TÉCNICO LISBOA

qNotify – Notification System and Integration with a Document Automation Platform

Leonardo Pereira Troeira

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Alberto Manuel Rodrigues da Silva
Engº João Paulo Pedro Mendes de Sousa Saraiva

Examination Committee

Chairperson: Prof. Daniel Jorge Viegas Gonçalves

Supervisor: Prof. Alberto Manuel Rodrigues da Silva

Member of the Committee: Prof. António Manuel Ferreira Rito da Silva

July 2020

Abstract

There has been a significant growth in the internet and web applications in the last years, which consequently has increased the need for web services and distributed applications to provide specific value to other complex applications.

This research proposes the design and development of the qNotify, a notification web server able to manage and deliver messages from diverse types (Push Notification, Email, SMS) to users. qNotify enables other applications to notify users, with the integration through web services.

The application under consideration to integrate with qNotify is qDocs. qDocs is a Document Automation Platform that enables citizens to create and manage personal documents and request specific documents to real organizations.

The result of this integration enables qDocs to notify citizens increasing awareness in time to optimize the process of document requesting, and other use cases such as to invite other citizens to participate in a document and to shared documents with other citizens.

qNotify is also tested and evaluated considering the following software quality attributes: Performance, Security, and Interoperability.

Keywords

Push; Notification; qNotify; Web services; Performance; Security; Interoperability; qDocs.

Resumo

Ao longo dos últimos anos tem havido um crescimento significante na internet e nas aplicações web que consequentemente aumentaram a necessidade das aplicações distribuídas e web services para fornecer serviços específicos para outras aplicações complexas.

Esta pesquisa propõe o desenvolvimento do qNotify, um servidor web de notificações que permite gerir e entregar mensagens do tipo (Push, Email, SMS) aos utilizadores. O qNotify permite a outras aplicações notificarem os seus utilizadores, com a integração através de web services.

A aplicação escolhida para integrar com o qNotify é o qDocs. O qDocs é uma Plataforma de Automação de Documentos que proporciona aos seus cidadãos criar e gerir documentos pessoais e fazer requerimentos de documentos específicos a organizações reais.

O resultado desta integração proporciona ao qDocs a funcionalidade de notificar os seus cidadãos, aumentando assim a perceção no tempo para otimizar o processo de requisição de documentos, e também outros casos de uso tais como o convite para participar num documento e a partilha de documentos com outros cidadãos.

O qNotify é também testado e avaliado sob os seguintes atributos de qualidade: Desempenho, Segurança e Interoperabilidade.

Palavras Chave

Push; Notificação, qNotify Web services; Desempenho; Segurança; Interoperabilidade; qDocs.

Acknowledgments

I would like to express my gratitude to my thesis supervisor Professor Alberto Manuel Rodrigues da Silva, which was always available whenever I had trouble or a question regarding this work, always helped me and guided in the right direction. Without his supervision, it would not have been possible to give this important step in my education.

I would like to acknowledge to my thesis co-supervisor Dr. João Saraiva, which was also one of the persons that always helped me to clarify and troubleshooting problems before and during this work.

I also want to thank my parents David and Fátima for all the support and encouragement that they gave me. My brothers Cristiano and Simão for being there for me and helping with everything in my life. And finally, but not least I want to thank my girlfriend Rita, that always encourage me to work on this thesis and contributed to my wellbeing and sanity during this road.

To each one of you, Thank you for everything!

Table of Contents

List of Figures

List of Tables

List of Acronyms

VAPID	Voluntary Application Server Identification
SW	Service Worker
REST	REpresentational State Transfer
SOAP	Simple Object Access Protocol
HTTP	Hypertext Transfer Protocol
FCM	Firebase Cloud Messaging
TTL	Time To Live
SaaS	Software as a Service
CA	Certification Authority
KDF	Key Derivation Function

1. Introduction

There has been significant growth in internet and web technologies in the last years which caused web applications to spread out into various business areas. The outstanding growth of web applications led to the emergence of two application metrics, named engagement and retention, related metrics used to define the application performance [1]. One particular feature currently available for communicating information in these applications is the “push” feature (also known as a “notification” feature or “alert” feature) [2].

Therefore, web push notifications come up as they are mainly focused on engagement and retention, with the addition of providing valuable information to users. Push notifications maintain the ability to inform users, while it captures their attention immediately. Whether an user is watching a movie or reading an article on his desktop or mobile device, when a push notification arrives, the user’s attention is immediately shifted to that [3]. How push notifications are used mostly depends on the business preferences and standard approaches [4].

This thesis presents and discusses a web server called qNotify. qNotify provides the ability to notify users via Push Notifications, Email, and SMS. For integration and further evaluation of qNotify with real use cases, it was used the qDocs system. qDocs is a web document automation platform intended to work on any device to provide a seamless interaction between citizen and administration services, making bureaucratic documents citizen centric [5].

In this thesis, we also evaluated qNotify, and its integration with qDocs, considering the following software quality attributes: Performance, Security, and Interoperability. This evaluation showed that qNotify is able to deliver 50.000 notifications to users in one minute, without producing errors and returning a correct response for every request. qNotify provides service only for authorized applications that prove their authenticity towards qNotify and assumes that every request exchanged is under TLS protocol, to ensure Confidentiality, Integrity and Authenticity in exchanged messages. qNotify also do not require to discover any application that will request for his (qNotify) services before runtime, instead qNotify is able to provide service to any application even when the registration of the application is made during runtime.

1.1. Motivation

qDocs is a Citizen-Centric and Multi-Curator Document Automation Platform. qDocs is a distributed platform for managed secure electronic documents, accessible through any connected device such as smartphones, tablets or computers. qDocs key feature allows the design of electronic documents (e.g. identification document, certificates, reports, forms, questionnaires) based on the orchestration of

services and data from different Curators (e.g., public and private organizations), managed according to a distributed, dynamic and secure manner [5]. After these document(s) approval by the specific curator (organization), the citizen gets access to document(s).

The problem that motivated this work emerged from qDocs necessity to notify citizens according to specific application events, such as when documents are approved by curators and consequently made available for citizens. This necessity to notify citizens is important in qDocs context because the timeline for a curator to approve a document, depends on each curator logistic, the citizen that is requesting for the document and the type of document. For instance, a request a certification degree takes more time to be approved by the curator (school) than a simple formulary request. qDocs also exposed the need to notify citizens when: a document is shared with another citizen; a citizen invites another citizen to participate in a document; and when a citizen is added or removed from a curator.

The approach to solve this problem is to increase awareness, by informing citizens when the documents are available to them and when other events occur, as previously presented.

Push notifications are the key channel preferred to inform citizens, but it is also object of research the Email and SMS channels to send messages to citizens. Push notifications notify users with information about an application's event, so they provide valuable and relevant updates, even when the application is not running. Chapter 2 details the Push Notifications subject and the qDocs platform, as they are the main subjects of these research.

1.2. Requirements

The general requirements considered during the design and development of qNotify are the following:

- Deliver notifications to users from Push, Email and SMS Channels according to user permissions;
- Users can consult and change their channels (Push, Email, SMS) permissions;
- Users can consult their notifications history;
- Users shall be notified in all devices where they are subscribed (i.e., user receives push notification in his tablet and PC at once);
- Integration with qDocs specific use cases (defined in subsection 5.2.4).

Considering these requirements, and given that the approach to solve this problem must be a monetary cost saving solution (i.e., commercial or proprietary services should be avoided), it was decided that the solution should be the development of a web server that would handle notifications, including features like: Send Notifications; Consult Notifications History; and Consult/Change User Channels (Push, Email, SMS) Permissions. The above requirements do not include performance, security, and interoperability requirements, because these are explicitly defined in Chapter 6.

1.3. Proposed Solution

This section presents the basic architecture of qNotify and its integration with qDocs. This is intended to be a simple and intuited architecture of the solution, just to give a general understanding of the basic interaction of qDocs and qNotify. As illustrated in Figure 1.1, qDocs platform will interoperate with qNotify web server, and vice-versa, via HTTPS requests such as: “SendNotification”, “GetNotifications”, “SavePermissions”, “GetPermissions”, “SaveSubscription”, “Unsubscribe”, “GetSubscription”, “SetUnseen”. The interaction between qNotify and the User is unidirectional: only qNotify delivers messages to users via Push, Email, and/or SMS channels. User and qDocs interactions are the same as in the qDocs original solution and are described in Section 2.2.

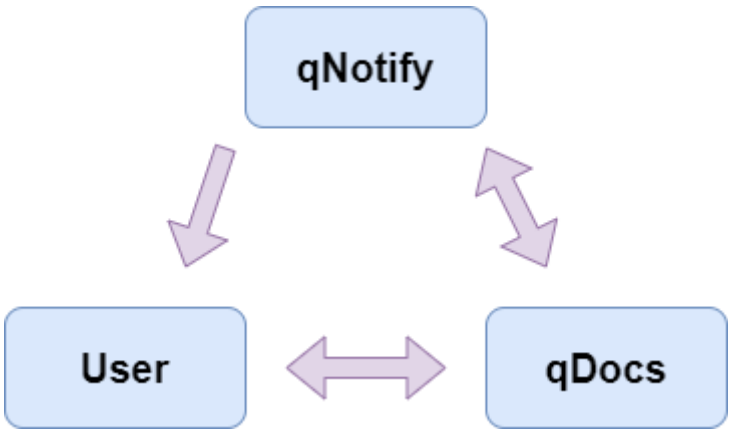

Figure 1.1. qNotify Integration with qDocs Basic Architecture.

1.4. Objectives

The purpose of this research is the design and development of the qNotify notification system and its integration with the qDocs platform. It is also part of the objectives, the evaluation of qNotify considering the qualities of Performance, Security, and Interoperability. To achieve this results there are some tasks to accomplish, namely:

- Analyse related work regarding to push notifications and qDocs;
- Define the development technologies to use;
- Design and implementation the qNotify system;
- Integration of qNotify with qDocs;
- Evaluation of the solution.

1.5. Dissertation Structure

This dissertation structure is composed of six chapters:

Chapter 2: This chapter provides background regarding to web push notifications, qDocs platform and the development technologies used to build the solution;

Chapter 3: This chapter presents related work regarding other authors proposals of notifications systems and how they interoperate with other components. It also presents a discussion of the existing push notification technologies compared with the chosen web push library;

Chapter 4: This chapter presents the design and implementation of qNotify, including the architecture, the domain model, algorithms used to solve problems and interfaces exposed;

Chapter 5: This chapter presents the integration of qNotify with qDocs with qDocs specific use cases;

Chapter 6: This chapter presents the evaluation of the solution according to software quality attributes: Performance, Security, and Interoperability;

Chapter 7: This chapter presents the conclusions and the future work regarding this thesis.

In addition, the dissertation includes three appendixes:

Appendix A: Presents performance test results made under Thread Groups 1 and 2 for both Test Plans (will be explained in chapter 6);

Appendix B: Presents performance test results made under Thread Groups 4 and 5 for both Test Plans (also explained in the evaluation chapter).

Appendix C: Presents qNotify API fully explained, with each data input and output, along with endpoint URL and description.

2. Background

This chapter introduces Push Notifications and qDocs Platform, as they are the main subjects discussed in this research. It is also provided background on technologies used to build the solution.

2.1. Push Notifications

Push notifications are a mechanism that allows applications to send messages that pop up on a user device via web browser or mobile app [6]. The key advantage of push is that web applications do not need to check if the user is online or not, it can simply request delivery of push message and the user will receive it as soon as possible.

To perform a correct web push notification, a Client App must obtain the Subscription from the Push Service, and then Distribute Subscription to Server App. When the Server App has the Subscription, it can Request Delivery to Push Service to Deliver Push Notification to that Subscription Endpoint [7]. Figure 2.1 presents a Push Notification Workflow as explained above.

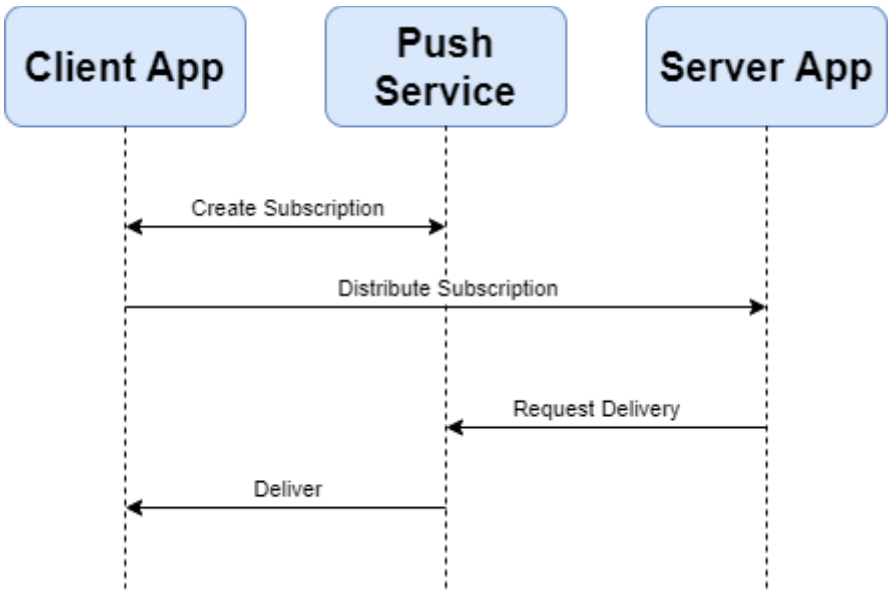

Figure 2.1. Push Notification Workflow.

2.1.1. Push Service

Push service is a third-party component that acts as an intermediary ensuring reliable and efficient delivery of push messages to a Client App. The term push service refers to a system that allows application servers to send push messages to a web client application [6].

A push service serves the push endpoint or endpoints for the push subscriptions it holds. The user connects to the push service used to create push subscriptions.

An application server requests delivery by sending a message to the push service, and the push service delivers that message with a push event [8].

