product Backlog defines all tasks that need to be made in order to finish the product in the long run. However, since this artifact is defined for the whole project duration, there’s a need to create work plans in the beginning of each iteration, which results in the extraction of some user stories from the Product Backlog. These work plans last for the duration defined by the Product Owner and the Scrum Master, and is based on these tasks that the development is focused.

The user stories that integrate work plans were chosen as the project progressed. As such, when the planned tasks were completed, a new work plan was made. Thus, it is possible to adjust the needs of the company, depending on what represents the highest priority at the moment. Instead of making a detailed planning for the entire project, risking to have to reschedule everything in case an higher priority urges, we chose to mitigate this problem with this approach.

If the reader wants to have a high-level overview of what was made, the Product Backlog serves this purpose and can be found in Annex D.

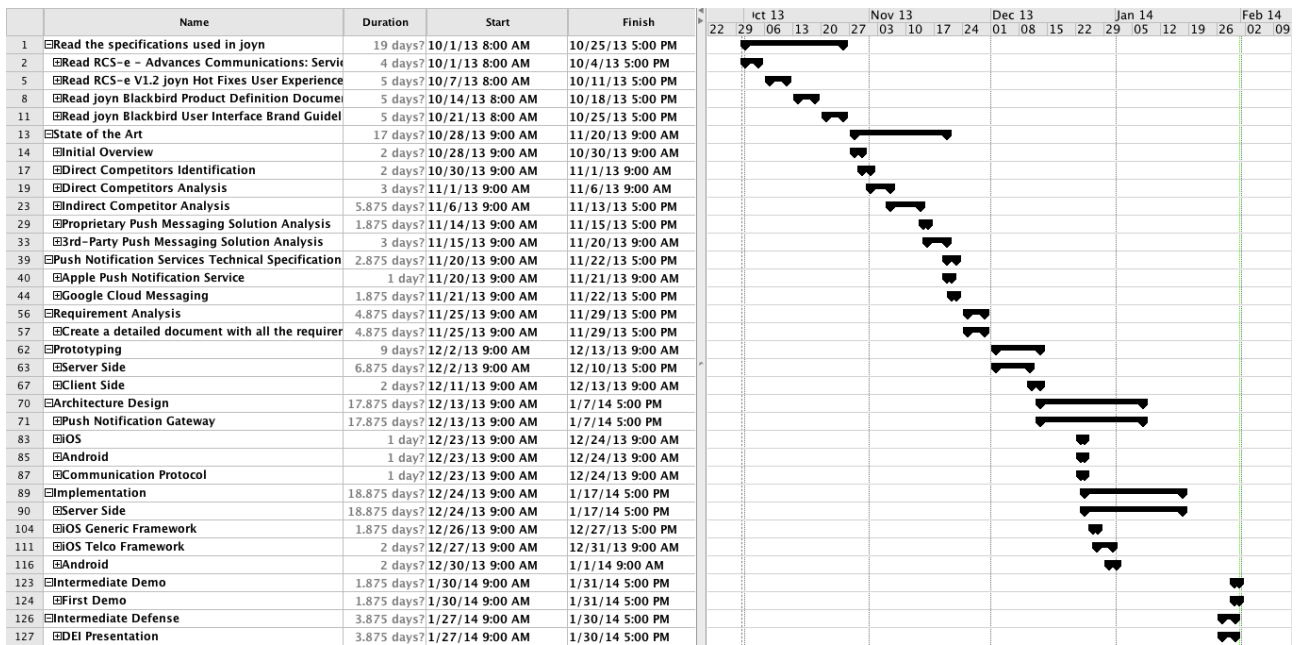


Figure 20 – First Semester Gantt diagram

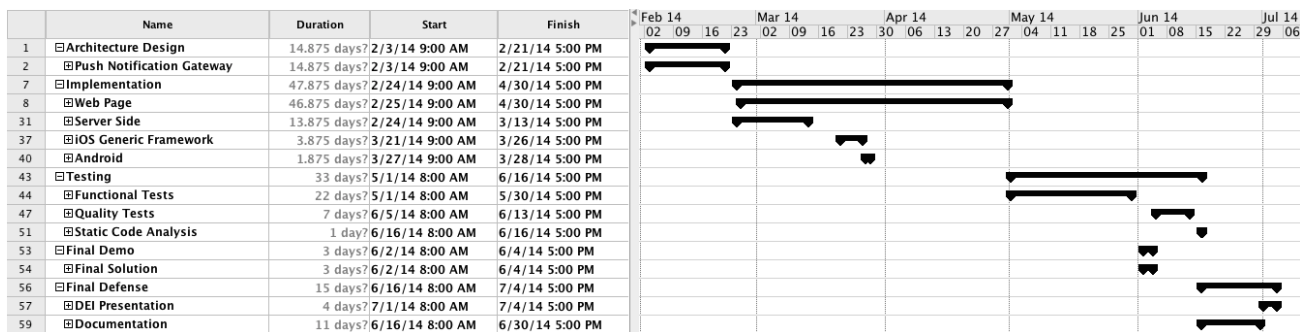


Figure 21 – Second Semester Gantt diagram

5.5. Risk Management

In order to achieve project success, it is important to proactively manage risks throughout the whole development lifecycle. Before becoming a threat, risks should be identified, addressed and their sources eliminated.

In the context of this internship, risks were identified in the beginning of the project and their probability was continually measured along the time. Their impact was classified according to the following values:

Classification	Description
1	Very low impact. Can be very easily overcome
2	Low impact. It would require minor changes in the project
3	Considerable impact. Although with harder work it would be possible to overcome.
4	High impact on the project's success. A great effort would be required in order for the project to not fail
5	If it happens, the project fails

Table 15 - Risks Impact Classification

The following list describes each one of the risks identified in the first semester, and has a final conclusion written in the end of the internship.

R_01 - Design a bad user interface in the web front-end

Impact: 4

A bad UI, or one that does not meet the target users preferences may lead to poor compliance of the final product in accordance to WIT's standards. It can also result in poor acceptance by the target market.

Mitigation Plan:

- Prototyping and design thinking.
- Integrate the design team in order to receive feedback and suggestions.

Final Conclusion:

Although the design team did not make the user interface, it was presented to them in the end of the internship and the feedback received was good.

R_02 - Not being able to use technologies for which I have no knowledge

Impact: 2

Since the project has a wide variety of technologies and libraries that need to be used and most of them are new to my knowledge, it's probable that it will represent a barrier on the project's evolution.

Mitigation Plan:

- Invest time learning and exploring those technologies.
- Seek help from other engineers.

Final Conclusion:

The use of new technologies was not an obstacle since the development occurred smoothly with no problems.

R_03 – Requirements change

Impact: 2

Since the product is aimed for the market and the development is inserted on an enterprise environment, client requirements can change according to the market needs.

Mitigation Plan:

- Frequent deliveries in order to gather feedback from stakeholders.

Final Conclusion

Following the mitigation plan, the project's development didn't suffer any sudden requirements change.

R_04 – Not meeting deadlines

Impact: 3

If the schedules are too ambitious, it could result in lack of time to perform the objectives and in the need to work over the budget. Investing considerable time in planning tasks can lead to lack of time to work on the product.

Mitigation Plan:

- Agree on deadlines with the Product Owner since the beginning.
- Do not be too optimistic on estimated execution times.
- Do not detail more than what is necessary.
- Gather opinion from Scrum Master.

Final Conclusion:

During the development, the deadlines were enough flexible for, whenever necessary, re-adjust the plan. This happened two times because there was improvements that needed to be made to the web UI.

R_05 - Push Notification Services API changes or offline

Impact: 2

Since the project depends on Apple Push Notification Service and Google Cloud Messaging to deliver push notifications, there's a possibility that their API changes, making necessary to change the gateway implementation. It's also possible that the service becomes offline.

Mitigation Plan:

- It's important to be aware of Apple and Google's news regarding their service in order to foresee possible needed changes.

Final Conclusion:

There is no conclusion to take regarding this risk, since it will always be present and there is nothing that can be done to prevent it.

This page was intentionally left in blank

6. Software Quality

Software testing is an indispensable part of the software development cycle. After all, it allows measuring software design quality, and the level of performance according to its design.

Good software development practices are crucial to the quality of software, and special attention to details is also a characteristic that guides developers towards this goal. Throughout the development process, besides the usual code-and-test approach, new tests were created and added to their suite. This is a very useful process since it acknowledges the detection of problems in an early stage, enabling easier and faster rectifications. It is also important to underline that the code comments made the whole debugging process easier.

The software quality assurance process also contemplates research to find the most appropriate tools for testing the software. Throughout this document, all testing procedures taken will be described as well as the conclusions taken from them. It's also important to clarify that the whole testing performed was entirely done by me and not by WIT Software's SWQA Team (Software Quality Assurance Team).

6.1. Functional Tests

Functional tests aim to validate and ensure the software's quality.

6.1.1. Acceptance tests

The acceptance test suite has all the tests that need to be performed in order to guarantee that the functional requirements are met. For this purpose, written tests do not go into great detail since their execution were intended to be done by a software quality team, whose aim is to validate in a simple and easily noticeable way that the features work as intended. The tester only needs to be aware of what the software must do, but not how it does it. In other words, this *modus operandi* can be seen as a black-box testing.

Although the acceptance tests are focused to be executed by a software quality team, as said earlier, they were performed by me. The suite of tests can be found in Appendix E 2.1 and have a total of 68 test cases. Table 16 defines the results obtained.

Status	Total	Percentage
Passed	68	100%
Failed	0	0%

Table 16 – Acceptance tests result

6.1.2. API Tests

Unit tests are a great way to ensure that the code is working correctly. However, since Push Notifications Gateway does not live within a deterministic environment, it is very hard to write precise tests. In other words, since data flow is very dynamic and constantly changing, it is very difficult to know the exact value to expect during an assert operation. This was the reason why was decided that unit tests, whose goal is to test small pieces of code, would not be performed, but instead test the APIs once it is through these that we can access the server to perform the various actions available. The

purpose of testing the system is not focused on details but in the entire platform as a whole. Eventually, as the development occurs if one the API test fails, the problem can be debugged with small unit tests.

For this purpose, a tool named Runscope will be used to perform all tests. Runscope is an automated backend service testing and API monitoring [28] which allows developers to build tests online to verify if data returned is the expected. It is even possible to use returned data in one response to make a new request. This is especially useful to test workflows or use cases where more than one API request is needed, for example:

- POST to `/api/push` a scheduled push and store the `pushId` returned
- GET to `/api/push/scheduled` to check if `pushId` exists

The main goal of the tests performed is to verify that the responses contain the expected status code, headers and JSON values and thus, evaluate values that can be expected. Table 17 resumes the results.

Status	Total	Percentage
Passed	247	100%
Failed	0	0%

Table 17 - API Test results resume

To understand how Runscope works and how the tests were structured, see Appendix E 2.2.

6.1.3. Regression tests

Along the development of the server-side code, as new features were implemented they had to be tested. Also, beyond testing the new code it is also important to check if the new features didn't cause any regression and as such, if the older code still passes in its tests. This was done along the project's development lifecycle, re-running the already existing tests in order to find eventual bugs.

6.2. Quality tests

The application behavior is tested using quality tests. Instead of analyzing functionality, quality tests focus more on evaluating the performance and usability of the application, and thus they are more related to non-functional requirements.

6.2.1. Software Performance

This chapter intends to present a detailed performance analysis of some server components. The platform is hosted in a virtual machine inside WIT Software and all the tests were run on it. The testing environment is as follows.

Server (Virtual Machine):

Ubuntu 12.04.02 LTS (Precise Pangolin)
2 CPU Core Intel Xeon X3363 @ 2.83GHz
4GB RAM

Client:

OSX 10.9 (Mavericks)
2 CPU Core Intel Core i5 @ 2.5GHz
10GB RAM

All the following tests were performed with a cable connection and inside WIT Software's Intranet.

6.2.1.1. Web Application

This chapter relates to tests made to the web application.

6.2.1.1.1. YSlow Rules

YSlow is a webpage performance analyzer that scores from "A" to "F" a set of 23 rules designed for high performance websites [29]. These rules will be used to verify the performance of two pages: index and dashboard.

#	Rule	Index	Dashboard
1	Make fewer HTTP Requests	A	A
2	Use a Content Delivery Network	E	F
3	Avoid empty src or href	A	A
4	Add Expires headers	A	A
5	Compress components with gzip	C	F
6	Put CSS at top	A	A
7	Put JavaScript at bottom	A	A
8	Avoid CSS expressions	A	A
9	Make JavaScript and CSS external	N/A	N/A
10	Reduce DNS lookups	A	A
11	Minify JavaScript and CSS	A	A

#	Rule	Index	Dashboard
12	Avoid URL redirects	A	A
13	Remove duplicate JavaScript and CSS	A	A
14	Configure entity tags (ETags)	A	A
15	Make Ajax cacheable	A	A
16	Use GET for AJAX requests	A	A
17	Reduce the number of DOM elements	A	A
18	Avoid HTTP 404 (Not Found) Error	A	A
19	Reduce cookie size	A	A
20	Use cookie-free domains	B	B
21	Avoid AlphaImageLoad filter	A	A
22	Do not scale images in HTML	A	A
23	Make favicon small and cacheable	A	A
Overall grade		A (96/100)	A (92/100)

Table 18 – YSlow rules evaluation

There are two rules that deserve special attention once they have a bad score, they are rule number 2 and 5. Regarding the Content Delivery Network, it is out of scope to improve this since it involves additional costs for deploying on various machines across different geographical areas. The compression of components with Gzip is in fact a good optimization that could be done. However, because of vulnerabilities such as CRIME and BREACH, which could be explored in HTTPS connections, this rule was not taken into account.

Regarding the overall score of the two pages measured, it can be said that the grade is very optimistic and represents a good optimization effort done (see Figure 22 and Figure 23).

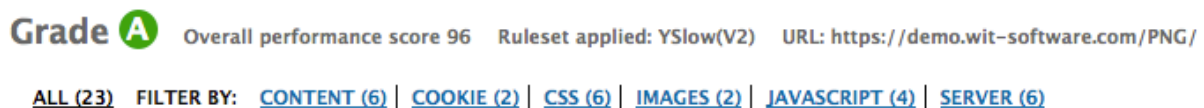


Figure 22 – YSlow Index page score

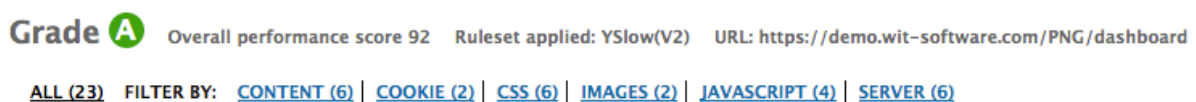


Figure 23 – YSlow dashboard page score

6.2.1.1.2. HTTP Requests

This chapter will present the analysis made to the HTTP Requests in a cached and not cached environment for each page.

Statistics The page has a total of 7 HTTP requests and a total weight of 1047.2K bytes with empty cache

WEIGHT GRAPHS

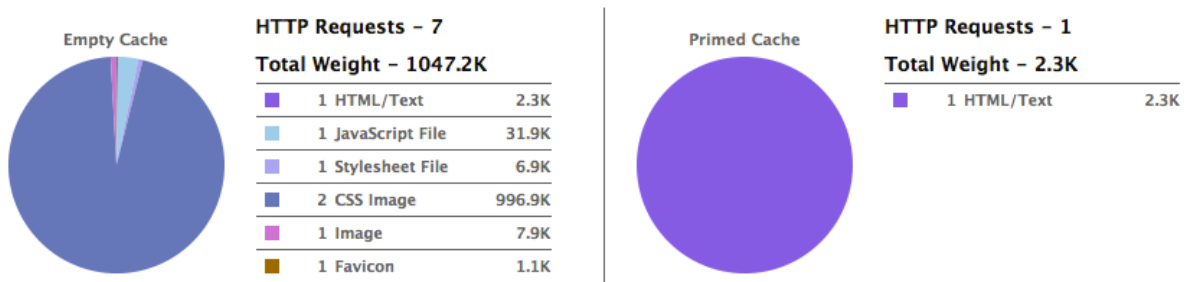


Figure 24 – Index page empty vs primed cache

Statistics The page has a total of 12 HTTP requests and a total weight of 529.6K bytes with empty cache

WEIGHT GRAPHS

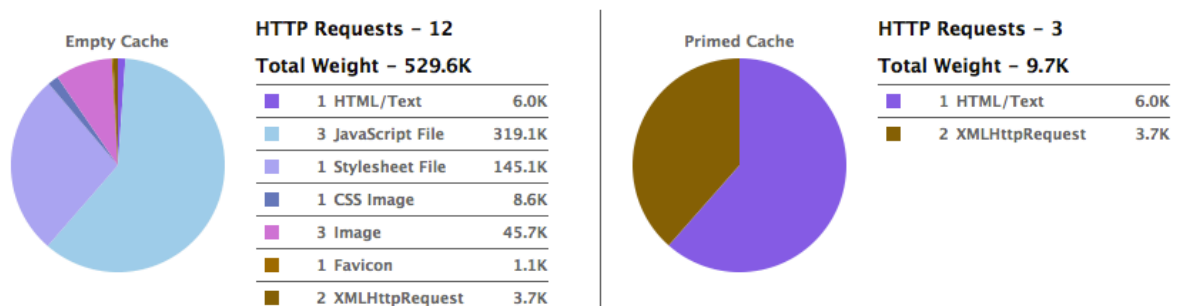


Figure 25 – Dashboard page empty vs primed cache

In the index page we can see that by caching all the components, except the HTML page, there is a gain of 455% on the number of bytes that need to be transferred between the server and the client. Although in the dashboard page the improvement is not so big, the amount of bytes transferred is still 54 times inferior. This optimization is achieved because the client doesn't need to perform so many HTTP requests to the server.

The *XMLHttpRequest* request types are Ajax calls made to the server to retrieve dynamic data asynchronously. This fetched data is constantly changing and thus, there is no added value on caching it.

6.2.1.1.3. Page Load Times

Attempt	Load times (sec)											
	Index page					Average (sec)	Dashboard page					Average (sec)
	#1	#2	#3	#4	#5		#1	#2	#3	#4	#5	
Empty cache	0.44	0.19	0.39	0.40	0.50	0.384	0.34	0.34	0.35	0.92	0.36	0.462
Primed Cache	0.28	0.13	0.26	0.33	0.40	0.28	0.27	0.27	0.29	0.86	0.30	0.398

Table 19 – Pages empty vs primed cache load times

Since without cache the average load times are already very fast, there's no huge improvement when it comes to cached results. Even so, in the index page the page loads 37% faster with cache and in the dashboard 14%. In order to improve these values, the Gzip compression could be turned on (we would

need to see if the extra CPU cycles to compress these files would be advantageous) and the caching for the Ajax calls could be made. However, as justified above, there's a reason for not doing that.

6.2.1.1.4. Benchmarking

For this purpose, accesses to the server's index page were benchmarked. The goal was to test how many requests per second the tomcat installation is capable of serving with the current configuration. To perform this kind of test, the Apache Benchmarking tool (ab) was used with the following principles:

- The total number of requests for all tests is 10.000.
- The number of multiple requests that are performed at a time are increased on each test (concurrency).
- Each request is performed within a separate connection, simulating different users.
- For each number of multiple requests the test was made three times and in the end the average was calculated.

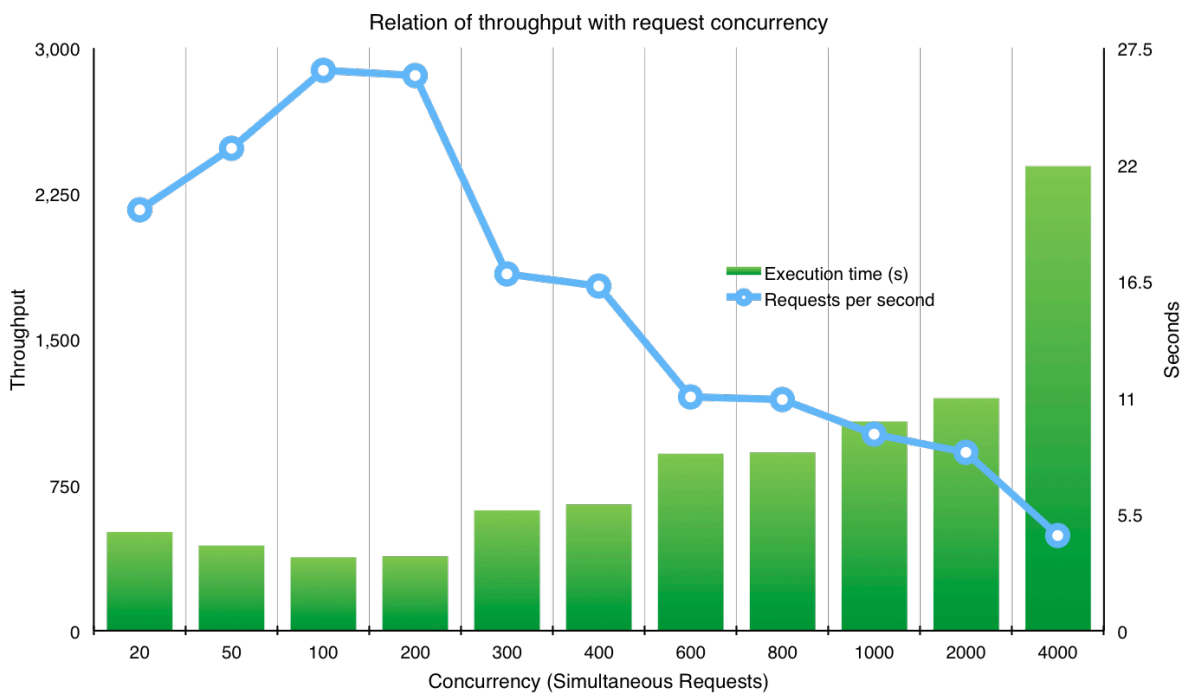


Chart 1 – Index website page relation of throughput with request concurrency

Chart 1 represents the number of requests that the server can process per second based on the number of concurrent (simultaneously) requests made. In the first three concurrency scenarios the server is able to return the Index page ever faster. This increasing performance happens because the clients take better advantage of the tomcat thread pool, which gets expanded until a maximum of 200 threads.