2.1.2. Voluntary Application Server Identification (VAPID)

The presence of a push service raises security and privacy concerns, one of such concerns is authentication. Each subscription to push service has its own unique URL, this means that if such URL would leak, other parties would be able to send a push message to related subscription. Therefore, it is required an additional mechanism to limit the potential senders. This mechanism is Voluntary Application Server Identification (VAPID) [9].

VAPID is a technique that allows an application server to identify itself to a push service. This identification information is used by the push service to attribute requests that are made by the same application server to a single entity.

VAPID requires Application Server Keys (public and private key pair). The public key must be delivered to the client and private keys must be kept in the server. The server validates the client by deciphering the data with the correspondent private key, if the user is authenticated (ciphered the data with his public key), the server succeeds to decipher and authentication is granted, otherwise, is someone trying to impersonate and the server rejects connection [9].

2.1.3. Service Worker (SW)

For a client application support push notifications, it requires to install an active Service Worker (SW) in the user agent. To register the SW, the user must allow notifications from his user agent. Given permissions, the SW is installed in the user browser and run independently as an application that sends regularly a query to the provider server and ask for any new event happening and then respond to the client with a popup message [10].

2.1.4. User Agent

User agent is a software that is acting on behalf of the user, such as a web browser that retrieves, renders and facilitates end-user interaction with Web content.

In many cases, a user agent acts as a client in a network protocol used in communications within a client-server distributed computing system. In particular, the Hypertext Transfer Protocol (HTTP) identifies the client software originating the request, using a user-agent header, even when the client is not operated by a user [6].

2.1.5. Push Subscription

Whenever an application requires to send push messages to a client, it needs to create a push subscription specific to that client. A push subscription is a message delivery context established between the user agent and the push service on behalf of a web application. For every push subscription that is made, it is required a push endpoint associated with that subscription and a SW registration of the new subscription.

2.1.6. Push Unsubscription

There are two ways of unsubscribing (deactivate) a push subscription: (1) In the creation of the push subscription, by setting an associated subscription expiration date and time; or (2) By removing the registration of that specific subscription in the SW.

When a push subscription is deactivated, subsequent push messages for this push subscription will not be delivered.

2.1.7. Push

Push is a style of Internet-based communication where the request for a given transaction is initiated by the publisher or central server [6]. It is contrasted with pull/get, where the request for the transmission of information is initiated by the receiver or client.

2.1.8. Push Message

Push Message is data sent from an application server to a client application, through SW with the push subscription to which the message was submitted.

2.1.9. Push Message Content

Title: This attribute corresponds to the Title of the Notification.

Message: This parameter corresponds to the Message of the Notification.

URL: The user is redirected to this URL upon clicking on the Notification.

Time To Live (TTL): This property corresponds to the maximum period of time (seconds) for which push service will try to deliver the message.

Icon: This parameter corresponds to the Icon presented in the Notification.

Tag: The tag read-only property idea is that more than one notification can share the same tag, linking them together. One notification can then be programmatically replaced with another to avoid the users' screen being filled up with a huge number of similar notifications [11].

2.2. qDocs – Document Automation Platform

qDocs is a Citizen-centric and multi-Curator document automation platform for managing dynamic electronic documents that are accessible through any device, such as smartphone, tablet or computer [5].

qDocs platform intends to promote and facilitate the relationship between Citizens and Curators (e.g. public and private organizations) through any process that involves the request, publication, access, delivery, and sharing of electronic documents. It also acts as a wallet of bureaucratic documents, such as the ID, Driver's License, or Certifications, where the citizen can search and access them by document type, life event, or curator specific classification schema [12].

qDocs also provides features for managing dynamic documents, designed by multiple Curators, and for searching them under several criteria such as by the metaphor of life events (e.g., birth, death, education, health or professional life related documents), by document types (e.g., identification documents, certificates, requests for certificates, all sorts of requests, forms and declarations), by Curators (e.g., Tax Services, Municipalities, Public Services), or even by curator-specific classification [12].

2.2.1. qDocs – Architecture

qDocs architecture is defined by the integration of several applications, namely (as suggested by Figure 2.2):

- **qDocs/Citizen:** Aimed to be used by any Citizen looking for the benefits of the system;
- **qDocs/Curator:** Particularly oriented to the Curators that make up the qDocs ecosystem;
- **qDocs/Admin:** To manage the general configuration and operation of qDocs;
- **qDocs/qBox:** That provides data integration mechanisms with the business applications of the respective curators.

A citizen to get access to a document, the following tasks have to happen: (1) the citizen requests a document to the qDocs platform; (2) the qDocs platform sends a ”request acknowledged” message to qBox; (3) qBox sends a key to qDocs; (4) qDocs sends that key with the document template to the citizen’s device; (5) the citizen’s device sends the key directly to qBox; (6) qBox sends the respective data to the citizen; and (7) the document is generated in the citizen’s device; Finally and optionally (8) qBox may save the data from the citizen so he/she can later on access that document [12].

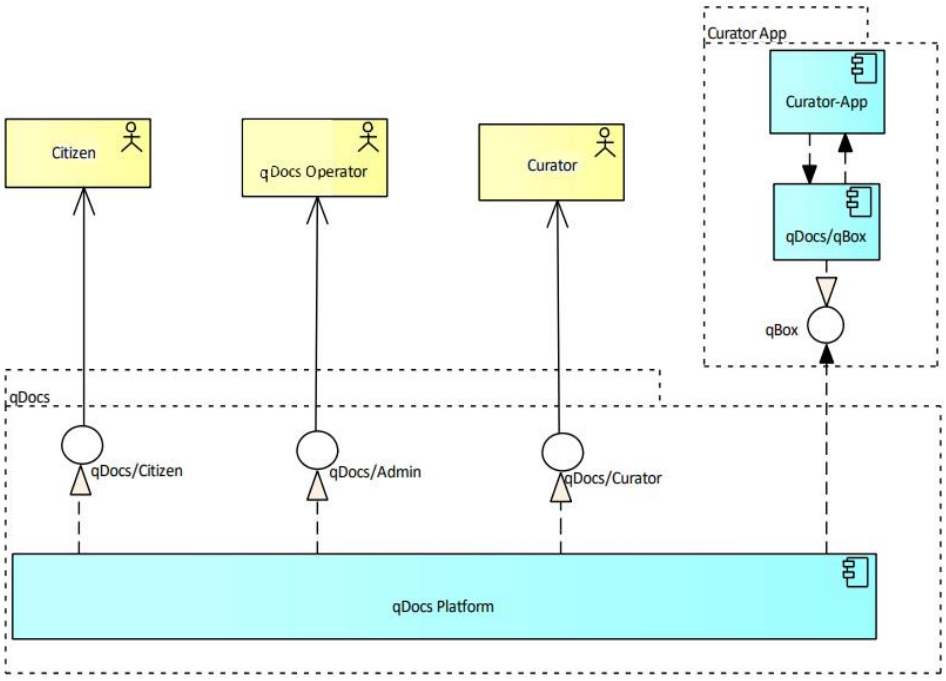

Figure 2.2 qDocs General Architecture (ArchiMate diagram) – extracted from [5].

2.2.2. qDocs/Citizen

The qDocs/Citizen application is used by each citizen, and provides the following features [13]: Access and management of his documents through the wallet/folder metaphor, where the citizen can have multiple roles as shown in Figure 2.3; Search and browse documents by classifiers defined by the respective Curators, namely by life events, document type, curator-specific classification schema; Share documents (using dynamic keys) with other users, being possible to limited in time these shares; Fill and submit input documents, feeding Curator forms; Integration with payment systems, allowing integrated document issuance fee collection; Handsfree authentication mechanism using qDocs Identity Token; Several levels of legally binding authentication, based on official systems ranging from ID smart cards to SMS based authentication methods (depending on integration with official mechanisms already in place); Authenticity validation system for critical identification documents (e.g., ID, driver license); Real time monitoring of activity, access to comprehensive dashboards and activity logs, including usage by third parties. Figure 2.3 presents an example of the multiple roles that a qDocs citizen can have.

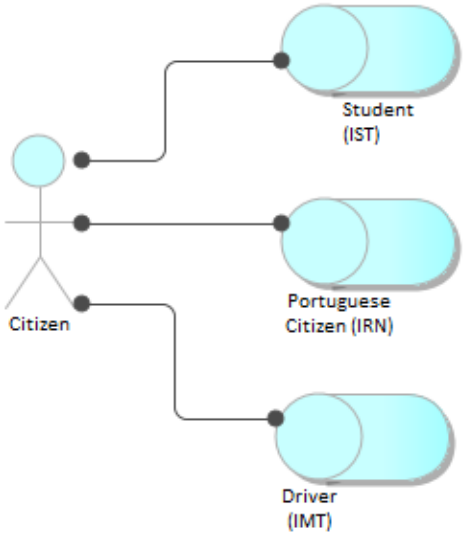

Figure 2.3. Multiple roles example of citizen entity can perform – extracted from [13].

2.2.3. qDocs/Curator

The qDocs/Curator application is used by different users (with different specific roles) defined at each curator level, and shall offer the following features [13]: Manage curator-specific users and roles assignment; Manage protocols and contracts with other Curators; Manage and configure internal and external data services (DataServices); Manage and configure merge fields and user interface objects (FormObjects, Snippets); Manage and configure document templates (DocumentTemplates); Monitor performed activity and transactions.

2.2.4. qDocs/Admin

The qDocs/Admin application allows users with the platform “Admin” role access to the following key features [13]: Configuration and management of platform-level configuration entities such as DocumentGroupPlatform (e.g., life-event), Country, and UserRole; Management of users includes create user (manually), create users (in bulk), or update users; Management of curators includes create, update, activate, suspend, and delete Curators; Monitoring activities and events relevant to the platform admin, this includes access to a comprehensive set of dashboards and activity logs;

2.2.5. qDocs/qBox

The qDocs/qBox application (or simply "qBox") is deployed and configured at each curator's computational environment, and supports the secure integration between the qDocs platform and each curator's specific business applications (e.g., ERP, CRM, DMS). qBox shall provide the following features: Management and versioning of data tables for supporting both input and output documents; Support secure output data requests, i.e., output data fields to be integrated in the produced documents; Support secure input data requests, i.e., input data fields of documents submitted; Access and management of input documents pending for approval (only accessed by CuratorDocsApproval users); Integration API to be used by curator’s business applications; Monitoring activities and events related to the qBox performance, includes access to a comprehensive set of dashboards and activity logs.

2.2.6. qDocs Technologies

qDocs is designed and implemented based on standard technologies, like databases being accessed using plain SQL, HTTPs, HTML5, CSS3, JavaScript, Typescript, XML and JSON. At any time qDocs documents can be exported to common standard document format, like PDF, XML and JSON. qDocs is built according to industry’s best practices in what concerns the generation and management of digital certificate, and fully enforcing strong encryption [5]. qDocs is deployed as a cloud and multiplatform technology, providing responsive access in both web browsers and mobile applications (in qDocs /Citizen).

2.3. Development Technologies

This section provides background on technologies used to build the solution presented in this research, describing the key features that enforced it.

qNotify webserver was developed in Node.js, as for qDocs front-end and back-end extensions it was used Angular and ASP.NET Core respectively, as these were already the languages qDocs was implemented. MongoDB is the database software used to store qNotify data. Some of these technologies (Node.js, Angular, ASP.NET Core) were learned by taking courses on Udemy [14].

2.3.1. Node.js

Node.js is an asynchronous event-driven JavaScript runtime, designed to build scalable network applications [15]. Node.js presents an event loop as a runtime construct instead of as library where it is common to have a blocking call to start the event-loop. Usually in other systems, behaviour is defined through callbacks at the beginning of a script, and at the end a server is started through a blocking call. In Node.js, there is no such start-the-event-loop call. Node.js simply enters the event loop after executing the input script and exits the event loop when there are no more callbacks to perform [15].

Concurrency and Throughput

JavaScript execution in Node.js is single-threaded, so concurrency refers to the event loop's capacity to execute JavaScript to callback functions after completing other work. Any code that is expected to run concurrently must allow the event loop to continue running as non-JavaScript operations, like I/O, are occurring [15].

To understand this concept, let's consider a case where each request to a web server takes 50ms to complete and 45ms of that 50ms is database I/O that can be done asynchronously. Choosing non-blocking asynchronous operations frees up that 45ms per request to handle other requests. This is a significant difference in capacity just by choosing to use non-blocking methods instead of blocking methods.

2.3.2. Express

Express is called the standard web application framework for Node.js, released as free and open-source software under the MIT License. It is designed for building web applications and APIs. One of the main features of express is routing. Routing refers to how an application's endpoints (URIs) respond to client requests [16]. Express routing methods specify a callback function called when the

application receives a request to the specified route (endpoint) and HTTP method. The routing methods can have more than one callback function as arguments, with multiple callback functions.

2.3.3. MongoDB

MongoDB is a cross-platform document-oriented database program [17]. Classified as a NoSQL database program, MongoDB uses JSON-like documents with schema, which means it stores data in JSON-like documents. MongoDB is developed by MongoDB Inc. and licensed under the Server Side Public License (SSPL) [17].

MongoDB provides among others, the following features: Rich JSON Documents; Powerful Query Language

JSON-like Documents – Schema Flexibility

One advantage of using databases with JSON data models is the dynamic and flexible schema they provide when compared to the rigid, tabular data models used by relational databases.

JSON documents are polymorphic (fields can vary from document to document within a single collection). Documents make modelling diverse record attributes easy to handle data of any structure. There is no need to declare the structure of documents to the database, because documents are self-describing and objects are persistent as they are created.

If a new field needs to be added to a document, it can be created without affecting all other documents in the collection, without updating a central system catalog and without taking the database offline. When there is required to make changes to the data model, the document database continues to store the updated objects without the need to perform costly “ALTER TABLE” operations – or worse, having to redesign the schema from scratch.

Through these advantages, the flexibility of the document data model is well suited to the demands of the application solution presented later in this thesis.