Although between the third and fourth tests (100 and 200 chunks of simultaneous requests) tomcat instantiates 100 new threads, being used at its full strength (1 thread for each request), the results do not improve, keeping the same throughput. The reason for this to happen is because the server starts to have a hard time dealing with this many threads concurrently.

For all the remaining concurrency tests, the throughput becomes increasingly smaller since there aren't enough threads to handle all the requests and as such, they have to sit idle for some time without being handled until another request threads is freed up.

Regarding the response time, it presents, as expected, a reverse trend in relation to throughput. We can observe that in the best-case scenario, the 10.000 requests to retrieve the index page are all processed in 3,46 seconds, which is an astonishing value.

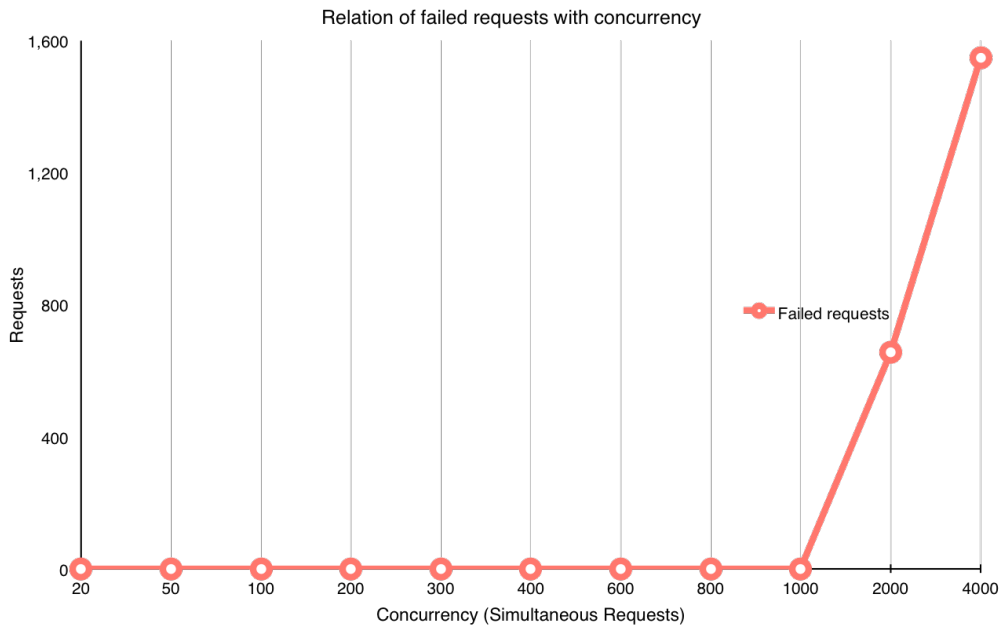


Chart 2 – Index website page relation of failed requests with concurrency

In this benchmark, it is also important to know when tomcat starts to fail while processing requests. When the server receives a big amount of simultaneous requests, tomcat is not capable of keeping up with the demand. This handicap is verified when request bursts are higher than 1000.

6.2.1.2. RabbitMQ Benchmarking

It is important to understand the throughput of this message broker, and for that purpose, a set of tests was designed. Common principles for all the following tests:

- Each test was performed four times.
- Push Jobs is the type of data used to travel inside queues.

6.2.1.2.1. Publishing

This test checks the maximum throughput possible when publishing messages to the queue. For this purpose, the consumers were shutdown and the producer looped 500.000 times until all Push Jobs were in the queue.

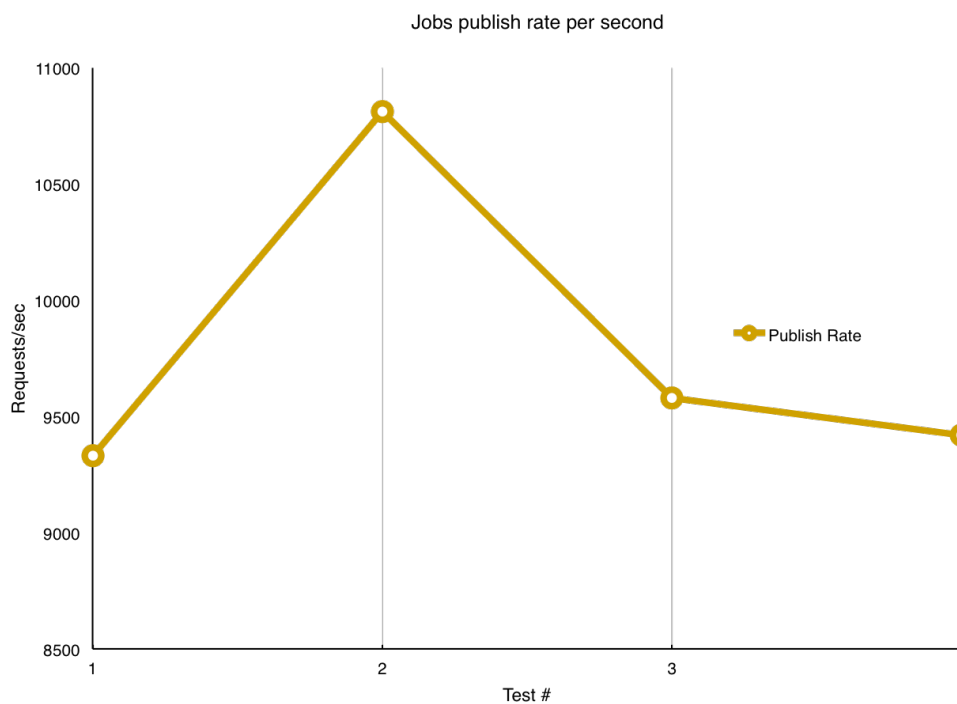


Chart 3 – RabbitMQ Job publishing throughput

It is possible to achieve publishing rates of approximately 10800 jobs per second. Although this number is beyond the necessary, it is greatly punished because the queue is set to be durable and thus, save every message to disk. This throughput would be much better if this setting was not in use because RabbitMQ wouldn't have to deal with disk IO operations. However, in order to meet reliability requirements, this setting had to be used.

6.2.1.2.2. Consuming (delivering)

In the same way we tested the performance of publishing in the queue, we also performed tests on consuming those messages. For this purpose, the queue was loaded with 500.000 Push Jobs and after that they were all consumed sequentially. Different number of consumers was used in order to compare the performance possible to achieve.

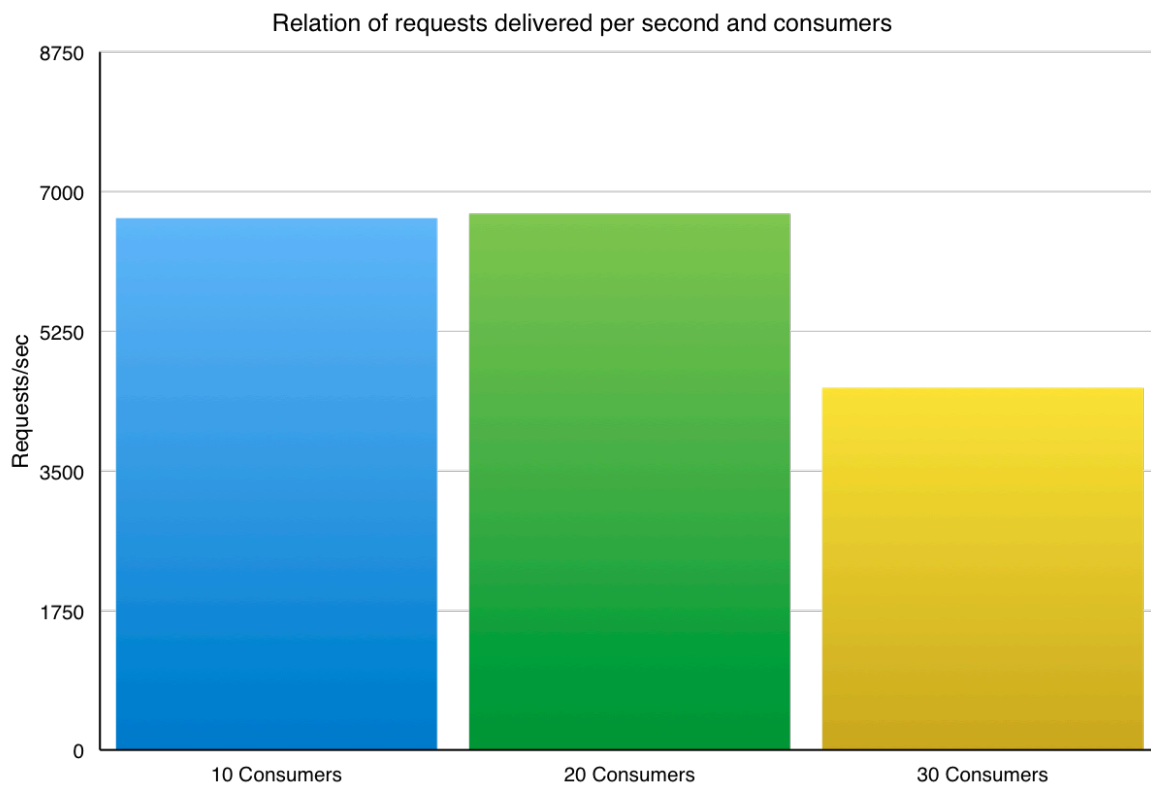


Chart 4 - RabbitMQ Job delivery throughput according to number of consumers

As decided in the architecture, the number of consumers currently in use for the gateway is 20.

Since the number of jobs delivered per second (between 10 consumers and 20) is almost the same, it is preferable to use the latter. This is because when long-running jobs are executing and, blocking the thread (and consumer) for a few milliseconds, there will be more of it available to keep-up with the server's requests.

When using 30 consumer threads, performance gets damaged because RabbitMQ needs to keep track of them all, and their additional overhead ceases to be compensatory.

6.2.1.2.3. Publishing and delivering simultaneously

When publishing and consuming are done simultaneously, the performance figures slow down a bit because RabbitMQ needs to handle and synchronize two types of operations (writing and reading) at the same time.

Since the optimal consumer number is already achieved, it is now important to understand what is the best value for the *Prefetch Count* (QoS), which is the number of messages that can be delivered to a consumer before being acknowledged.

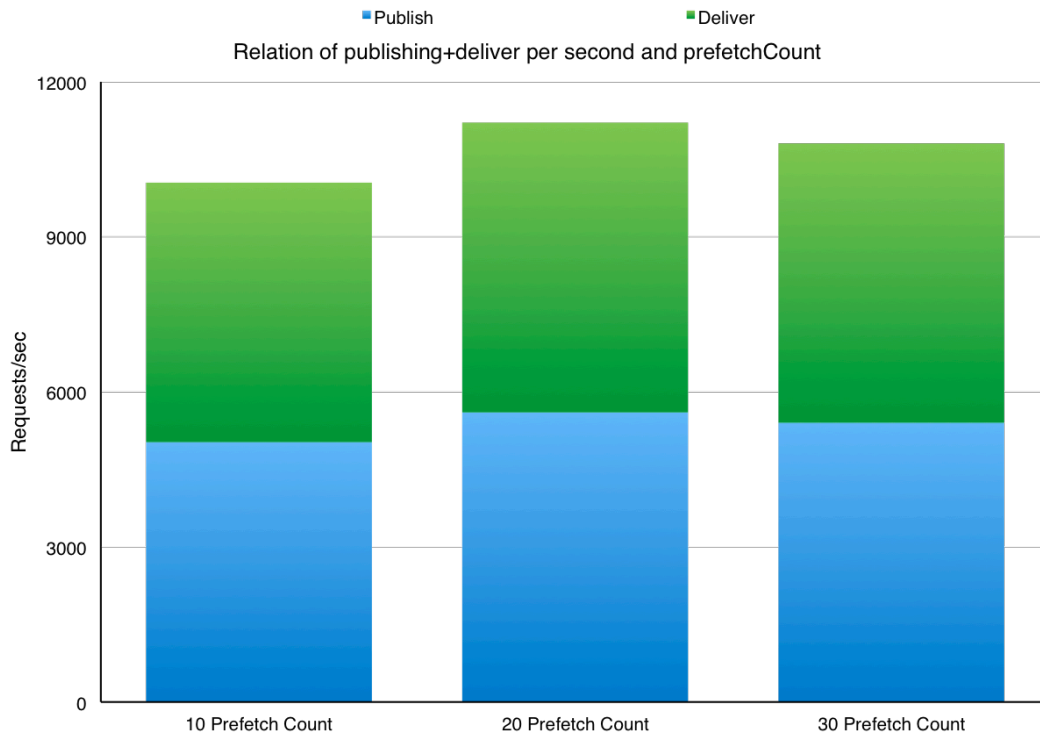


Chart 5 - RabbitMQ Job publishing + delivering throughput according to QoS Value

The prefetch count (or QoS) is a value that needs to be increased until there is no more performance gained, becoming stagnant. The goal is to achieve a number where the consumers are always busy with work to do and thus, never starves.

With a QoS of 10 we were able to get a throughput of around ~5000 requests/second, although it's possible to have more. With 20 we saw a boost of around 600, which is great and sets a constant consumer utilization of 100%.

Above a prefetch count of 20 the throughput figures were around the same values as can be seen with the higher 30 QoS. The difference is that, with higher numbers the consumers are abruptly targeted with many messages instantaneously, and when the prefetch count is reached, RabbitMQ needs to wait for those messages to be acknowledged before sending new ones to that consumer. This makes the delivery of messages not constant but instead with successive oscillations. RabbitMQ will then need to switch the TCP Connection state to flow, blocking it temporarily because it is publishing to the queue too quickly (Figure 26).

Once this already happened with the prefetch count set to 30, the best value that guarantees constant flow of messages is 20.

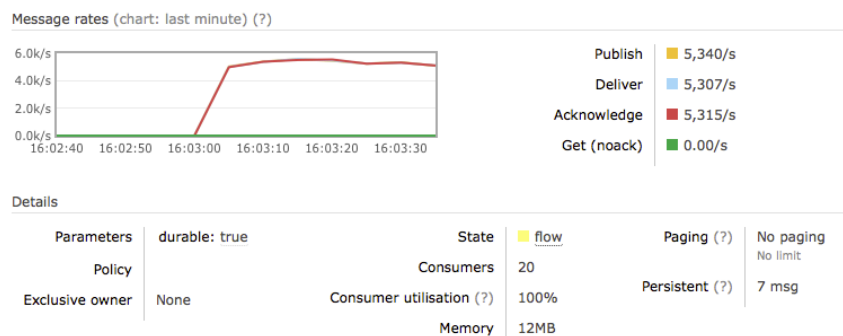


Figure 26 – RabbitMQ' Flow Control with prefetch count of 30

6.2.1.3. Push Notifications Benchmarking

After analyzing RabbitMQ, the major server component, we came to realize that the throughput offered by the broker did not represent a bottleneck to satisfy the unique performance-related non-functional requirement, the push notifications throughput.

When it comes to process a request there are several steps that must be done. The whole execution cycle time involves an initial validation of the request, its conversion to a Job and consequential publish to the queue, and finally its execution by a consumer.

The next two chapters present some tests that were made to understand the maximum number of iOS and Android devices to which the gateway is able to push notifications to. For each platform the test followed the next purposes:

- Each push request targets a different number of devices.
- Each push request was made 4 times
- The duration of each request is a sum of four different times: pre-queue validation, time in queue, time in consumer and also in the respective push engine.
- When benchmarking the throughput for Apple, the notification was sent to two different iOS devices. The same happened for the Google test, for which the notification is sent several times to two different devices.

The requests made to the server has the following base format where the unique difference between them is the number of repeated *device_ids*:

```
{
  "to": {
    "device_id": [
      "+351911222333",
      "+351919999999",
      .....
      .....
    ]
  },
  "notification": {"alert": "Hi. Testing hard."}
}
```

6.2.1.3.1. Apple Engine

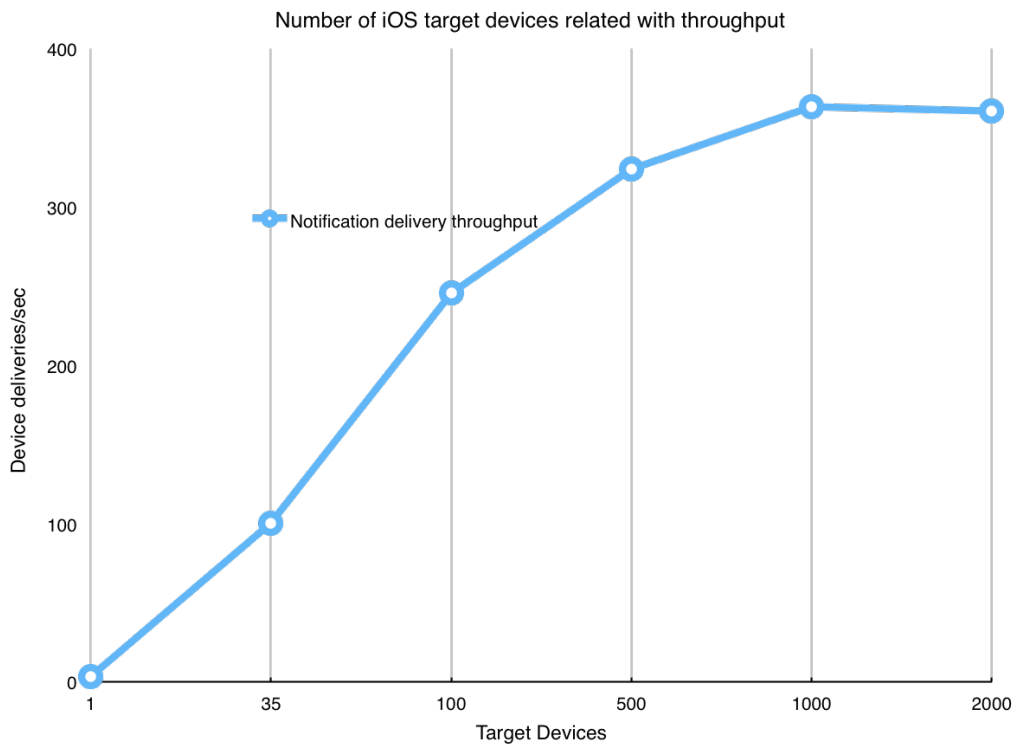


Chart 6 – Throughput of push requests to iOS devices

On a push request that targets iOS devices, it was possible to obtain results far above the initially set goal, which aimed to an average of 35 devices reached per second.

The greater the number of devices to reach with a push request, the better the throughput. As described before, Apple Push Engine creates more threads and connections when more devices need to be targeted. This improves parallelism and the engine's capacity to deliver more notifications. However, the gateway's performance reaches its limit at around 360 devices per second.

Even with the expected 35 notifications per second, the Apple's engine is able with a single thread and connection to deliver them in 0.35 seconds.

6.2.1.3.2. Google Engine

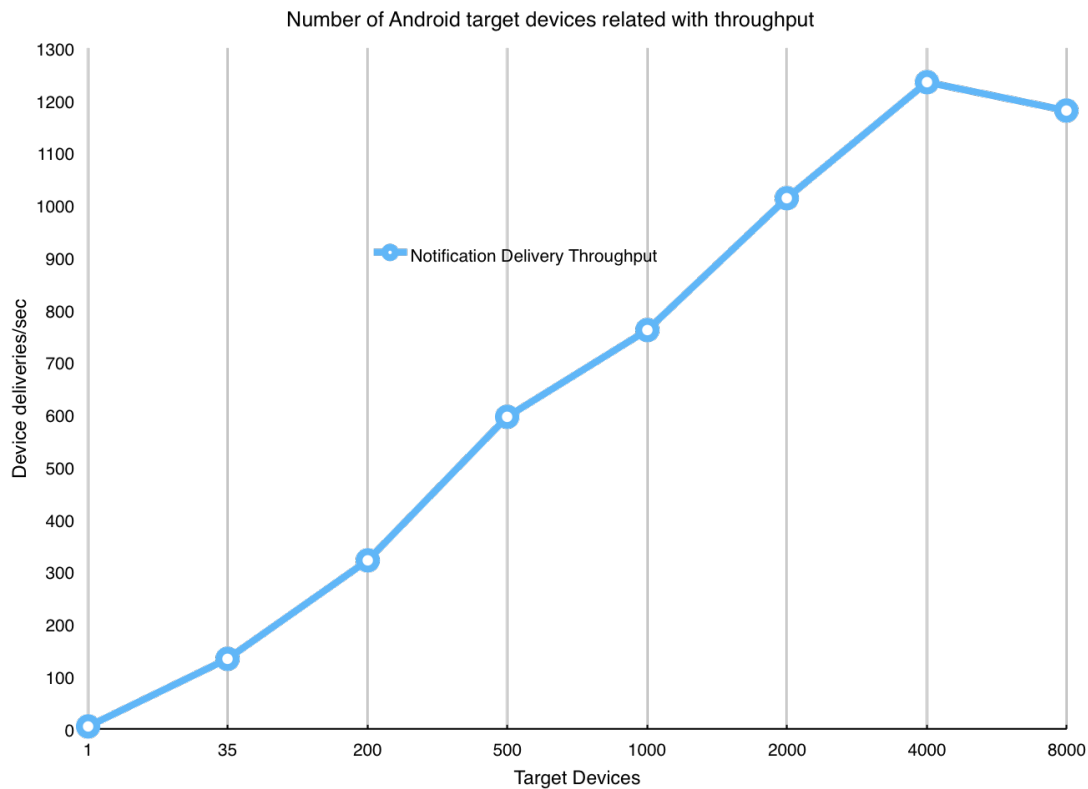


Chart 7 - Throughput of push requests to Android devices

We can observe that, once again, with the 35 notifications per second goal, Google's engine is able to take care of that request in 0.26 seconds, which is 4 times faster than the target value. Since it is already proved that the non-functional requirement is achieved with distinction on either platform, we switched focus to the gateway's maximum capability to deliver notification to android devices.