2.3.4. Angular

Angular is the Google cross-platform framework for front-end development [18]. It is TypeScript based (a language by Microsoft that is converted to JavaScript for browser compatibility) [19], and allows easy manipulation of HTML and CSS.

Routing is used as a way to link requested URI’s to the corresponding components and passing on information through resolvers. To protect and inform the user of possible mistakes and malfunctions guards are used as a verification method [13].

qDocs Platform front-end was already developed in Angular framework, reason why the required extensions to the qDocs front-end to integrate with qNotify was also coded in Angular.

2.3.5. ASP.NET Core

ASP.NET Core is the Microsoft cross-platform framework for internet-connected applications [20]. It is used as a tool to build web app's back-end. Data Transfer Objects (DTOs) are used to interact with the database, created from these objects through migrations. To establish HTTP endpoints Controllers are used, defining the URI's associated with the resources that are provided.

ASP.NET Core was used in the same conditions as Angular framework, only for required extensions regarding the back-end of qDocs Platform, to integrate with qNotify functionalities.

3. Related Work

This chapter presents the related work of an approach on how to design and build a notification system with a comparison of the similarities and differences of this prototype with qNotify. It also presents a comparison of webservices exchange pattern and Firebase Cloud Messaging (FCM). Finally, it is presented a discussion based on other push notification services versus qNotify solution.

3.1. Push Server Prototype

To build a server able to deliver push notifications, it is important to analyse other works. This section describes the Push Server Prototype developed by the authors [1]: Emre Isikligil; Semih Samakay; Deniz Kılınç.

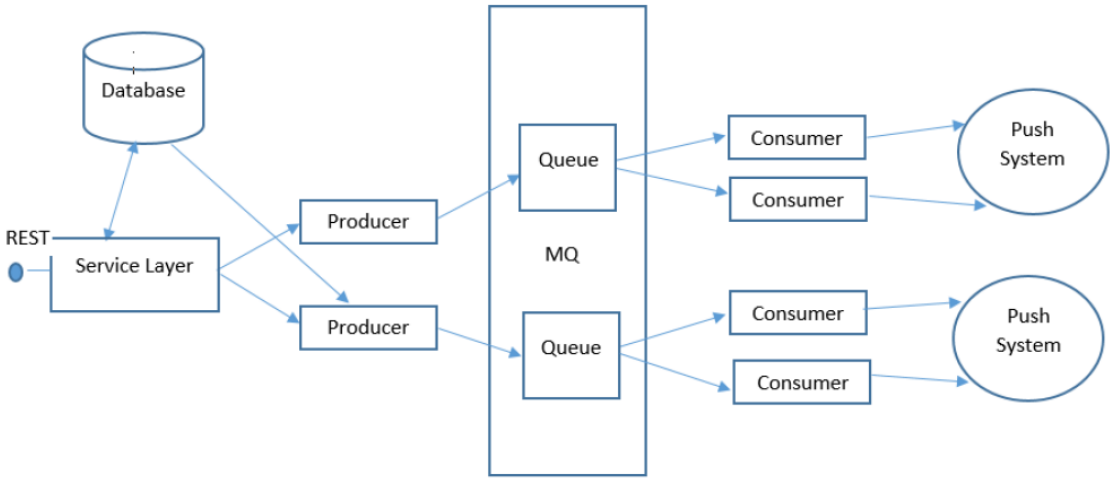

Figure 3.1. Push Server Example – extracted from [1].

From Figure 3.1, we can conclude that the architecture of this prototype system consists of 6 modules, REST interface, Service Layer, Database, message Producer, message Consumer, and Queue manager.

3.1.1. Service Layer

The service layer is the module that manages the application. It provides inter-module communication. The data provided from the interface is stored in the database by the services in this layer. This layer is also where the notification process is initiated to process the data in the database [1].

3.1.2. Producer

Producer threads are created by the service layer depending on the device type. They are used for preparation of notifications to be sent out and lining them up in the queues to be delivered to the consumers. These producers are responsible for generating notification messages by retrieving data from the related data tables. Each producer is responsible for only one notification process. Threads are terminated after all messages have been created [1].

3.1.3. Consumer

The message consumers receive messages waiting in the queue and send them to the notification systems depending on device type [1].

3.1.4. Queue Manager

The queue manager provides asynchronous operation of message producers and consumers. Incoming messages are queued up and served to consumers on a first come first served basis [1].

3.1.5. Comparison with qNotify

The solution developed in our research (qNotify) is in some parts based on the architecture presented by these authors, with the difference that there is no Producer/Consumer Model explicitly designed. Instead, it is used the advantages of Node.js + Express Framework that allows us to perform asynchronous endpoint calls, adding the system a non-blocking state in every request (explained in subsection 2.3.1).

Another difference from this prototype to qNotify solution, is that qNotify provides three delivering message systems (Push, Email, SMS), comparing with this prototype where there is only Push system integration. As for the Rest Interface, the Service Layer, and Database interactions, it was taken the same approach of this prototype.

3.2. Message Exchanging

This section briefly describes the operation and architectures of Web Services (WS) and Firebase Cloud Messaging (FCM). It also presents a comparison of both architectures, showing which message exchanging architectures perform better. Both are designed for message exchange, but unlike WS which follows the classic client-server communication paradigm, FCM forces all messages to pass through servers (managed by Google), making them distinct in terms of architecture.

On the other hand, these servers enable FCM to provide value-added services, such as push technology, data encryption, and collapsible messages, which cannot be directly provided by any WS [21]. These dissimilarities should likely imply a difference in performance.

3.2.1. Web Services (WS)

Web Services (WS) are programmatic interfaces for communication between software agents using the HyperText Transfer Protocol (HTTP) [21]. Software agents that provide services are referred to as service providers. They are responsible for publishing programmatic interfaces and description of the services to discovery agencies. The published services are discoverable and consumable over the web by other software agents called service requesters, who need to discover the published services from discovery agencies before they can start using them. Once services are discovered, service requesters interact with providers through message exchanging. Figure 3.2 depicts the WS architecture modelling the interactions among the service provider, service discovery service, and service requester.

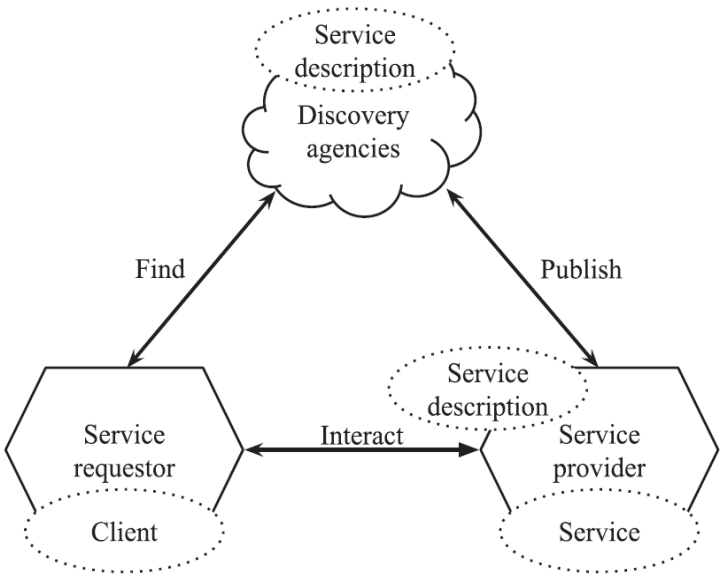

Figure 3.2. WS Message Exchange Pattern – extracted from [21].

As far as the messages exchanged between the service requester and provider are concerned, different WS implementations use different protocols and message structures. Most well-known WS architectural styles currently used on the internet are: REpresentational State Transfer (REST) [22]; and Simple Object Access Protocol (SOAP) [23].

REpresentational State Transfer (REST)

REST is an architectural style that defines a set of recommendations for designing loosely coupled applications that use the HTTP protocol for data transmission. In its purest form, a RESTful WS uses HTTP methods (such as POST, GET, PUT and DELETE), is stateless to improve performance by saving bandwidth and minimizing server-side application state caching, uses Uniform Resource Identifier (URI) to address resources, is data-driven and transfers data structures by serializing them in eXtensible Markup Language (XML) or JavaScript Object Notation (JSON) [12].

Simple Object Access Protocol (SOAP)

SOAP is a protocol defining strict rules for messaging and Remote Procedure Calls (RPCs) using XML format that uses any application layer protocol and is usually coupled with Web Services Description Language (WSDL). In its purest form, it also uses HTTP methods, is stateless (but can be made stateful), function-driven and transfers data structures by serializing them only in XML [24].

3.2.2. Firebase Cloud Messaging

Firebase Cloud Messaging (FCM) also known as Google Cloud Messaging (GCM), is the google cross-platform cloud solution for sending messages and notifications to Android, iOS, and web applications. FCM is a non-free software, licensed by google.

As illustrated in Figure 3.3, the fundamental components in the FCM architecture are [21]: FCM connection server; Trusted environment with Application server based on HTTP and XMPP; Trusted environment with Cloud functions; Client application. FCM supports notification and data messages. Notification messages are automatically handled by the FCM SDK to show a notification on behalf of the client application. Data messages, on the other hand, have only custom key-value pairs and are completely handled by the client application.

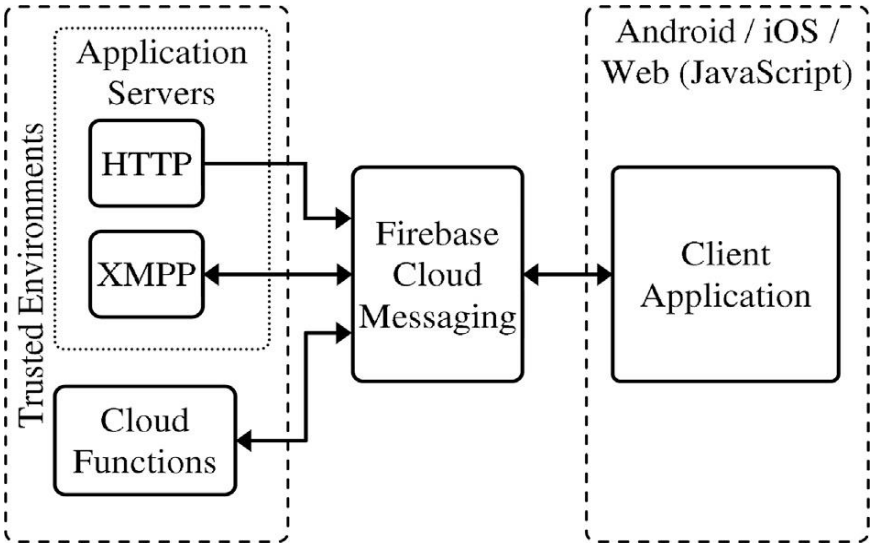

Figure 3.3. FCM general architecture – extracted from [21].

3.2.3. Comparison (WS vs FCM)

As explained in the previous subsections, WS strongly differs from FCM in terms of architecture, FCM have a third entity between the client and the application server or trusted environment, which adds some delays in the communication path between a mobile terminal and an Internet-located server.

According to these authors [21], “A set of tests in a simple client-server scenario were done for four messaging techniques: SOAP WS, RESTful WS, and FCM. The tests evaluated time-related performance, data usage, and battery duration”.

The results confirmed that REST WS outperforms other communication techniques in all evaluated performance matrices. FCM showed the worst performance, however, it is important to mention that FCM supports more features out-of-the-box than WS, such as push technology, data encryption, native Android support as well as official iOS SDK support by Google. However, for scenarios where the delivery of larger payloads is required or where there are time constraints, WS should be the preferred solution [21].

To conclude, FCM is a message exchanging service as WS, with the advantage that also provides inside features such as Push technology, which is one of the subjects of this thesis. On the other hand, WS outperform FCM in terms of performance, which is also a quality attribute to considered in this thesis, and although it doesn’t provide push technology feature, it can be implemented using external software (Push API). The solution of this thesis (qNotify) is implemented using REST Web Services, and the push technology is implemented using the Push API. Both technologies are open source and free to use, while FCM is a pay to use service.

3.3. Push, Email and SMS Technologies

This section describes technologies used in qNotify, to deliver push notifications, emails, and SMS's messages to users. subsection 3.3.1 describes the web Push API (used to build the Push component), in subsection 3.3.2 it is described the Nodemailer library (used to build the Email component), and in subsection 3.3.3 it is presented the Nexmo service (used to build the SMS component). Finally, subsection 3.3.4 discusses why it was used Push API, Nodemailer and Nexmo technologies to build Push, Email and SMS components instead of using a "Software as a Service" (SaaS) solution.

3.3.1. Push API

The Push API is a web push service under the W3C license that allows the use of software in commercial applications. Web Push interface allows web server applications to send push messages via the push service. These application servers can send a push message at any time, even when a web application or user agent is inactive. The push service ensures reliable and efficient delivery to the user agent. Push messages are delivered to the service worker that runs in the origin of the web application, which can use the information in the message to update local state or display a notification to the user.

The Push API contains two key components, the Push Service and Service Worker. The role of push service is to handle subscription and push requests. The Service Worker role is to be installed in the user agent so that the user be able to receive the push message.

As illustrated in Figure 3.4, to perform a push notification using the Push API, the following steps may happen: (1) Register the Service Worker; (2) Request permission from the user; (3) Subscribe the user and get the PushSubscription object; (4) send the PushSubscription object to the server; (5) register the new client subscription; (6) Send the Push message.