On Google's Engine, a thread is created for every multiple of 1000 devices and each of them receives the same amount of devices to which it needs to dispatch the notification. As such, like Apple's Engine, it is expected that the greater the amount of threads created, the bigger the parallelism achieved, and thus more notifications send per second. This is indeed what happened until a limit was reached, which is around 1200 notifications per second, when the total number of target devices are around 4000. Reached this value, the overhead related to the every higher thread number ceased to deliver better results. The number of accesses to the database to translate the *deviceId* to a *registrationId* also contributed to the stagnation of the throughput.

The performance values achieved when communicating with Google Cloud Messaging are far greater than with Apple's Push Notification Service. Throughput is ~3.4 times faster which is greatly due to the simplicity of this service in comparison to Apple's. A notification is sent up to 1000 devices with a unique HTTP Post instead of having to write each device notification to a socket.

6.2.2. Usability tests

Usability testing is a technique focused on the evaluation of a product. These tests are done by real users on a controlled environment, and represent very useful information once they give feedback on how the community will use the designed system.

The component subject to usability testing is the web application, and for this purpose a set of tasks were written and performed by a group of WIT Software's employees that never had previous contact with the webpage and knowledge about the project's scope. Once the web platform's target audience is software engineers and people in the marketing business, and since WIT has this job positions, these were the subjects' profile.

Five people belonging to these job positions were chosen, where two of them are women who work in marketing and the other three are software engineers. The choice of WIT's employees to be the test's subjects was also due to the fact that the application is confidential. The guru of web page usability, Jakob Nielsen, explains the reason for the five individuals. He states, "a usability test with 5 users will typically uncover 80% of the site-level usability problems" [30]. Eighty percent is sufficient given the limitations of confidentiality. In case it will be necessary to make a deeper study, focus groups shall have to be created in order to have a wider range of subjects.

The metrics used to evaluate the usability of the web application were:

- Task result: There are three different possible results. Task performed in the first attempt (2 points), able to execute but with a few hesitation (1 point), doesn't know how to do it (0 points).
- Number of actions: During the test execution, the number of actions taken by the user in order to perform one task was monitored. The goal is to know if he was able to complete the task with the minimum number of actions required.

The tasks of the usability script are defined in Appendix E Table 7 and the results in Table 8. The reader must consult these two tables in order to understand the following conclusions regarding the tasks that caused more confusion.

Task 2

Description: Switch application in use to "Push It Development".

Observation: When users finished logging in, they don't seem to locate the upper right button to switch application. Two users were not able to switch application, and the other three achieved the result by clicking on the "Push Notifications" shortcut inside the application box and then returning to the dashboard.

Possible Solution: All subjects gave the same feedback by saying that each application's box should have an icon to select them.

Task 3

Description: Edit the account's email.

Observation: Although all users were able to edit the account's email, in general they all had trouble on doing so. When the subjects were asked to do this task, their general immediate reaction was to click on the "Settings" button on the left panel, which is not related to the account but instead to the application selected. After realizing that these settings were not related to the account, they came to the

conclusion that they were in the wrong place and only then they were able to locate the upper right button with the account name and access its configurations.

Possible Solution: Since there seemed to be a general confusion with the left panel, it is important to underline that all of its actions are related to the application being used. A possible solution would be to change the “Settings” name to “Application Settings”.

Task 8

Description: Create a pre-defined push notification, also known as a campaign. Define two different texts for iOS and Android and give a name to that campaign.

Observation: I realized that the subjects after being specifically asked to define a text for each platform, they tended to only select iOS, which is the left-side button. Because of the platform button's format, which can be compared in the real world to a light switch, the users thought that only one of the two platforms (iOS and Android) could be selected, and not both.

Possible Solution: Since the problem lies on the representation of the buttons of the platform, the UI should be rethought in order to offer the user a more intuitive way for him to come to the conclusion that there are three possible choices: select iOS, Android or both.

Task 9

Description: Use the campaign just created and send it to users using the Android platform.

Observation: Only one subject was able to successfully do the task, two had a hard time to execute it and the other two gave up. Users didn't understand that the green button with a pen inside makes possible to use that campaign, some even thought that the icon described an edit action and thus didn't even try to click it. On the other hand, there were testers who kept going to the Quick Composer Push Notifications page to find a way to pick the campaign to use.

Possible Solution: As suggested by the subjects, there are two possible ways to solve this usability issue, which at least for them would be a good fit. Changing the icon inside the green button to something more intuitive or, in the Quick Composer insert an option that makes possible to select a campaign to use, similar to the audience's button.

Task 17

Description: Display the average number of application opens per month

Observation: Four users didn't find a way to display the average number of application opens per month and the other one after some investigation within the graph's options, gave up and found this information available below. After some time analyzing the results of this task, I came to the conclusion that the subject's mind was focused on manipulating graph options because the 5 previous tasks were related to that. When the testers were told to perform this operation, their immediate action was to select a time frame of the last 30 days and then find a way to calculate the average. Well, although this task is designed to be completed on the app opens page, it has nothing to do with graph manipulation. The average number is already calculated and displayed to the user below the graph.

It was considered that this task induces the user in error due to its malpositioning in the script. As such, the script was rewritten and this task was repositioned between 11 and 12. In order to verify if this was really a problem in the script, five new testers were chosen, this time only male software engineers. The tests were performed again until this task (now on position 12) and no other task results were saved.

The results were positive, since the testers did not come from a sequence of tests related to parameters manipulation of graphs, their behavior was not biased and as such the results were quite different as can be seen on Table 20.

Task #	Subject 1		Subject 2		Subject 3		Subject 4		Subject 5	
	Result	Actions	Result	Actions	Result	Actions	Result	Actions	Result	Actions
12	2	2	1	3	2	2	2	2	2	2

Table 20 – Usability test reformulated task result

This usability test was written and performed in the end of the application's development. The overall results were good and allowed to understand the website's usability and which parts have to be improved in future work.

6.3. Static program analysis

Static program analysis consists on analyzing the source code without actually executing it.

6.3.1. Javadoc

All code written in Java has Javadoc compliant comments. This includes the server-side, the android SDK and the Java API SDK. Not related to Javadoc but regarding comments, the iOS frameworks are commented with common conventions, not following any specification.

6.3.2. CodePro Analytix

CodePro Analytix is a Java software testing tool created by Google for Eclipse [31].

6.3.2.1. Code Audit

The first feature used is Code Audit which aims to give useful information about common mistakes found in the code. For the server-side code the following report was generated.

Name	Occurrences	Description	Fixed
Command execution	1	Do not use <code>Runtime.exec()</code> commands because the command is platform dependent.	No
Empty catch clause	37	Catch clauses should not be empty.	Yes
Log exceptions	76	Exceptions that are caught should be logged.	No
Disallow Unnamed Thread Usage	9	Threads should be named to aid in debugging.	Yes

Table 21 – Code Audit Results

The first rule mentioned is indeed true but since we are deploying the application in a controlled environment, there's a guarantee that the command executed works and so, this rule was ignored.

The empty catch clauses found were fixed by logging the exceptions thrown. Regarding the exceptions logged, CodePro doesn't consider `debug()` or `error()` to be logging methods. Only the `logger()` method, which is not used frequently, accounts for the auditing rules. Regarding the last rule, there were a couple of threads being created without a name and as such, they were given one.

Neither the Android SDK nor the Java API SDK reported any rules being violated.

6.3.2.2. Code Metrics

This feature generates a report about a set of metrics. This chapter will be divided in the three projects written in Java.

6.3.2.2.1. Server

The following data is related to Java files only and is a subset of the metrics presented by CodePro.

Metric	Value
Average Lines Of Code Per Method	15.53
Lines of Code	17,976
Number of Characters	934,532
Number of Classes	146
Number of Comments	996
Number of Constructors	173
Number of Methods	825
Number of Packages	29

Table 22 – Server code metrics

With the values from Table 22, the project's dimension can be easily perceptible. The server code is organized in 29 different packages where 146 classes reside. These classes have, in turn, a total of 825 methods and in the end everything sums a total of 17976 lines of code.

As a matter of curiosity, the 3 packages with more lines of code are: the one holding all the database query accesses, the package which has all the struts' actions and finally the one with all the API endpoints. They have respectively 5394, 3419 and 1703 lines of code.

6.3.2.2.2. Android SDK

Table 23 presents only a subset of the data computed by CodePro.

Metric	Value
Average Lines Of Code Per Method	12.77
Lines of Code	915
Number of Characters	52,192
Number of Classes	12
Number of Comments	52
Number of Constructors	9
Number of Methods	52
Number of Packages	2

Table 23 – Android SDK code metrics

The Android SDK is by far simpler than the server-side code. It is composed of only 2 packages, where one of them has all the logic and the other has the interfaces, which the developer should implement if he wants to receive the request's responses.

6.3.2.2.3. Java API SDK

Table 24 presents only a subset of the data computer by CodePro.

Metric	Value
Average Lines Of Code Per Method	13.82
Lines of Code	1,243

Metric	Value
Number of Characters	53,946
Number of Classes	20
Number of Comments	61
Number of Constructors	24
Number of Methods	54
Number of Packages	3

Table 24 – Java API SDK code metrics

This SDK has 3 packages, which contains the logic, custom exceptions and interfaces. The two classes with more code are *PushIt*, which has all the request-specific methods that receive the parameters to compose the JSON body and *WebServiceTask*, which is responsible to perform the HTTP requests to the gateway.

6.3.2.3. Code Dependency Analysis

With this feature is possible to observe a project’s dependencies. In other words, the libraries (.jar) from which it depends.

This chapter will be divided in the three projects written in Java.

6.3.2.3.1. Server

Figure 27 represents the server’s dependencies.

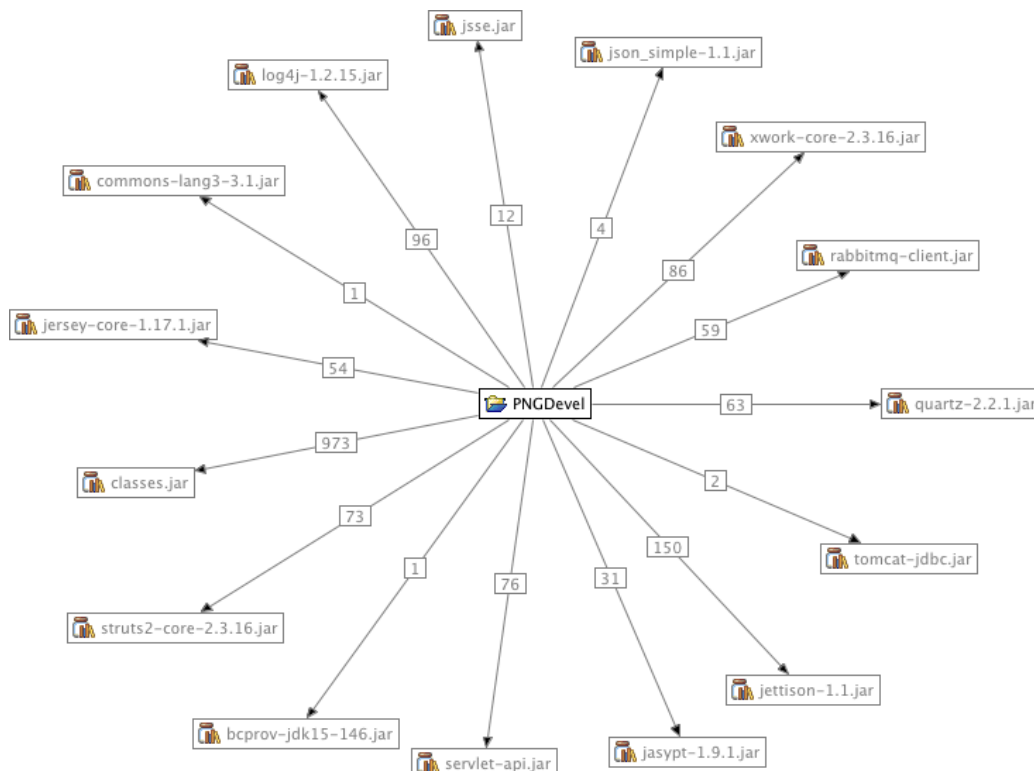


Figure 27 – Server dependency analysis

- jsse.jar – Java Secure Socket Extension library.
- log4j-1.2.15.jar – The Java logging library.
- commons-lang3-3.1.jar – Helper utilities for the *java.lang* API. Used for *APP_ID* random generation.
- jersey-core-1.17.1.jar – Has all the core module jars related to the RESTful Web Services Jersey framework.
- classes.jar – Java-related functions.
- struts2-core-2.3.16.jar – Struts 2 Web Application Framework library.
- bcprov-jdk15-146.jar – Bouncy Castle Library. Cryptography-related features.
- servlet-api.jar – Apache tomcat specific library. Contains all *javax.servlet* classes, methods and interfaces.
- jasypt-1.9.1.jar – Encryption library.
- jettison-1.1.jar – Collection of Java APIs for JSON manipulation.
- tomcat-jdbc.jar – Provides JDBC connection pool features.
- quartz-2.2.1.jar – The Job Scheduling Library
- rabbitmq-client.jar – Library to connect to the RabbitMQ broker.
- xwork-core-2.3.16.jar – Features used by Struts 2 for actions and interceptors implementation.
- json_simple-1.1.jar – Java Toolkit for JSON.

6.3.2.3.2. Android SDK

Figure 28 represents the Android SDK dependencies.

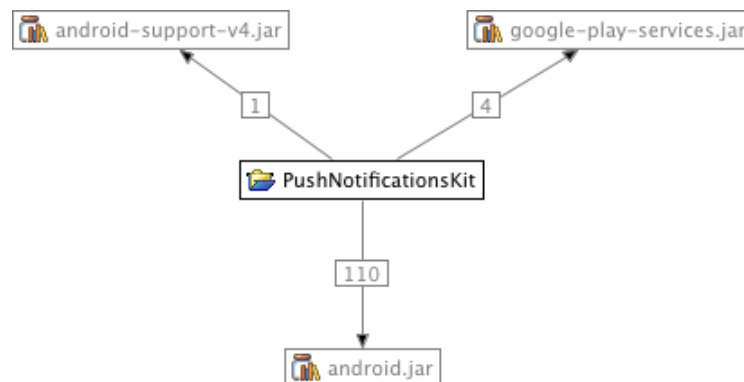


Figure 28 – Android SDK dependency analysis

- android.jar – The android SDK code library
- android-support-v4.jar – Provides support for older Android versions.
- google-play-services.jar – Allows to access Google services. Used for Google Cloud Messaging integration.

6.3.2.3.3. Java API SDK

Figure 29 represents the Android SDK dependencies.

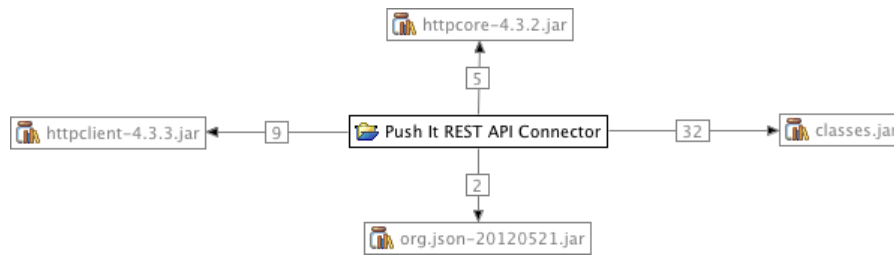


Figure 29 – Java API SDK dependency analysis

- httpcore-4.3.2.jar – Set of low level HTTP transport components.
- httpclient-4.3.3.jar – Implements the client-side of the most recent HTTP standards.
- org.json-20120521.jar – A JSON manipulation library.
- classes.jar - Java-related functions.

6.3.3. Xcode Statistician

This program generates a report with simple statistics regarding the Xcode project selected. The iOS generic framework can be statistically resumed to the values presented in Table 25.

Metric	Value
Lines of Code (excluding whitespace)	1,137
Number of Characters (excluding whitespace)	34,999
Number of .m files	8
Number of .h files	8
Number of Classes	8

Table 25 – iOS generic framework statistics by Xcode Statistician

This framework has three main groups of files: logic, delegates and utils. The first one has all the request-specific methods, which receives the parameters that will feature the request’s JSON body.

The second group of files has a delegate class per request, which implement the *NSURLConnection* and *NSURLConnectionData* delegate methods, in order to receive the request’s responses. It is inside these delegate methods that the response is, in turn, send to the application’s delegate methods.

In the utils group exist two classes, the *Reachability.m*, which is the Apple’s official code to check for Internet connectivity, and the *UIDeviceHardware.m* that is responsible to return a formatted string representing the device’s model.

This page was intentionally left in blank

7. Conclusion

7.1. Work done

It is fair to say that the project development occurred was successful and that the goals were achieved.

The work done along the internship is represented in the following list:

- State of the art

In order to have an overview of where my internship fits, and what are the similar solutions already in the market, a research was done and is reflected on the state of the art. The approach sought to know the existing push messaging solutions in the market in order to understand and analyze how they operate and which features they integrate.

- Competitor Analysis Document

Since push notifications gateway is a platform that could be integrated with the RCS applications developed by WIT Software, delivering push notifications to them, it was important to know the competitors in this field. This information can be found in Appendix F.

- Product Backlog

The product backlog containing all the user stories needed to be implemented in order to successfully fulfill the project objectives were written in a document which is provided in Appendix D.

- Push Notification Services Technical Specification

Since push notification providers like Apple and Google are an integral part of my internship, it was important to thoroughly understand how they work. As a part of this learning process, a document with all the details of these services was made and is provided in Appendix A.

- Requirements document

A requirement analysis was made and had successive amendments and additions as I continuously received feedback from the stakeholders of the project. This document is provided in Appendix B.

- Architecture document

The final solution, which was subject to slight modifications as the project progressed, is thoroughly detailed and explained in Appendix C.

- Software Quality document

In order to thoroughly test all the components of the final application, a test document was made describing all the test cases that were be subject to validation. This document can be found in Appendix E.

- Paper for APCER NP 4457

This project, Push Notifications Gateway, was chosen to be part of a document made by WIT Software for APCER to renovate the certification NP 4457 (Research, Development and Innovation Management System). It was structured in four chapters: state of the art, goals, work done (related to the architecture) and results. The certification was renewed, being my project a contribution towards this end.

- iOS Generic Framework

The framework aims to simplify the integration of client applications with the push notifications gateway.

- iOS Telco Framework

The framework was made according to a specification made by WIT Software for a European Network Operator, which wanted to be able to send Push Notifications to iOS Devices. This API was made specifically for network operators to use with their apps, and can be considered a lighter version (a subset) of the generic.

- Android Library

This library allows the same features of the iOS Generic Framework however it is directed to Android applications.

- Android prototype

A simple android application integrated with the library developed was made in order to assure that all features were working as expected.

- iOS prototype

A simple iOS application integrated with the generic framework developed was made in order to assure that all features were working as expected.

- iOS Telco Framework Integration

The iOS Telco Framework was integrated with an RCS application in order to test its features.

- Java REST API SDK

To ease the communication with the 3rd party-servers API, a Java SDK was made. It facilitates the payload construction and the HTTP communication.

- Server

The server, deployed on a tomcat instance, includes several different components developed such as: REST API, Job-oriented architecture, RabbitMQ integration, PostgreSQL database, Quartz scheduler integration and the Push Engine which has the Apple and Google engine.

- Web Application

The web application was made in order to expose the features available in a more intuitive and user-friendly way. Its usage is intended to be done by less-experience users who don't want to use the API to integrate the features with their application servers.

- Tests

As described in Appendix E, the application was tested in a variety of ways. Acceptance and API tests were made and benchmarks to RabbitMQ and the push engines were performed.

7.2. Contributions

Since April 2014, the knowledge that I acquired in this project regarding push notifications was useful to help other WIT Software employees. I worked in parallel with other project with the goal to help defining the solution design in the push notifications part.

As already mentioned in 7.1, the engineering component of my project helped WIT Software to renew their NP 4457 certification. This was an important achievement since my work was acknowledged as being a great example of research, development and innovation.

7.3. Deviations from the initial plan

Regarding the internship proposal made to DEI, there were slightly little changes made that do not affect the internship, but rather complement it. Anyway they should be mentioned.

The proposal made initially focused too much on advertising delivery via push notifications. However, after concluding the state of the art and analyzing companies that develop push messaging solutions, I realized the potential that this type of notifications can offer. Thereafter, I promptly suggested that the focus of the internship should not be so confined to advertising but instead, on its mean of transportation (push notifications). Thus, a new world of opportunities emerged with features that can offer to potential customers, ways of targeting segments of users depending on different variables, for example their location, which nowadays is a feature of a great demand once it allows application owners to reach users in a more marketing oriented way. Allied to this new universe, arose the idea of providing the service in two major areas, one for less experienced users through a web platform, and another designed for developers, providing them an API so they can communicate dynamically depending on their needs.

However, the vision to do something different and with more impact did not end here. One of the goals established by the company for my internship passed by providing this service for mobile network operators. However, being a very versatile service we came to the idea that it was possible to reach a much broader market by turning the product into something much wider, more generic, that could be used by network operators as well as by software developers who want to reach their customers with push notifications.

With this explanation, it is noticeable and understandable that these changes were made in order to make the internship more interesting and complete, both technically and commercially.

7.4. Future work

Although all the project's goals were successfully achieved, I have several ideas that could bring the gateway's capability much more forward and make it unique in the push notifications/marketing business. Some of them are the following:

- iBeacons

iBeacons is an indoor proximity system presented by Apple. A beacon is an equipment similar to a router that is placed on a wall for example, and emits Bluetooth low energy signals to detect devices around its location. A common use case would be for a beacon to trigger a push notification on a nearby device in order to warn him of some event or approximation to a point of interest.

- Passbook Integration

Passbook is Apple's application to store tickets, boarding passes or discount coupons. Making the iOS framework capable of delivering tickets to the device's passbook would be an advantage.

- Google Wallet Integration

Like Passbook, Wallet is Google's application to store a user's offers in one place. It would be a nice feature to integrate the Android Library with this application.