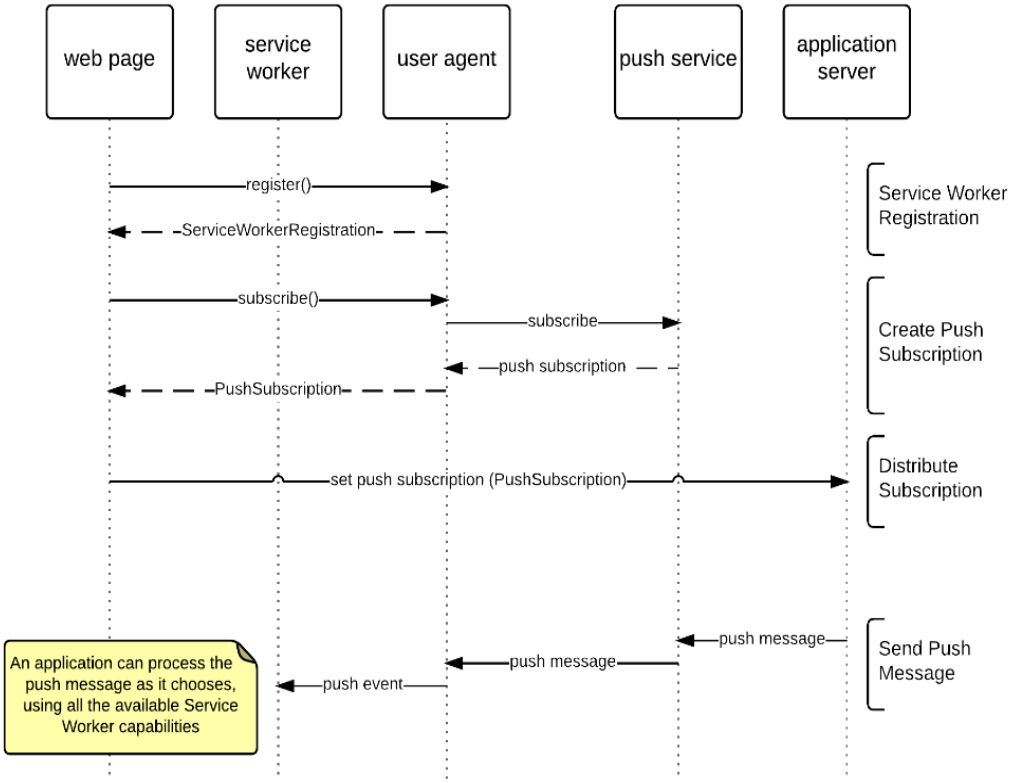

Figure 3.4. Push API flow of events for subscription and push message delivery – extracted from [6].

Service Workers, Subscriptions and User Agents

At this point, Service Workers, Subscriptions, and User Agents are explained individually (despite they are related to each other). Concisely, the User Agent is usually the browser from where the user is accessing to; the Service Worker is registered in that specific browser, so that when the Push occurs, the browser pops up the Notification. To create a correct Subscription, a Service Worker must be registered in the User Agent, the outcome of that registration, it is called the Subscription. The Subscription contains the endpoint for that User Agent, the VAPID Keys, and the expiration date.

Web Push Facts

It is not possible to register more than one service worker instance per user agent simultaneously, meaning that if the user agent has already a service worker registration, it will be used that service worker instance, instead of register another.

Different browsers in the same machine can have different service worker registrations, meaning that if a user authenticates with Chrome and allows to receive Push Notifications, a service worker is registered in Chrome, and if the authenticates with Firefox and allows to receive Push Notifications, a different service worker instance is registered in Firefox. If the user authenticates again with one of those browsers, the previous service worker registered instance is used.

3.3.2. Nodemailer

Nodemailer is a module for Node.js applications to allow email sending. The project was released in 2010 and today is still one of the solutions that most Node.js applications use. Nodemailer is licensed under MIT license (it allows the use of the software in commercial applications) [25].

Nodemailer provides the following features [25]: A single module with no dependencies; Unicode support to use any characters, including emojis; Use HTML content, as well as plain text alternative; Add Attachments to messages; Secure email delivery using TLS.

SMTP is the main protocol used between different email hosts, and it is also the main transport in Nodemailer for delivering messages. Many email delivery providers support SMTP based sending.

Simple Mail Transfer Protocol (SMTP)

SMTP is a TCP/IP communication protocol for electronic mail transmission, used in sending and receiving emails [26]. Mail servers and other message transfer agents use SMTP to send and receive mail messages. SMTP servers commonly use the Transmission Control Protocol on port number 25.

User-level email clients typically use SMTP only for sending messages to a mail server for relaying, typically submit outgoing email to the mail server on port 587 or 465 as per RFC 8314. For retrieving messages standards are IMAP and POP3.

3.3.3. Nexmo

Nexmo is a Proprietary Licensed service that allows applications to integrate it and send SMS messages [27]. Nexmo SMS API allows to send and receive text messages to and from users worldwide, using REST APIs with the following features: Programmatically send and receive high volumes of SMS globally; Send SMS with low latency and high delivery rates; Receive SMS using local numbers; Scale applications with familiar web technologies; Pay only for what you use, nothing more.

3.3.4. Discussion

This subsection discusses Software as a Service (SaaS) solutions such as OneSignal; PushPro; Amazon SNS, comparing them with open source solutions (Push API for Push Notifications and Nodemailer for Email messages), and explains the main considerations why it was chosen to build our own Notification System with those libraries instead of holding on to Proprietary Services/SaaS. As for the SMS messages, it was used a SaaS (Nexmo) to do this functionality, because there is no way to deliver SMS messages without monetary costs associated.

The main reason why it was discarded proprietary solutions (for Push and Email components), it is because all follow a subscription model, were the customer pay a recurring price at regular intervals (monthly, yearly, or seasonal) to access the service [28].

The main reasons why it was discarded a SaaS are:

- **Cost:** Proprietary software services are explicitly provided under subscription model, which implies that you need to pay for the service, once you stop paying, you have nothing. Plus, many of these services have an history that in many cases the payment contract is changed, and the price increases just because they want/the product they serve increased revenue;
- **Dependence on the service provider:** Users (organizations) that use SaaS are dependent of the provider demands. Whenever the provider decides to change the rules of the contract/increase payment fees/ change features, the organizations will have to work around with those changes and comply with it;
- **Lack of control** – If a customer requires a more specific feature to its product, and the service does not provide that functionality, the costumer has no choice and has to work around with the features that the service provide. Typically, every costumer has to use the latest version of the software application and cannot defer upgrades or changes in the features;

On the other hand, the main reasons why it was decided to build qNotify with Push API are: There are web push libraries for most of the programming languages (Python, Java, PHP) including Node.js; It is easy to use and provides a single module dependency (web-push) [29]; Its license allows the use of this software in commercial applications, without monetary costs (this was one of the requirements of the solution); It provides structured documentation and a stable release version.

Regarding Nodemailer, the main reason why it was chosen to use as the module to integrate with qNotify Email component due to its easy integration process (no dependencies) and of course it is free to use in commercial applications. Nodemailer also provides all the features that email sending requires and uses the SMTP Protocol.

As for the Nexmo service, instead of other services (uSendIT, Gateway API), it is because it does not require to pay for a period of time, instead you pay for the number of messages sent. That feature allowed us to use Nexmo in a trial version to develop the SMS Component.

4. qNotify Design and Implementation

This chapter presents the solution to solve qDocs problems originally referred in Chapter 1. This solution is a notification server, called “qNotify”. qNotify is a webserver that allows qDocs platform as well as any other application, the ability to send notifications to users. Section 4.1 introduces the architecture of qNotify, the domain model is presented in Section 4.2. Section 4.3 explains details of the implementation regarding push endpoints and authorized applications synchronization. Section 4.4 presents qNotify interfaces with details of each endpoint.

4.1. qNotify Architecture

As suggested in Figure 4.1 (ArchiMate Diagram [30]), qNotify exposes a standard interface (REST API) through which any application (named as “General Application” for this explanation) communicates using standard data type (JSON).

ClientApp and ServerApp communicate through ServerApp API, and the User access ClientApp through a User Agent (browser). These components and interfaces (ClientApp, ServerApp, User Agent, ServerApp API) are presented as general to an application, to explain the interactions with qNotify. The User is not part of the architecture, for simplicity reasons.

Push service makes use of Push API to deliver push notifications to Users, as for the Email service, it makes use of Nodemailer library to send Emails, and finally, it is used Nexmo service to deliver SMS’s.

qNotify provides the following features: Subscribe for Push Notifications; Send Notification via Push / Email / SMS Channels; Consult Notifications History; View/Change Channels Permissions;

For an Application to send push notifications (using qNotify) to users, the user needs to be subscribed for push notifications. To perform a correct push subscription, the following steps may happen (as suggested in Figure 4.1): (1) ClientApp Creates the Subscription in the Push Service; (2) Push Service return the Push Subscription to ClientApp; (3) ClientApp Sends the Push Subscription to ServerApp; (4) ServerApp performs a SaveSubscription request to qNotify, and qNotify saves the Push Subscription in the database.

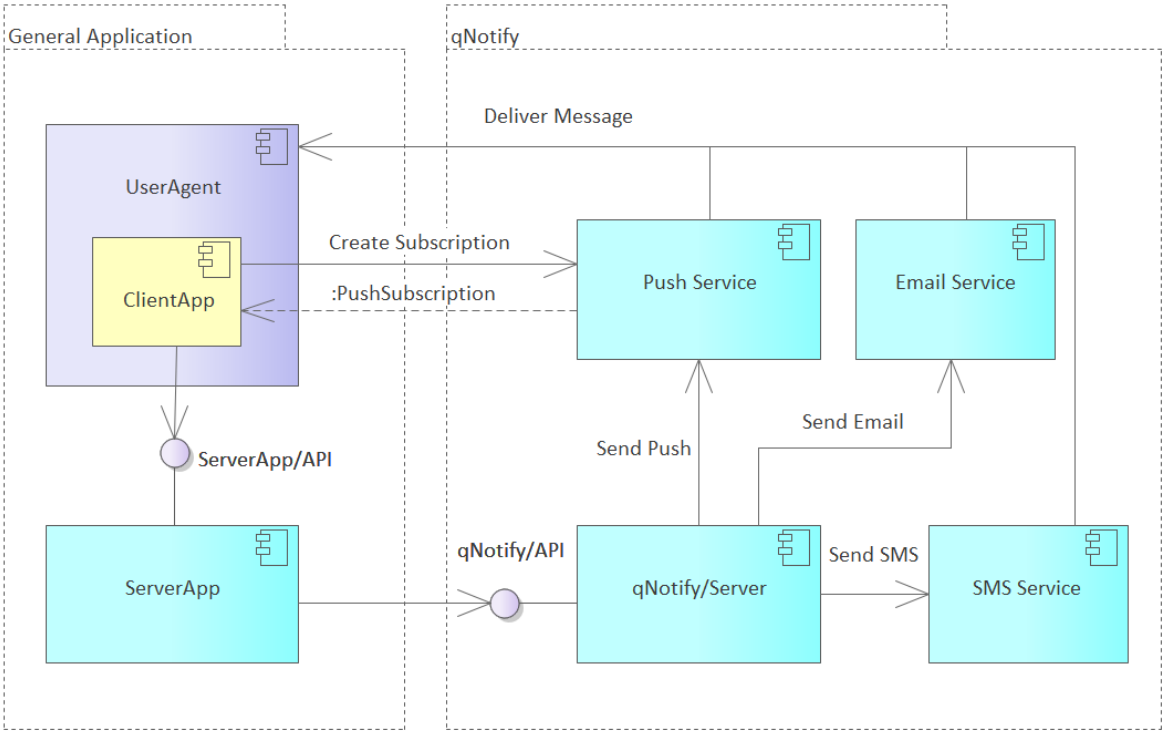

Figure 4.1. qNotify General Architecture (ArchiMate diagram).

After the user is subscribed for push notifications, the Application can send Push, Email and/or SMS notifications (using qNotify), the following steps explain this process as suggested in Figure 4.2: (1) ServerApp performs a SendNotification request to qNotify/Server, providing the User, the Message and the Channels list from which the notification will be sent (Push, Email, SMS); (2) qNotify obtains User Permissions list from database and checks if they match with the provided (from ServerApp) channels list; (3) qNotify sends the Message to respective delivering services (Push, Email, SMS); (4) Push, Email, and SMS services deliver the notification to all active User Agents; (5) qNotify saves the notification sent in the database.

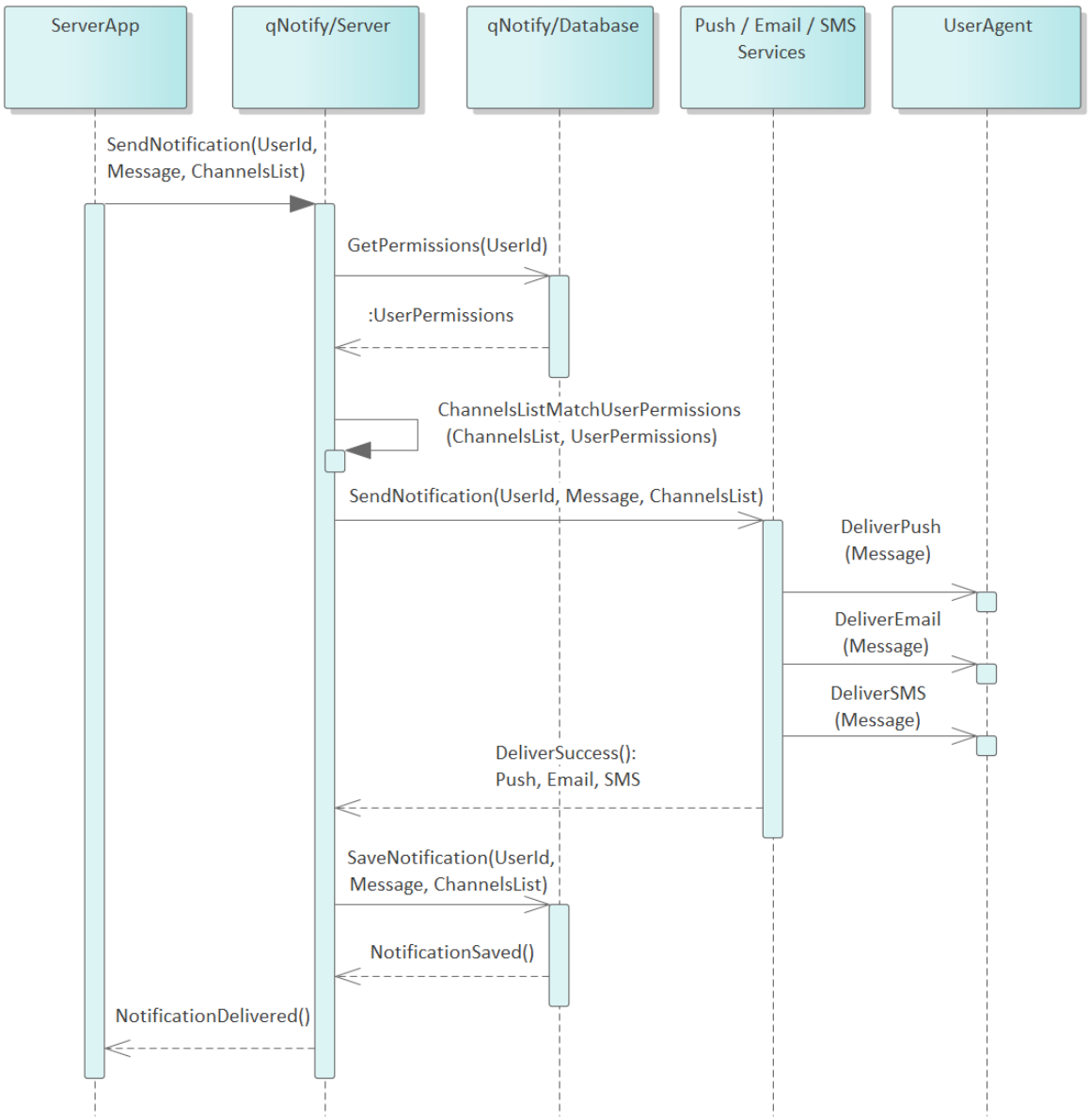

Figure 4.2. SendNotification From Push, Email and SMS Channels Workflow.