- More segment creation criteria

Having more criteria available when creating segments would be good to reach users different groups of users. For example, creating segments of users who have been in a certain location during the past week, or that were in a place at least once, or even those who have been in more than x countries in the last y weeks/months.

- Location-aware automated notifications

Create automated notifications that are triggered when a certain conditions happens. For example when a user enters a location or if he doesn't use the application for more than 1 week.

Now imagine all of these features combined. A clothing store could install beacons and have its app send discount coupons to the Passbook or Google Wallet along with a notification text: "50% discount on the t-shirts located in the left shelf". An airline company application could send discount tickets to users who were in more than 10 countries in the past 6 months. A trip advisor application could configure the gateway to deliver notifications to a user who arrives at Paris with the text: "Have you ever seen the Mona Lisa ? Visit Louvre!".

Possibilities are endless, this project can always be bettered and improved. After all for some reason there are big-sized companies developing business in this area.

7.5. Final Thoughts

Along the internship at WIT Software I've contacted for the first time with a real environment world of work. It was possible to access to a set of countless opportunities, learnings and challenges that contributed to my development either in personal and professional level. The diversity of technologies that I had to work with during the internship, provided me a vision of what is working in areas of development for mobile applications, back-end and front-end. The fact of having contacted with different areas of work, made me realize which were my personal preferences for future professional development.

One of the amazing surprises was the operationalization of engineering processes in the business environment. The great difference in software development, when comparison to academy environment, is the planning and organizational component. Its value is crucial to the success of a project and to prevent potential problems in advance

I would like to highlight the good ambience that prevails in the company, and in particular the feeling of mutual aid, always present. The possibility of sharing and acquiring knowledge among senior members of the team granted to evolve my technical skills at all levels.

The opportunity to intern at WIT Software has greatly contributed to the success of my internship since it allowed me to accomplish all the proposed objectives and also, as my personal option, to enrich the project by adding various features to the product developed.

This page was intentionally left in blank

References

- [1] mobiThinking. (n.d.). *Global mobile statistics 2012 Part C: Mobile marketing, advertising and messaging*. Retrieved October 3, 2013, from mobiThinking: <http://mobithinking.com/mobile-marketing-tools/latest-mobile-stats/c#ottmessaging>
- [2] Wikipedia. (October 29, 2013). *Apple Push Notification Service*. Retrieved November 14, 2013, from Wikipedia: http://en.wikipedia.org/wiki/Apple_Push_Notification_Service
- [3] Apple Inc. (September 18, 2013). *Apple Push Notification Service*. Retrieved November 14, 2013, from Apple iOS Developer Library: <https://developer.apple.com/library/ios/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/Chapters/ApplePushService.html>
- [4] Google Inc. (n.d.). *Overview*. Retrieved November 14, 2013, from Google Developers: <http://developer.android.com/google/gcm/gcm.html>
- [5] Microsoft Corporation. (December 4, 2013) *Push notifications for Windows Phone*. Retrieved December 7, 2013, from Windows Phone Dev Center: [http://msdn.microsoft.com/en-us/library/windowsphone/develop/ff402558\(v=vs.105\).aspx](http://msdn.microsoft.com/en-us/library/windowsphone/develop/ff402558(v=vs.105).aspx)
- [6] Push IO Inc. (n.d.). *About Push IO Inc.* Retrieved November 17, 2013, from LinkedIn: <http://www.linkedin.com/company/push-io-inc>
- [7] The Wall Street Journal. (January 22, 2014). *Responsys Acquires Leading Cloud Provider of Mobile and Web Push Notifications, Push IO*. Retrieved June 20, 2014 from: <http://online.wsj.com/article/PR-CO-20140122-906607.html>
- [8] Push IO Inc. (n.d.). *Choose a Plan*. Retrieved November 17, 2013, from Push IO: <http://push.io/pricing/>
- [9] Cutler, K. (April 25, 2013). *Facebook Buys Parse To Offer Mobile Development Tools As Its First Paid B2B Service*. Retrieved November 17, 2013, from Tech Crunch: <http://techcrunch.com/2013/04/25/facebook-parse/>
- [10] Parse. (n.d.). *Pricing that scales with your needs*. Retrieved November 17, 2013, from Parse: <https://parse.com/plans>
- [11] Urban Airship. (n.d.). *About Us*. Retrieved November 19, 2013, from Urban Airship: <http://urbanairship.com/about>
- [12] Urban Airship. (October 1, 2013) *Pricing, Billing and Invoice Questions*. Retrieved November 19, 2013, from Urban Airship Support Center: <https://support.urbanairship.com/customer/portal/articles/1061364-pricing-billing-and-invoice-questions>
- [13] Urban Airship. (n.d.). *How to Buy*. Retrieved November 19, 2013, from Urban Airship: <http://urbanairship.com/products/how-to-buy>
- [14] MacRae, D. (October 4, 2013). *IBM acquires Xtify to help digital marketers reach mobile customers*. Retrieved November 19, 2013, from CBR Online: <http://www.cbronline.com/news/ibm-acquires-xtify-to-help-digital-marketers-reach-mobile-customers>
- [15] Xtify Inc. (n.d.). *Choose a Package*. Retrieved December 24, 2013, from xtify: <http://www.xtify.com/pricing.html#tab=tab-A>

- [16] Objectivian. (March 15, 2012). *Welcome to Xtify 2.0*. Retrieved December 24, 2013, from Objectivian: <http://www.objectivian.com/2012/03/welcome-to-xtify-2-0/>
- [17] Jacob, S. (n.d.). *Speed Is A Killer – Why Decreasing Page Load Time Can Drastically Increase Conversions*. Retrieved June 23, 2014 from: <http://blog.kissmetrics.com/speed-is-a-killer/>
- [18] Jersey. (May 26, 2014). *RESTful Web Services in Java*. Retrieved May 27, 2014 from: <https://jersey.java.net>
- [19] Salvan, M. (n.d.). *A quick message queue benchmark: ActiveMQ, RabbitMQ, HornetQ, QPID, Apollo...* Retrieved January 13, 2014 from Muriel's Tech Blog: <http://x-aeon.com/wp/2013/04/10/a-quick-message-queue-benchmark-activemq-rabbitmq-hornetq-qp-id-apollo/>
- [20] RabbitMQ. (n.d.). *RabbitMQ Performance Measurements, part 2*. Retrieved January 13, 2014 from RabbitMQ: <http://www.rabbitmq.com/blog/2012/04/25/rabbitmq-performance-measurements-part-2/>
- [21] Turakhia, B. (May 6, 2010). *RabbitMQ vs Apache ActiveMQ vs Apache qpid*. Retrieved June 21, 2014 from Bhavin's Blog: <http://bhavin.directi.com/rabbitmq-vs-apache-activemq-vs-apache-qp-id/>
- [22] MacMullen, S. (n.d.). *[rabbitmq-discuss] Connection and Channel Management*. Retrieved May 20, 2014 from RabbitMQ Lists: <http://lists.rabbitmq.com/pipermail/rabbitmq-discuss/2012-March/018683.html>
- [23] Bahar, H. U. (n.d.). *World City Locations*. Retrieved May 31, 2014 from GitHub: <https://github.com/bahar/WorldCityLocations>
- [24] robotadam (n.d.). *Performance of Push Notification Service*. Retrieved June 4, 2014 from iPhoneDevSDK Forum : <http://iphonedevsdk.com/forum/iphone-sdk-development/57520-performance-push-notification-service.html>
- [25] Ryan. (n.d.). *Apple push notification limitation*. Retrieved June 4, 2014 from stackoverflow: <http://stackoverflow.com/questions/6421252/apple-push-notification-limitation>
- [26] Fhanik. (n.d.). *The Tomcat JDBC Connection Pool*. Retrieved June 6, 2014 from Apache People: <http://people.apache.org/~fhanik/jdbc-pool/jdbc-pool.html>
- [27] Hoogveld, M. (March 11, 2013). *Project Google – Phase 1: Get found by Google*. Retrieved December 19, 2013, from Mark Hoogveld Wordpress: <http://markhoogveld.files.wordpress.com/2013/03/scrum-overview-mark-hoogveld.jpg>
- [28] Sheehan, J. (November 15, 2013). *Introducing Runscope Radar: Automated API Testing and Monitoring*. Retrieved June 13, 2014, from Runscope Blog: <http://blog.runscope.com/posts/introducing-runscope-radar-automated-api-testing-and-monitoring>
- [29] YSlow. (n.d.). *Web Performance Best Practices and Rules*. Retrieved June 10, 2014, from Yahoo! Developer Network: <https://developer.yahoo.com/yslow/>
- [30] Nielsen, J. (May 3, 1998). *Cost of User Testing a Website*. Retrieved June 17, 2014, from Nielsen Norman Group: <http://www.nngroup.com/articles/cost-of-user-testing-a-website/>

[31] Google Developers. (n.d.). *CodePro Analytix User Guide*. Retrieved on June 14, 2014 from: <https://developers.google.com/java-dev-tools/codepro/doc/>

This page was intentionally left in blank

Bibliography

Figueiredo, D. A. (May 9, 1997). *Estratégia para a elaboração de uma tese*. Consulted on September 21, 2013, from Eden DEI: <http://eden.dei.uc.pt/~ctp/teses.htm>

Science Buddies. (n.d.). *Writing a Bibliography: APA Format*. Consulted on September 21, 2013, from Science Buddies: http://www.sciencebuddies.org/science-fair-projects/project_apa_format_examples.shtml

GSMA. (n.d.). *Rich Communications*. Consulted on September 23, 2013, from GSMA: <http://www.gsma.com/futurecommunications/rcs/>

Green, T. (February 27, 2012). *Trying to breathe some life into struggling Rich Communications Suite project*. Consulted on September 24, 2013, from mobile entertainment: <http://www.mobile-ent.biz/news/read/mwc-2012-gsma-unveils-joyn-brand-for-rcs/017183>

Chavin, J., Ginwala, A. & Spear, M. (n.d.). *The future of mobile messaging: Over-the-top competitors threaten SMS*. Consulted on September 24, 2013, from mckinsey: http://www.mckinsey.com/~media/mckinsey/dotcom/client_service/Telecoms/PDFs/Future_mobile_messaging_OTT.ashx

MVTUBB. (n.d.). *From GSMA: The RCE Project and Service Vision*. Consulted on September 25, 2013, from Mobile Cloud Services: <http://mobile-cloud-services.com/2011/08/01/rich-communications-ecosystem/>

GSM Association. (September 26, 2013). *joyn Blackbird Product Definition Document*. Consulted on September 27, 2013, from GSMA: <http://www.gsma.com/futurecommunications/wp-content/uploads/2013/10/Blackbird-Product-Description-Document-v2.0.pdf>

GSM Association. (July, 2012). *RCS-e v1.2 joyn Hot Fixes User Experience Guidance Document*. Consulted on September 30, 2013 from GSMA: <http://www.gsma.com/futurecommunications/wp-content/uploads/2012/10/V1-2joynHotFixesUXGuidanceDocument-July-2012.pdf>