Users can consult their notifications history as suggested in Figure 4.3: (1) User selects the option in ClientApp to consult notifications history; (2) ClientApp requests ServerApp to obtain user notifications history; (3) ServerApp performs a GetNotifications request, providing the UserId; (4) qNotify returns the NotificationsList sent to that User; (5) ClientApp displays the Notifications History to User.

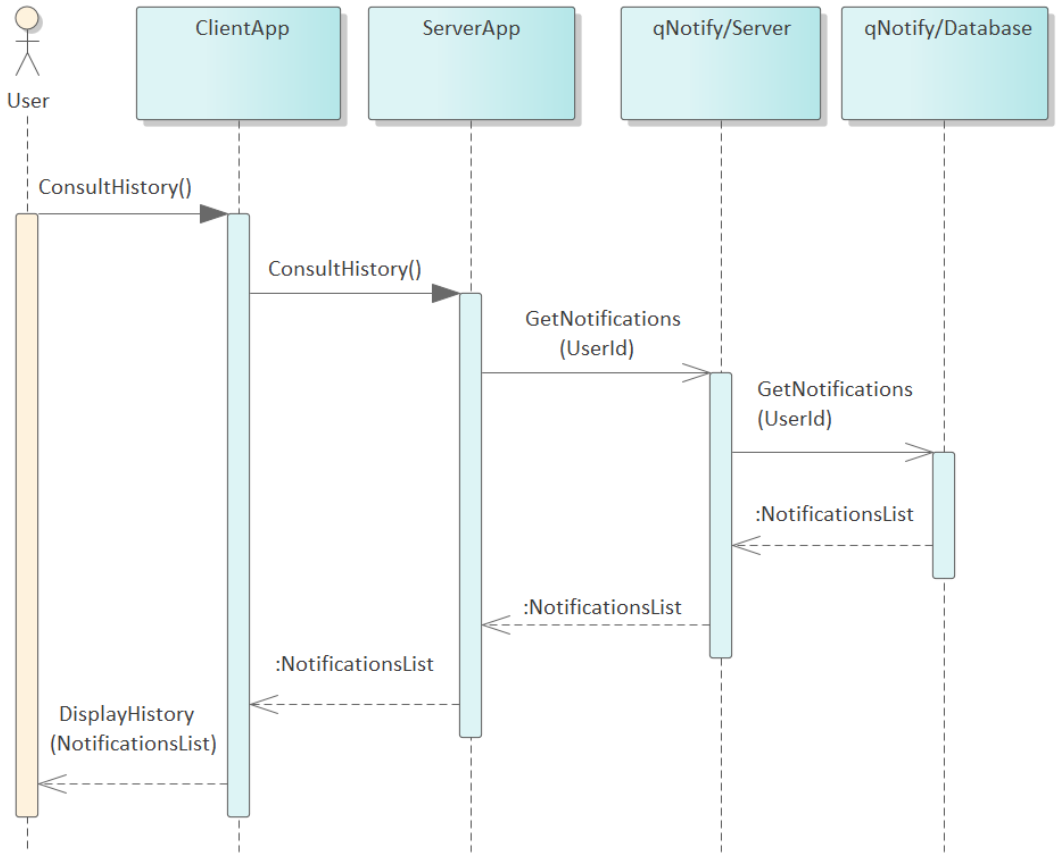

Figure 4.3. Notifications History Workflow.

qNotify allows users to receive notifications through Push, Email and SMS channels. Users can define their permission channels, as suggested in Figure 4.4: (1) User chooses which channels allow/deny to receive notifications; (2) ClientApp sends permissions choice to ServerApp; (3) ServerApp performs a SavePermissions request to qNotify, providing the UserId and the Channels choice; (4) qNotify updates user permissions for the provided UserId in the database.

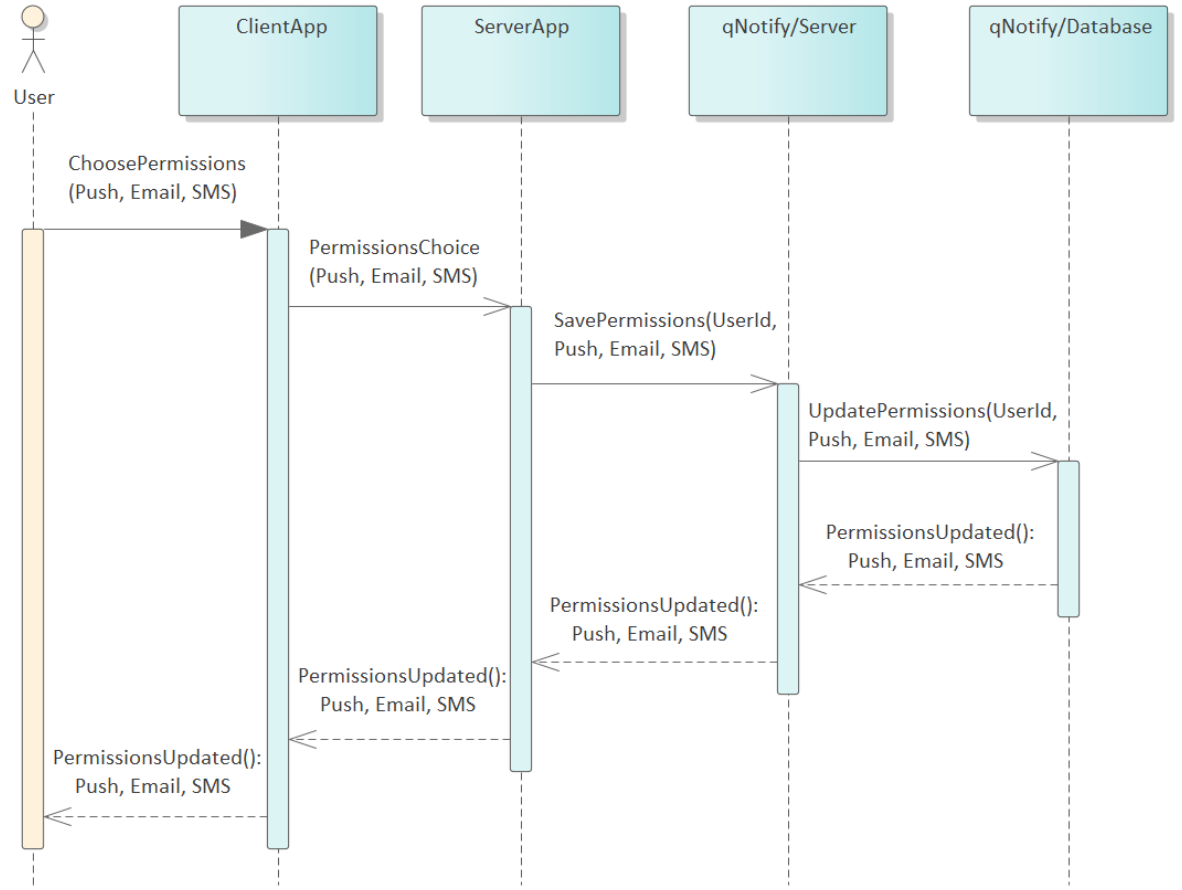

Figure 4.4. Save Permissions Workflow.

Finally, for an application to perform requests to qNotify, first needs to be registered in the system (otherwise, anyone would be able to perform requests to qNotify). As suggested in Figure 4.5 for an Application (General Application) to be registered in the system, the following steps may happen: (1) Application Operator performs a RegisterApp request to qNotify, providing the application name and secret; (2) qNotify computes a Key Derivation Function (KDF) to the secret (further explained in security section) and then stores the application name and secretHash in the database; (3) After registration success, qNotify connects to a new database through the registered application name, in order to handle requests coming from that application.

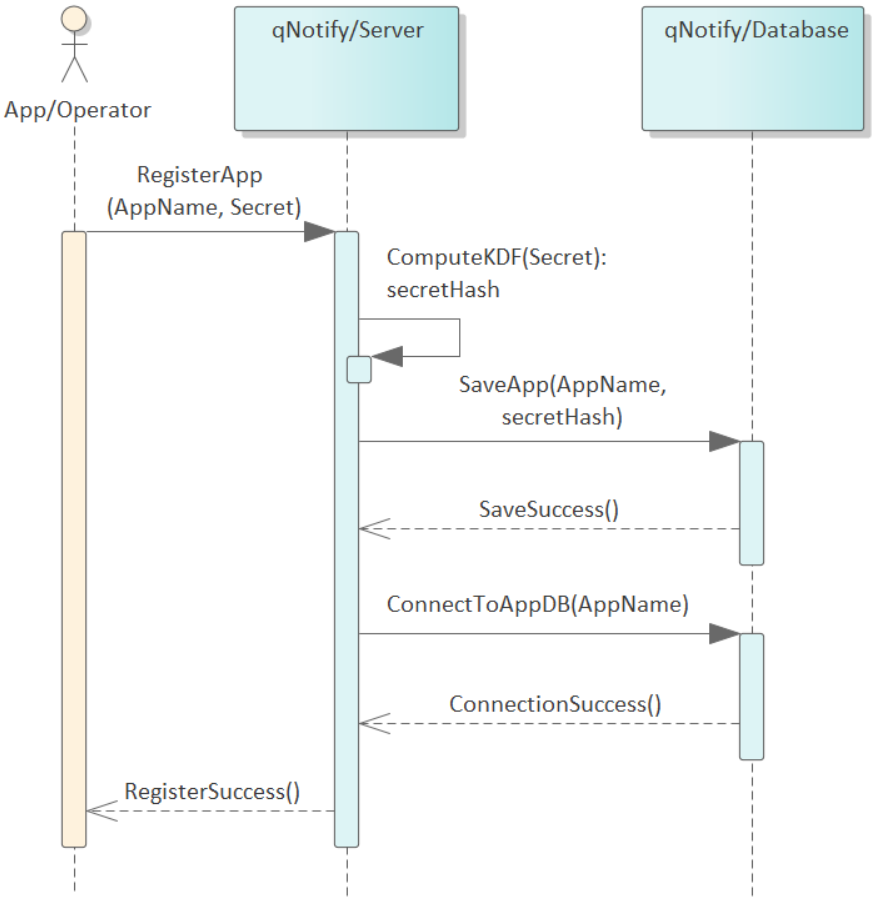

Figure 4.5. Application Registration Workflow.

4.2. qNotify Domain Model

As suggested in Figure 4.6, regarding qNotify domain model, the AuthorizedApp class represents any Application that is registered in the system and consequently authorized to use qNotify services (as explained in Figure 4.5). As for the User class, it represents any User of that Application.

Users can be subscribed to receive push notifications from the application in multiple devices, reason why one User can have multiple PushSubscriptions. PushSubscription contains the following attributes: Push Endpoint (User Agent Address/URL for Push Notifications); VapidKeys (Public and Private Authentication Keys, for the Application to be able to deliver Push Notification to UserAgent Address/URL; and ExpirationDate (date when the push subscription expires).

Users can define their Permissions to receive Notifications from Push, Email and/or SMS channels, by setting the respective Boolean attributes to True (allow) or False (deny): AllowPush; AllowEmail; AllowSMS.

The Authorized Application can only deliver Notifications through allowed channels defined by User. Notifications contain the following attributes: DeliverChannels (channels from which the notification

was sent); DeliverDate (date when the notification was sent); Unseen (boolean attribute to inform Users if they have open/seen the notification or not); and Content (Subject; Message; Push URL: link to which the user is redirected when opens the notification pop-up; and PushIcon: Icon displayed in the Push Notification). Users have also access to their Notifications History.

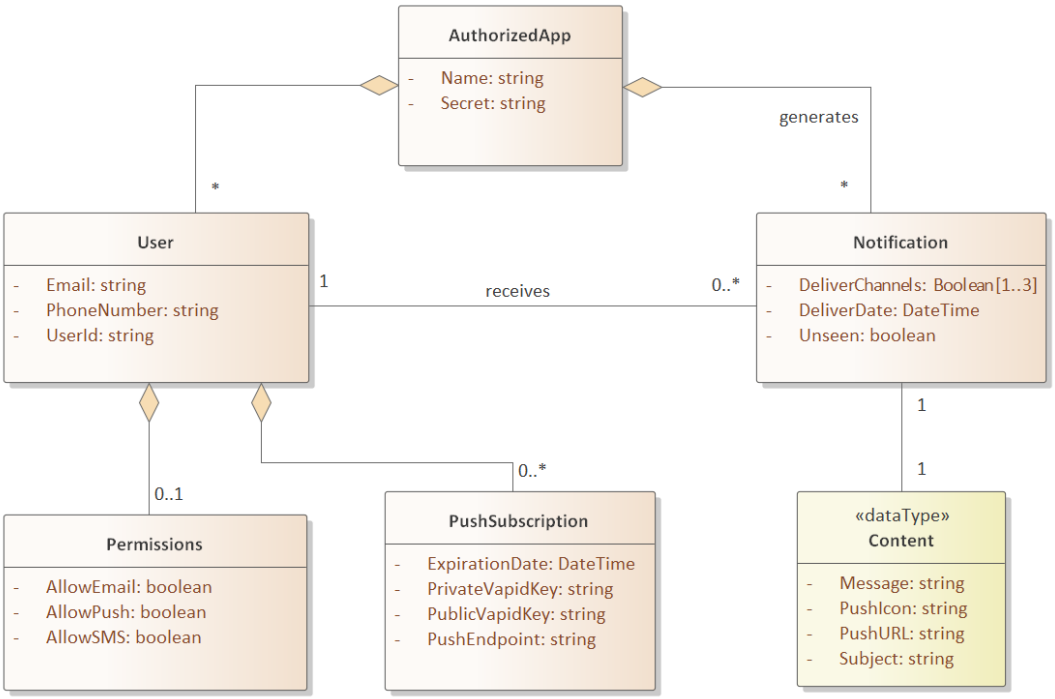

Figure 4.6. qNotify Domain Model.

4.2.1. qNotify Data Models

This subsection presents qNotify models that compose the data collections of the system. As suggested in Figure 4.7, qNotify stores for each Authorized Application the data in separated databases: each application registered in the system has its own database, and any data operation (save / remove / update / get) is made specifically to that database (Notifications, Permissions, PushSubscriptions collections) of that application. The names of the databases (of allowed applications to interoperate with qNotify) are the application “name” provided during the registration process and are stored in AuthorizedApp collection.

PushSubscriptions collection stores user push subscriptions, every push subscription contains the following attributes: UserId; Endpoint; VapidKeys (Public; Private); ExpirationDate. This collection allows an application to check if user: Is Subscribed to Push Notifications; Is Subscribed in other devices than the current from which is accessing to; Is not Subscribed at all.

Permissions collection allows the application to save/update User Permissions and retrieve them, respectively. Each permission contains the following attributes: UserId; AllowPush; AllowEmail;

AllowSMS. This structure allows: To Send Notifications to Users only through allowed channels; and Users to View/Change their channel permissions.

Notifications collection contains Push/Email/SMS notifications sent to users; this model contains the following attributes: User (UserId; Email; and PhoneNumber); DeliverChannels; Content (Subject; Message; PushURL; PushIcon); Unseen; DeliverDate.

Notifications model allows: Users to access their Notifications History with detail, including the Subject, Message, PushURL, the Channel(s) from which were sent, and the date of the event (DeliverDate). PushURL contains the link to which the user is redirected when opens the notification pop-up (Get Notifications Component); Users to see the number of notifications that he does not seen/open yet (Unseen attribute).

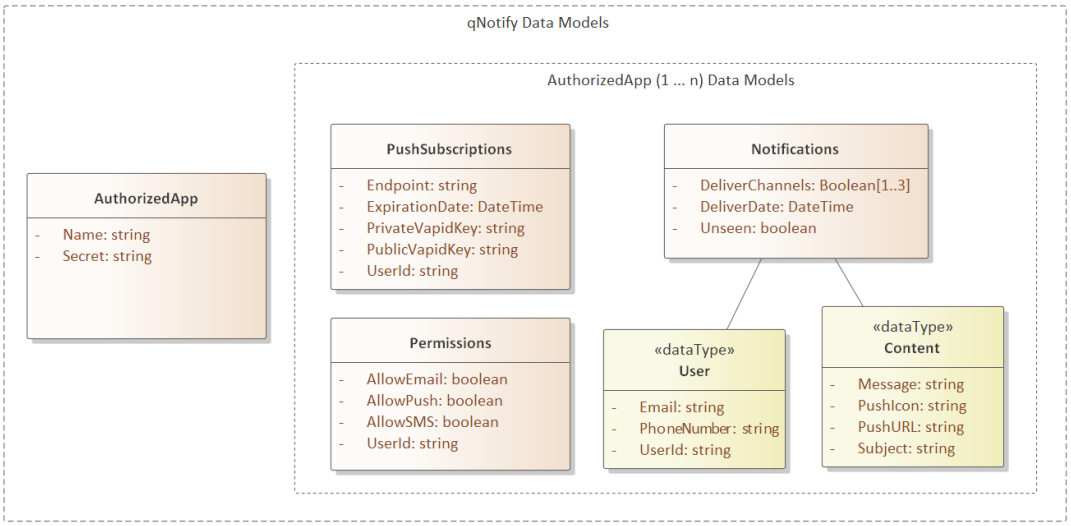

Figure 4.7. qNotify Data Models.

4.3. Implementation Details

This section explains the process used to synchronize applications that are allowed to perform requests on qNotify, and, how qNotify manages all push subscription endpoints specific of each user and how synchronization process is done, to get rid of outdated endpoints and coexist with multiple up to date ones.

4.3.1. Authorized Applications Synchronization

As explained in Section 4.2, qNotify has its own database, to store the applications that are registered in the system, so that it can manage all applications that currently are allowed to make requests. When a request is made, qNotify can handle it properly according to the information presented in the database for that application, for instance: ServerApp (qDocs) performs a “GetNotifications” request

providing the UserId of User “B”; qNotify returns all notifications of User “B” only from ServerApp (qDocs) database.

The applications registered in the system are stored in AuthorizedApp Collection (through AuthorizedApp Model, see Figure 4.7), which contains the “name” and “secret” of the application that is authorized to make requests, to prove its identity and authenticity.

For qNotify to keep track of all applications correctly, before and during runtime, the following conditions may happen:

- Every time qNotify starts, it connects to qNotify database only, after that it is waiting for requests;
- When a request arrives, it is intercepted by the validation function (this function is explained in 6.2), and checks if the provided “name” is registered in the system, and if it is, connects to the database of that application, by the provided “name”;
- If an application registers during execution, qNotify will be able to connect to its database also when that application makes the first request;
- If qNotify already contains a connection to the database of the requester, it is used that connection instead of a new one;
- Every time a request is made all data saved in the database according to that request is made in the specific database of the application that made the request.

With the steps previously described, qNotify is able to connect to an applications database at any moment, and every request that is processed is treated and saved in the database specific of that application.

4.3.2. User Push Endpoints Synchronization

This subsection present three problems regarding User Push Endpoints, that emerged during qNotify development: (1) How many Push Endpoints can a User have?; (2) What if a User authenticates in a friend’s device?; (3) What if the Service Worker instance disappears from the User Agent?. For each of these statements it is presented the approach to the solution, and then the algorithms with the strategies to solve it.

How many Push Endpoints can a User have?

When considering sending an Email or SMS message, the logic is simple: the endpoint is: (1) for email the user’s email address; and (2) for SMS, the phone number. On the other hand, when delivering a push notification, the endpoint is not direct. For instance, if a user allows to receive push notifications

on his smartphone app, and on his tablet app, both the subscriptions have different endpoints; and when the push is sent it is expected that the user receives the push at all his devices.

What if a User authenticates in a friend's device?

If user 'A' authenticates in application and the user agent does not contain a service worker instance, the service worker is registered, and it is saved in the database along with the user 'A' ID associated with that specific subscription (endpoint).

If user 'A' authenticates in the application and the user agent already contains a service worker instance, the program checks if user 'A' that is accessing the application is the same user associated to that service worker instance, if true, nothing happens, if not, user 'A' is added to that service worker instance (it is saved in the database a new subscription to the same endpoint, but with user 'A' ID associated). With this approach, if qDocs sends a push to user 'A', all User Agents where user 'A' permitted to receive push notifications will pop the notification. Making the user receive notifications in his multiple devices (Smartphone, PC, Tablet, etc.).

What if the Service Worker instance disappears from the User Agent?

If user 'A' deletes the service worker instance of his user agent, the subscription is still saved with that user 'A' ID in the database, but when the push is required, it won't pop up on the user agent. In that case, when the push is sent, if it fails, the program will delete from the database the subscription, because it is an outdated subscription endpoint, making all the subscriptions saved in the database synchronized and up to date.

4.3.3. Push Endpoints Algorithms

This section presents two algorithms, Subscription Control and Send Push. Both algorithms allow us to keep track of Users Push Endpoints Synchronized in the database.

Algorithm 1. Subscription Control Algorithm.

```
begin
    swReg ⟵ null
    subscription ⟵ null
    userAlreadySub ⟵ false
    if ServiceWorker and PushManager then
        swReg ⟵ navigator.serviceWorker.register()
        subscription ⟵ swReg.pushManager.getSubscription()
        userAlreadySub ⟵ getSubscriptionDB(subscription)
        if userAlreadySub = false or subscription = null then
            subscription ⟵ swReg.pushManager.subscribe()
            saveSubscriptionDB(subscription)
            User Subscription Saved.
        else
            User Was Already Subscribed.
    else
        Push Not Supported.
```

Subscription Control algorithm only saves a subscription to the User if:

- There is not a Service Worker Registered in the current User Agent.
- There is a Service Worker Registered in current User Agent, but does not match to the Current User Accessing the System.

Algorithm 2. Send Push Algorithm.

```
begin
    subscriptions ⟵ findSubscriptions(req.body.user)
    foreach sub ∈ subscriptions do
        res ⟵ webPush.sendNotification(sub, req.body.content)
        if res then
            Notification Sent.
        else
            removeSubscription(sub)
            Outdated Subscription Removed.
```

Send Push algorithm allows us to keep subscriptions synchronized in the database. The algorithm first, gets all Subscriptions from the database that matches the provided User, then for each Subscription (each device where the user accepted to receive notifications) sends the Push. If the Push fails, the current Outdated Subscription is removed from the database. With these two algorithms, all Subscriptions are Synchronized and up to date in the system.

4.4. qNotify API

qNotify exposes API's in two groups: qNotify/Public API Endpoints; and qNotify/Protected API Endpoints. For each group, it is described the endpoints and respective functionality along with the required input. This section is intended to describe the qNotify API succinctly, a complete API description is presented in appendix C.

4.4.1. qNotify/Public API Endpoints

As suggested in Table 4.1, qNotify/Public API handles requests coming from any application (ServerApp) that wishes to use qNotify services. First the ServerApp must be registered in the system, and only after registration successfully done, it can perform requests to qNotify protected endpoints.

RegisterApp saves into qNotify database (in AuthorizedApp Collection) two attributes, the application name and secret. The "name" must not be registered yet in the system, otherwise the registration is rejected. The authorization process is further explained in security section 6.2.

Table 4.1. qNotify/Public API Endpoints.

qNotify/Public API	
Name	Description
RegisterApp (Name, Secret)	Store provided Application Name and Secret in Database

4.4.2. qNotify/Protected API Endpoints

Notify/Protected API handles requests coming from any ServerApp previously registered in the system. qNotify/Protected interface provides the following endpoints, described in Table 4.2: SaveSubscription; Unsubscribe; GetSubscription; SendNotification; GetNotifications; SetUnseen; SavePermissions; GetPermissions. For each protected endpoint, it is imperative to provide the "Name" and "Secret" of the requester application, in the Headers of the request. Reason why in the above table these attributes are specified as "_Name" and "_Secret" because they are passed in headers instead of body of request.

Table 4.2. qNotify/Protected API Endpoints.

qNotify/Protected API	
Name	Description
SaveSubscription (UserId, Subscription, _Name, _Secret)	Subscribes User for the provided UserId
Unsubscribe (UserId, _Name, _Secret)	Removes User Subscription for the provided UserId
GetSubscription (UserId, Subscription, _Name, _Secret)	Returns Subscription for the provided UserId
SendNotification (User, Content, Channels, _Name, _Secret)	Send Notification from provided channels to User with provided UserId, Email, PhoneNumber
GetNotifications (UserId, _Name, _Secret)	Returns Notifications List History for the provided UserId
SetUnseen (NotificationId, _Name, _Secret)	Updates Notification Unseen Attribute for the provided Notification ObjectId
SavePermissions (UserId, Channels, _Name, _Secret)	Save Channels Permissions for the provided UserId
GetPermissions (UserId, _Name, _Secret)	Returns Channels Permissions for the provided UserId

Endpoints regarding to subscriptions are:

- **SaveSubscription:** Save Subscription endpoint excepts a service worker registration (Endpoint, VAPID Keys, ExpirationDate) and a UserId, and saves it along with the UserId in the database. This process was explained in detail in section 4.3;
- **Unsubscribe:** Unsubscribe endpoint receives from input the UserId and deletes from the database all the Subscriptions Objects the contains the same UserId as the provided one;
- **GetSubscription:** Get Subscription endpoint receives from input the UserId and the current service worker registration in the current user agent (from which the User is accessing), and returns true if there is at least one match of the subscriptions saved in the database. This endpoint must be called before the Subscribe endpoint, and only if the Get Subscription returned value is false, the user is subscribed (this workflow if explained in detail in 4.3).

The following endpoints are related with notifications:

- **SendNotification:** This endpoint receives from input the UserId, Email, PhoneNumber, Subject, Message, URL, and Channels list. UserId contains the address endpoint to push channel (UserId => Subscription Endpoint), the Channels list contains the channels from which the Notification will be sent. Send Notification endpoint delegates the sending process to the corresponding channel services (Push / Email / SMS) and after sending resolution, saves in the database the Notification along with the channels that successfully performed the sending process;
- **GetNotifications:** This endpoint expects from input the UserId, and retrieves from the database the list of notifications sent to the provided UserId. Section 5.2.3 provides use cases of user consulting notification history in qDocs context;
- **SetUnseen:** This endpoint receives from input the Notification ObjectId and sets the Unseen attribute to false. When a notification is stored the Unseen attribute is set to true. Unseen attribute

allows to display to the User how many notifications he has not seen/open yet (section 5.2.3 shows this use case with details).