Google Cloud Platform. (n.d.). *Orchestrating iOS Push Notifications on Google Cloud Platform*. Consulted on November 6, 2013 from: <https://cloud.google.com/developers/articles/ios-push-notifications>

Kishore, M. (July 5, 2010). *Understanding apple push notification in less than 4 mins..* Consulted on November 14, 2013, from Youtube: <http://www.youtube.com/watch?v=sUy0Cjzq8c8>

ParivedaInterns. (July 9, 2012). *Push Notifications: What are they and how do I send them?* Consulted on November 14, 2013, from Youtube: <http://www.youtube.com/watch?v=ATYhOIK11QM>

Push IO Inc. (n.d.). *Features*. Consulted on November 17, 2013, from Push.io: <http://push.io/features/>

Eskenazi, J. (July 22, 2013). *Advanced Segmentation*. Consulted on November 17, 2013, from Push IO Support: <https://pushio.zendesk.com/entries/24913171-Advanced-Segmentation>

Parse. (n.d.). *The perfect cloud for your apps*. Consulted on November 17, 2013, from Parse: <https://parse.com/products>

Parse. (n.d.). *Parse Push. Creating, scheduling, and segmenting push notifications just got a whole lot easier*. Consulted on November 17, 2013, from Parse: <https://parse.com/products/push>

- Urban Airship. (n.d.). *Newsstand*. Consulted on November 19, 2013, from Urban Airship Docs: http://docs.urbanairship.com/connect/ios_push.html#newsstand
- Urban Airship. (n.d.). *Products*. Consulted on November 19, 2013, from Urban Airship: <http://urbanairship.com/products>
- Williams, A. (October 3, 2013). *IBM Acquires Xtify, A Mobile Messaging Company*. Consulted on November 19, 2013, from Tech Crunch: <http://techcrunch.com/2013/10/03/ibm-acquires-xtify-a-mobile-messaging-company/>
- Xtify Inc. (n.d.). *xtify features*. Consulted on November 19, 2013, from xtify: <http://www.xtify.com/products.html#tab=tab-campaign>
- McCobb, J. (February 11, 2013). *Engage customers with push notification services*. Consulted on November 21, 2013, from Tech Republic: <http://www.techrepublic.com/blog/ios-app-builder/engage-customers-with-push-notification-services/>
- Martínez, A. (November 28, 2012). *How To Choose the Best Backend Provider for your iOS App: Parse vs. Stackmob vs. Appcelerator Cloud and More!* Consulted on November 21, 2013, from Ray Wenderlich: <http://www.raywenderlich.com/20482/how-to-choose-the-best-backend-provider-for-your-ios-app-parse-vs-stackmob-vs-appcelerator-cloud-and-more>
- Ewan. (January 6, 2012). *Urban Airship is absolutely flying: hires former Skype CSO Christopher Dean*. Consulted on November 21, 2013, from Mobile Industry Review: <http://www.mobileindustryreview.com/2012/01/urban-airship-is-absolutely-flying-hires-former-skype-cso-christopher-dean.html>
- Wikipedia. (December 3, 2013). *Best-effort delivery*. Consulted on December 4, 2013, from Wikipedia: http://en.wikipedia.org/wiki/Best-effort_delivery
- Griffiths, R. (n.d.). *GUIDE – Define Users and Usability Requirements*. Consulted on December 13, 2013, from University of Brighton: <http://www.it.bton.ac.uk/staff/rng/teaching/notes/guide/GUIDE1UsersUsability.html>
- Dalbey, J. (n.d.). *Non-Functional Requirements*. Consulted on December 13, 2013, from Cal Poly Professor page: <http://users.csc.calpoly.edu/~jdalbey/SWE/QA/nonfunctional.html>
- Debono, M. (April 5, 2012). *Functional vs Non Functional Requirements*. Consulted on December 13, 2013, from ReQtest: <http://www.reqtest.com/blog/functional-vs-non-functional-requirements/>
- OpenUP Administrator. (July 1, 2008). *Supporting Requirements*. Consulted on December 13, 2013, from OpenUP Wiki: <http://process.osellus.com/sites/wiki/OpenUP/Wiki%20Pages/Guidance%20-%20Supporting%20Requirements.aspx>
- Hope, P. (2009). *Software Security Requirements. The Foundation for Security*. Consulted on December 13, 2013, from Cigital: http://www.cigital.com/presentations/SecurityReqs_Hope_ISSA09.pdf
- Scrum Alliance Inc. (n.d.). *Why Scrum?* Consulted on December 18, 2013, from Scrum Alliance: <http://www.scrumalliance.org/why-scrum>
- Scrum Alliance Inc. (n.d.). *Core Scrum – Values and roles*. Consulted on December 18, 2013, from Scrum Alliance: <http://www.scrumalliance.org/why-scrum/core-scrum-values-roles>

- Wikipedia. (December 18, 2013). *Scrum (software development)*. Consulted on December 18, 2013, from Wikipedia: [http://en.wikipedia.org/wiki/Scrum_\(software_development\)](http://en.wikipedia.org/wiki/Scrum_(software_development))
- Wikipedia. (November 29, 2013). *Non-Functional requirement*. Consulted on December 24, 2013, from Wikipedia: http://en.wikipedia.org/wiki/Non-functional_requirement
- Barthel, J. (September 13, 2009). *Getting started with AMQP and RabbitMQ*. Consulted on February 27, 2014 from infoQ: <http://www.infoq.com/articles/AMQP-RabbitMQ>
- High Scalability. (April 9, 2012). *The Instagram Architecture Facebook Bought For A Cool Billion Dollars*. Consulted on February 27, 2014 from: <http://highscalability.com/blog/2012/4/9/the-instagram-architecture-facebook-bought-for-a-cool-billio.html>
- Hadlow, M. (September 24, 2013). *RabbitMQ: AMQP Channel Best Practices*. Consulted on February 28, 2014 from DZone: <http://architects.dzone.com/articles/rabbitmq-amqp-channel-best>
- Bick, C. (June 16, 2012). *Messaging with RabbitMQ – A Review*. Consulted on February 28, 2014 from bit suppliers: <http://bitsuppliers.com/messaging-with-rabbitmq/>
- Quartz Scheduler. (n.d.). *Quartz Features*. Consulted on March 23, 2014 from: <http://quartz-scheduler.org/overview/features>
- Codingpedia. (n.d.). *Tomcat JDBC Connection Pool configuration for production and development*. Consulted on April 9, 2014 from: <http://www.codingpedia.org/ama/tomcat-jdbc-connection-pool-configuration-for-production-and-development/>
- MuleSoft. (n.d.). *Improving Apache Tomcat Security – A Step by Step Guide*. Consulted on May 2, 2014 from: <https://www.mulesoft.com/tcat/tomcat-security>
- Chetan S., Kumar G., Dinesh, K., Mathew, K., & Abhimanyu M.A. (n.d.). *Cloud Computing for Mobile World*. Consulted on May 7, 2014 from: <http://chetan.ueuo.com/projects/CCMW.pdf>
- OReillyMedia. (February 12, 2009). *Tomcat Performance Tuning*. Consulted on May 15, 2014 from: <http://www.devshed.com/c/a/braindump/tomcat-performance-tuning/>
- Generic Articles (December 6, 2013). *How to optimize tomcat performance in production*. Consulted on May 16, 2014 from: http://www.genericarticles.com/mediawiki/index.php?title=How_to_optimize_tomcat_performance_in_production
- RabbitMQ. (n.d.). *What can RabbitMQ do for you?* Consulted on May 29, 2014 from: <http://www.rabbitmq.com/features.html>
- Klishin, M., & Duncan, C. (n.d.). *AMQP 0.9.1 Model Explained*. Consulted on May 29, 2014 from: <http://rubydoc.info/github/ruby-amqp/amqp/master/file/docs/AMQP091ModelExplained.textile>
- Ivan Ristic (September 14, 2012). *CRIME: Information Leakage Attack against SSL/TLS*. Consulted on June 8, 2014 from Qualys Blog: <https://community.qualys.com/blogs/securitylabs/2012/09/14/crime-information-leakage-attack-against-sslts>
- BREACH. (n.d.). *SSL, GONE IN 30 SECONDS*. Consulted on June 8, 2014 from: <http://breachattack.com>

Poole, N. (November 21, 2010). Preventing CSRF Attacks with AJAX and HTTP Headers. Consulted on June 9, 2014 from: <https://nealpoole.com/blog/tag/x-requested-with/>

Apache Software Foundation. (n.d.). *ab – Apache HTTP server benchmarking tool*. Consulted on June 10, from: <http://httpd.apache.org/docs/2.2/programs/ab.html>

Stackoverflow. (August 13, 2013). *RabbitMQ by Example: Multiple Threads, Channels and Queues*. Consulted on June 11, 2014 from: <http://stackoverflow.com/questions/18531072/rabbitmq-by-example-multiple-threads-channels-and-queues>

MacMullen, S. (March 9, 2012). *[rabbitmq-discuss] Connection and Channel Management*. Consulted on June 11, 2014 from: <http://lists.rabbitmq.com/pipermail/rabbitmq-discuss/2012-March/018683.html>

MacMullen, S. (April 17). *RabbitMQ Performance Measurements, part 1*. Consulted on June 12, 2014 from RabbitMQ: <http://www.rabbitmq.com/blog/2012/04/17/rabbitmq-performance-measurements-part-1/>

MacMullen, S. (April 25). *RabbitMQ Performance Measurements, part 2*. Consulted on June 12, 2014 from RabbitMQ: <http://www.rabbitmq.com/blog/2012/04/25/rabbitmq-performance-measurements-part-2/>

Software Testing Fundamentals. (n.d.). *Black Box Testing*. Consulted on June 13, 2014 from: <http://softwaretestingfundamentals.com/black-box-testing/>

Microsoft Developer Network. (n.d.). *Unit Testing*. Consulted on June 14, 2014 from: [http://msdn.microsoft.com/en-us/library/aa292197\(v=vs.71\).aspx](http://msdn.microsoft.com/en-us/library/aa292197(v=vs.71).aspx)

Google Developers. (n.d.). *CodePro Analytix User Guide*. Consulted on June 14, 2014 from: <https://developers.google.com/java-dev-tools/codepro/doc/>

Nielsen, J. (January 1, 1995). *10 Heuristics for User Interface Design: Article by Jakob Nielsen*. Consulted on June 17, 2014, from Nielsen Norman Group: <http://www.nngroup.com/articles/ten-usability-heuristics/>

Nielsen Norman Group (n.d.). *How to Conduct Usability Studies*. Consulted on June 17, 2014 from: <http://www.nngroup.com/reports/how-to-conduct-usability-studies/>

Danino, N. (September 3, 2001). *Heuristic Evaluation – a Step-By-Step Guide Article*. Consulted on June 17, 2014 from Site Point: <http://www.sitepoint.com/heuristic-evaluation-guide/>

Judy, B. (January 18, 2013). *How to Do a Heuristic Evaluation with Scores*. Consulted on June 17, 2014 from Big Design Events: <http://bigdesignevents.com/2013/01/how-to-do-a-heuristic-evaluation-with-scores/>

Dongcheul, L. (July, 2011). *Designing the Multimedia Push Framework for Mobile Applications*. International Journal of Advanced Science and Technology, Vol. 32.