As for channels permissions, there are two endpoints:

- **SavePermissions:** Save Permissions endpoint updates the channels permissions in the database for the provided UserId;
- **GetPermissions:** Get Permissions endpoint returns the permissions channels list (Push, Email, and SMS) for the provided UserId. Section 5.2.2 provides save permissions and get permissions use cases in qDocs context, with the User consulting and changing permissions.

Chapter 5, demonstrates qNotify integration with qDocs, providing the architecture of the system and implemented use cases in qDocs context.

5. qNotify Integration with qDocs

This chapter presents qNotify integration with qDocs platform. Section 5.1 presents qDocs architecture with qNotify integration, section 5.2 describes the functionalities and use cases implemented in qDocs context with qNotify integration.

5.1. qNotify – qDocs Architecture

This section presents the architecture of qDocs Platform after the integration with qNotify. As suggested in Figure 5.1, qDocs communicate with qNotify via qNotify/API interface (qNotify/Public and qNotify/Protected APIs as explained in Section 4.4).

qDocs Platform, as explained in Chapter 2, contains three types of users: Citizen, Curator, and Operator. Any user accesses qDocs from a User Agent (example: browser). Citizens, Curators and Operators are not presented in the architecture for simplicity reasons. In qNotify Integration with qDocs, the CitizenApp, AdminApp and CuratorApp take place where ClientApp was in the explanation of qNotify with a General Application (Figure 4.1), as for the qDocs Platform, takes place where the ServerApp was.

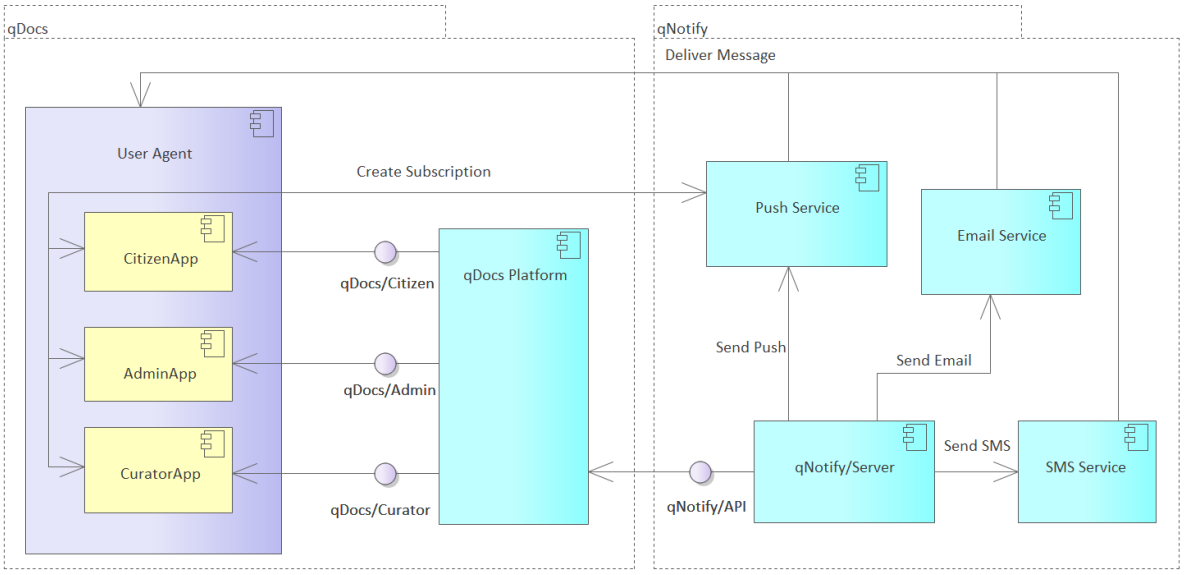

Figure 5.1. qNotify Architecture Integrated with qDocs (ArchiMate diagram).

For qDocs Platform to send push notifications (using qNotify) to users (Citizens, Curators, Operators), the user needs to be subscribed for push notifications. To perform a correct push subscription, the

following steps may happen: (1) ClientApp (CitizenApp, CuratorApp, AdminApp) Creates the Subscription in the Push Service; (2) Push Service returns the Push Subscription to ClientApp; (3) ClientApp Sends the Push Subscription to qDocs Platform; (4) qDocs Platform performs a “SaveSubscription” request to qNotify, and qNotify saves the Push Subscription in the database.

After the User (Citizen, Curator, Operator) is subscribed for push notifications, qDocs can send Push, Email and/or SMS notifications (using qNotify), the following steps explain this process: (1) qDocs Platform performs a “SendNotification” request to qNotify/Server, providing the User, the Message and the Channels list from which the notification will be sent (Push, Email, SMS); (2) qNotify checks if the User permissions match the provided channels list and sends the Message to respective delivering services (Push, Email, SMS); (3) Push, Email, and SMS services deliver the notification to all active User Agents; (4) qNotify saves the notification sent in the database.

For Citizens to consult their notifications history, the following steps may happen: (1) Citizen selects the option in CitizenApp to consult notifications history; (2) CitizenApp requests qDocs Platform to obtain the citizen notifications history; (3) qDocs Platform performs a “GetNotifications” request, providing the CitizenId; (4) qNotify returns the NotificationsList sent to that Citizen; (5) CitizenApp displays the Notifications History to Citizen.

For Citizens to define their permission channels, the following steps may happen: (1) Citizen chooses which channels allow/deny to receive notifications; (2) CitizenApp sends permissions choice to qDocs Platform; (3) qDocs Platform performs a “SavePermissions” request to qNotify, providing the CitizenId and the Channels choice; (4) qNotify updates citizen permissions for the provided CitizenId in the database.

5.2. qNotify - qDocs Functionalities

For qNotify integration with qDocs (or other ServerApp), it is required to extend the client and server applications, to Send Notifications and to display information such as: Notifications History; View/Change Permissions. In qDocs context, it was extended the client applications (CitizenApp, CuratorApp, AdminApp) and qDocs Server to exchange data and build the logic to accomplish the following use Functionalities: Manage User Push Subscriptions; Manage User Permissions; Display Notifications History to User; Send Notifications qDocs Use Cases.

5.2.1. Manage User Push Subscriptions

The first time a User authenticates in qDocs, it is prompted with an “Allow Notifications” pop-up (Figure 5.2), following the user acceptance to receive notifications, a service worker is registered in the user agent from which the User is accessing and then the Push Subscription is saved into qNotify database.

In case another User authenticates in qDocs from a user agent that already contains a service worker registration (friend device use case explained in subsection 5.3.3), qNotify will take that same service worker and save in the database the Push Subscription with the current UserId (Figure 5.3 exposes a Push Subscription example from the database).

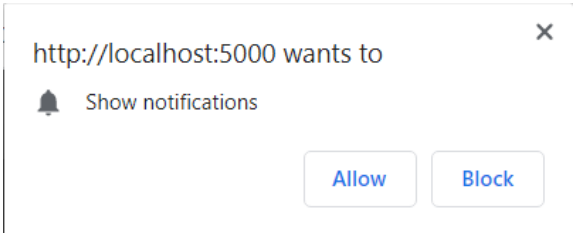

Figure 5.2. Allow Notifications Pop-up – extracted from DEV Instance.


```
_id: ObjectId("5ed124a4d4ae673a607f9465")
endpoint: "https://fcm.googleapis.com/fcm/send/dsg1Veffuc8:APA91bEtD3fcE_7VETMm8l..."
> keys: Object
UserId: "917326346"
createDate: 2020-05-29T15:05:08.338+00:00
__v: 0
```


Figure 5.3. Push Subscription Example - extracted from MongoDB.

5.2.2. Manage User Permissions

In this work, it was implemented three different channels to send notifications to users, Push, Email, and SMS. These channels require permission from the User to enable qNotify to send notifications from only permitted channels. It was established that when a User “Allow Notifications” in the pop-up event, all the three channels will be set as allowed to receive notifications. After these steps, the User can view and change his permissions under the Settings menu (Figure 5.4). To it so, the User only need to select in each channel if allow or deny notifications, and finally select Save Permissions.

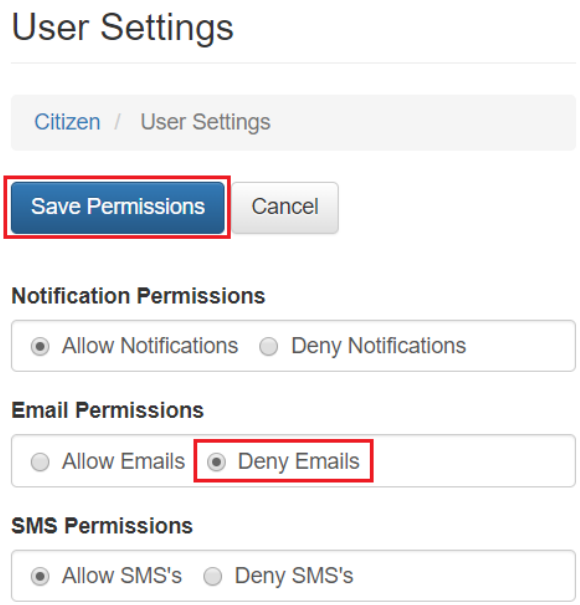

Figure 5.4. View/Change Permissions from Settings Menu.

5.2.3. Display Notifications History to User

Users can view their notifications history in two different ways, from the Messages Menu panel or the Notifications Bell Icon. In the Messages Menu panel, the User has access to all notifications in a scrollable list, each notification displays the channel(s) from which the notification was sent, creation date, the subject, and the message content (Figure 5.5). Push Notifications also contain an URL attribute, specifically defined to each type of notification, to redirect the user to the specific Document/Curator/Navigation Menu of qDocs.

The User can also consult his notifications history immediately from the Notifications Bell Icon. The bell icon also contains the number of Notifications that the User has not seen/open yet, and when the user selects it, all unseen notifications have a slightly darker background color, so that the user be aware of which he didn't seen/open (these features are illustrated in Figure 5.6, with an example of two Unseen notifications).

Users can also view their notifications when the notification pops-up (in case of push) and from the Email and SMS inbox's.

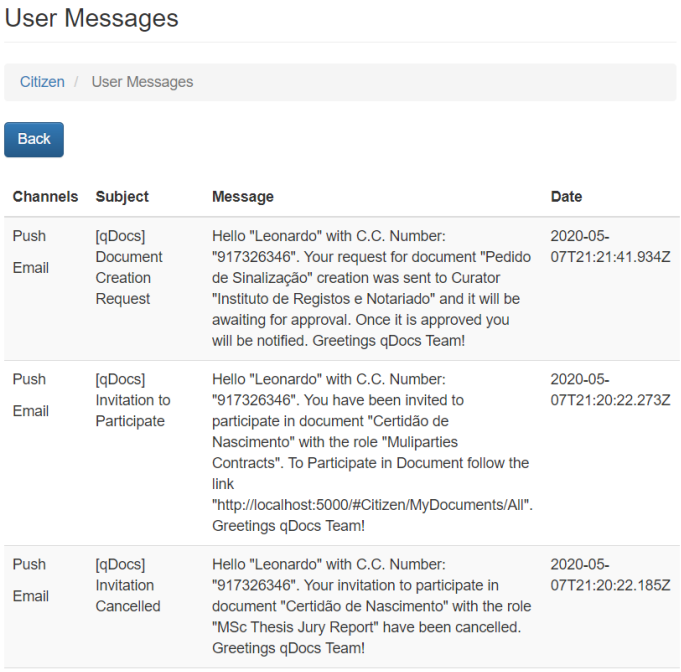

Figure 5.5. Notifications History from Messages Menu Panel.

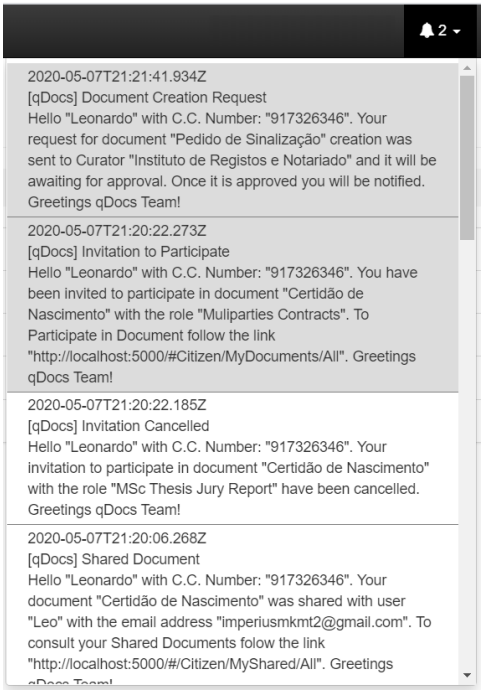

Figure 5.6. Notifications History from Bell Icon with an example of two Unseen Feature.

5.2.4. Send Notifications Use Cases

In qDocs context it was implemented six “SendNotification” use cases: (1) User Association to Curator with Specific Roles; (2) User Disassociation from Curator; (3) Shared Document Accepted; (4)

Invitation to Participate in Document with Specific Roles; (5) Invitation to Participate in Document Cancelled; and (6) Document Request. Next list explains which information is passed in each use case and how the event is triggered.

(1) User Association to Curator with Specific Roles

In this use case, the Citizen is notified when: An Admin associates a Citizen to any Curator of the system; A Curator associates a Citizen to himself (to his Organization).

The notification sent to the Citizen includes the following information: Citizen Name; Citizen CC Number; Curator Name; Roles the Citizen was added in that specific Curator; URL to Curators Menu List.

(2) User Disassociation from Curator

In this use case, the Citizen is notified when: An Admin disassociates a Citizen from any curator of the system; A Curator dissociates a Citizen from himself (from his Organization).

The notification sent to the Citizen includes the following information: Citizen Name; Citizen CC Number; Curator Name; URL to Curators Menu List.

(3) Shared Document Accepted

In this use case, the Citizen is notified when adds a Shared Document (via URL or Hyperlink) in the documents section. The notification sent contains the following information: Citizen who Shared the Document Name; Document Name; URL to access the Shared Document.

(4) Invitation to Participate in Document with Specific Roles

In this use case, the Citizen is notified when he is invited to participate in a document with a specific role for that document. The notification sent contains the following information: Document Name; Participation Role; URL to participate in the specific document.

(5) Invitation to Participate in Document Cancelled

In this use case, the Citizen is notified when the owner of a document or an admin cancels the invitation to participate in that document for that Citizen. The notification contains the following information: Document Name; Participation Role.

(6) Document Request

In this use case, the Citizen is notified when requests to a Curator for a specific document of that Curator. The notification contains the following information: Curator Name; Document Name; URL for document request.

5.2.5. Template Messages Construction

As already mentioned before, qNotify was developed as an isolated webserver to allow not only qDocs, but to other applications, to take advantage of informing their users either with push notifications, emails and/or SMS. To make this goal possible, the approach chosen regarding messages construction, is to generate messages from the requester side (qDocs), so that qNotify only receives the User (UserId, Email, PhoneNumber), the content of the message (Title, Message, URL), and channels list from which will send the notification.

The mechanism to generate these messages with sensitive data specific to Citizens/Curators/Documents is using the ES6 String Interpolation approach [31]. Each notification was implemented in both qDocs supported languages, Portuguese and English. Figure 5.7 and Figure 5.8 show an example of a “Disassociation Notice” in both languages.

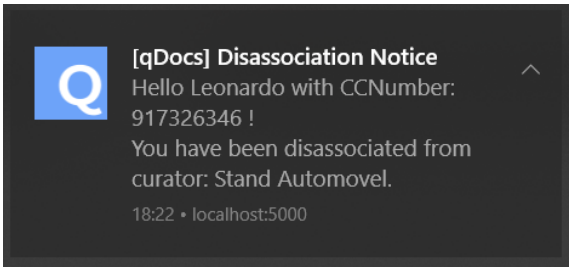

Figure 5.7. Disassociation Notice EN – extracted from DEV Instance.

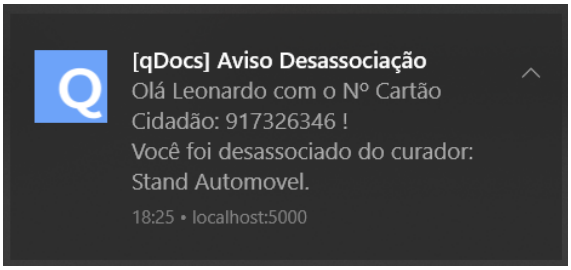

Figure 5.8. Disassociation Notice PT – extracted from DEV Instance.

5.3. qNotify Integration Conclusion

qNotify enables qDocs to send notifications to citizens, through three different channels to deliver messages (Push, Email, SMS). qDocs is now able to notify citizens in six use cases, previously presented in 5.2.4, namely: User Association to Curator with Specific Roles; User Disassociation from Curator; Shared Document Accepted; Invitation to Participate in Document with Specific Roles; Invitation to Participate in Document Cancelled; and Document Request.

Furthermore, qDocs citizens can define which channels they want to receive such notifications, again via Push, Email and SMS channels (as seen in 5.2.2). qDocs is also able to show notifications history to citizens from two perspectives, from the Messages menu panel and from the Notifications icon (as seen in 5.2.3). Both ways display the notifications history to the user, with the difference that the second alternative (Notifications icon) provide the feature of unseen and unopen notifications number displayed near notifications icon (as seen in 5.2.3).

6. qNotify Evaluation

This chapter presents and discusses the evaluation of the solution considering the following software quality attributes: Performance, Security, and Interoperability.

6.1. Performance

This section starts by introducing performance metrics taking in consideration in this performance evaluation: Throughput; Response Time; and Application Performance Index (Apdex). Subsections 6.1.1 and 6.1.2 presents four performance scenarios designed according to qNotify performance requirements, then it is described the environment setup conditions where the scenarios were tested and finally the discussion of the results obtained.

qNotify response measure is evaluated considering the following aspects: Throughput, Response Time, and Application Performance Index (Apdex).

Throughput

Throughput refers to how much data can be transferred from one location to another in a given amount of time [32]. Throughput for the purpose of this research is the number of messages successfully delivered per unit time between qNotify (hosted in the cloud) and a requester machine.

Response Time

After a request is sent to an application (qNotify), “response time” measures how long it takes for a response to return back from the application (qNotify). (This is also sometimes referred to as “round-trip” time [33].) Response time is important in performance testing because it represents how long a user must wait for a request to be processed by an application (qNotify).

Application Performance Index (Apdex)

Apdex is an industry standard to measure user’s satisfaction with the response time of web applications and services [34]. Apdex is a measure of response time based against a set threshold, Toleration (T), and Frustration (F). All responses handled in T or less time are satisfactory, responses handled between T and F are tolerated, any request handled in F or more time is frustrated. According to [35], an excellent score falls in 0.940-1.000, a good score ranks from 0.850-0.930, a fair score hits 0.700-0.840, and a poor one between 0.490-0.690. Any lower score is unacceptable.

6.1.1. Performance Requirements

To design scenarios to test qNotify, it is required to clarify performance requirements for each metric presented previously: Throughput; Response Measure; Apdex.

qNotify performance requirements were defined according to qDocs needs, as this is the application that for the moment will be using qNotify services. After some meetings with qNotify stakeholders and based on qDocs expectations for the future, it was established the following requirements: (1) qNotify must handle 10.000 requests in one minute during normal operation period of qDocs platform (23H-17H); (2) qNotify must handle 30.000 requests in one minute during overload operation period of qDocs platform (17H-23H); (3) qNotify must handle requests with an average response time <= 300 milliseconds during overload operation of qDocs (17H-23H); (4) qNotify must handle requests with an average response time <= 200 milliseconds during normal operations of qDocs (23H-17H).

6.1.2. Performance Scenarios

Performance scenarios address requests coming from qDocs platform, such as the following: Send Notifications; Get Notifications; Save Permissions, and Subscribe; Scenarios S1 and S3 address qNotify performance in normal mode (average number of requests), Scenarios S2 and S4 address qNotify in overload mode (maximum number of requests). Response measure address the actual metrics and performance of qNotify under each scenario. It is considered three metrics to evaluate qNotify: Throughput/unit of time; Response Time; Apdex.

First Performance Scenario (S1):

- Source: qDocs Platform
- Stimulus: Send Notifications (Push/Email/SMS/All) Requests
- Artifact: System
- Environment: Normal mode
- Response: qNotify Deliver Notifications
- Response Measure: Throughput 10.000 requests/minute with an average Response Time of 200 milliseconds and an APDEX Score >= 0.9 with Toleration Threshold = 500 milliseconds and Frustration Threshold = 1500 milliseconds

Second Performance Scenario (S2):

- Source: qDocs/Platform
- Stimulus: Send Notifications (Push/Email/SMS/All) Requests
- Artifact: System
- Environment: Overload mode
- Response: qNotify Deliver Notifications
- Response Measure: Throughput 30.000 requests/minute with an average Response Time of 300 milliseconds and an APDEX Score >= 0.8 with Toleration Threshold = 500 milliseconds and Frustration Threshold = 1500 milliseconds

Third Performance Scenario (S3):

- Source: qDocs Platform
- Stimulus: Get Notifications History / Save Permissions / Subscribe Requests
- Artifact: System
- Environment: Normal mode
- Response: qNotify, Retrieves Notifications List / Updates Permissions / Subscribes User
- Response Measure: Throughput 10.000 requests/minute with an average Response Time of 200 milliseconds and an APDEX Score >= 0.9 with Toleration Threshold = 500 milliseconds and Frustration Threshold = 1500 milliseconds

Fourth Performance Scenario (S4):

- Source: qDocs Platform
- Stimulus: Get Notifications History / Save Permissions / Subscribe Requests
- Artifact: System
- Environment: Overload mode
- Response: qNotify, Retrieves Notifications List / Updates Permissions / Subscribes User
- Response Measure: Throughput 30.000 requests/minute with an average Response Time of 300 milliseconds and an APDEX Score >= 0.8 with Toleration Threshold = 500 milliseconds and Frustration Threshold = 1500 milliseconds

6.1.3. Test Environment

To test these performance scenarios, previously presented, qNotify was deployed in the cloud. It was used Amazon Web Services (AWS) to install qNotify in an Elastic Compute Cloud (EC2) instance, running a Linux/UNIX OS. These performance tests were executed under the Apache JMeter project [36] and the results were exported both to a .csv file and a dashboard report.

To test the load of qNotify, it was designed two test plans, addressing the stimulus presented in the scenarios above. As presented in Table 6.1, the first test plan contains the following requests: SendPush; SendEmail; SendSMS; SendAll. And the second with: GetNotifications; SetPermissions; Subscribe. To find the maximum load qNotify can handle (number of requests/time unit), it is necessary to stress test the server, so that when qNotify starts responding slowly and producing errors, it means it has reached his limit. To do it so, each test plan will be exposed to a list of thread groups, as suggested in Table 6.2, with the number of threads per thread group increasing.

Table 6.1. Test Plans

Test Plan	Requests			
1	SendPush	SendEmail	SendSMS	SendAll
2	GetNotifications	SetPermissions	Subscribe	-

Each thread group contains the following parameters: Number of Threads; Ramp-up Period; Loop Count; Duration. The Number of Threads is the number of concurrent sources requesting qNotify, Ramp-up period is the interval in seconds between one thread and the next thread starts doing the job. Loop Count is the number of times each thread will be requesting qNotify, finally, the duration is the amount of time that threads will be doing job. For example, Thread Group 2 will have 10 threads starting at the same time, requesting in an infinite loop to qNotify, for a period of 60 seconds. Parameters Ramp-up, Loop Count and Duration are always the same because it is not necessary to add more time duration than 60 seconds to interpret qNotify behavior of handling requests.

Table 6.2. Thread Groups

Thread Group	N. of Threads	Ramp-up Period	Loop Count	Duration (sec)
1	1	0	Infinite	60
2	10	0	Infinite	60
3	100	0	Infinite	60
4	200	0	Infinite	60
5	500	0	Infinite	60

6.1.4. Performance Results

This section presents performance results obtained for Test Plan 1 and Test Plan 2, under Thread Group 3 and 4 Conditions. For reasons of simplicity, this section does not present tests results under Thread Groups 1, 2 and 5, those results tables are exposed in appendix.

Test Plan 1 – Thread Group 3

Test Plan 1 correspond to tests made for requests of type SendNotification, namely: SendPush; SendEmail; SendSMS; and SendAll. Under Thread Group 3 (100 Active Threads, with a starting delay of 0 seconds, requesting for a period of 60 seconds).

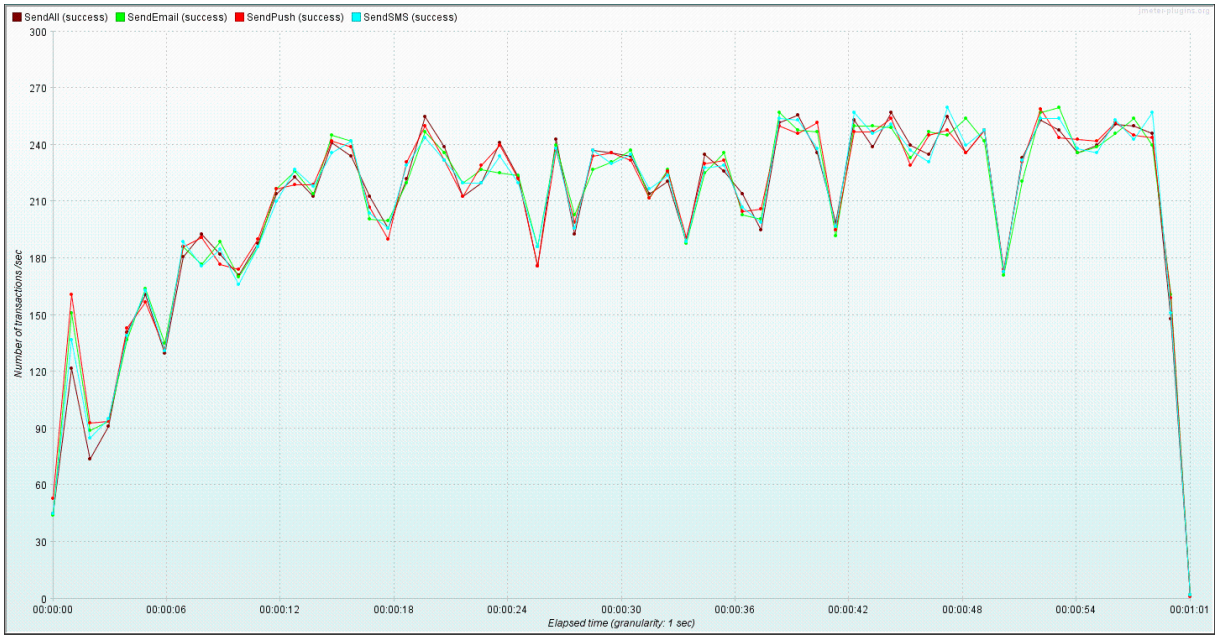

Figure 6.1. Test Plan 1 - Number of Transactions/Second with 100 Threads for 1 Minute (extracted from JMeter).

As we can see from the graph results of Figure 6.1, for a period of one minute, qNotify is able to satisfy all requests of type (SendPush / SendEmail / SendSMS / SendAll) with success, we can see that in the first 12 seconds, qNotify is increasing the number of transactions per second (possible because of all threads starting at the same time causing some delay), passing the first ¼ of time, qNotify satisfied all the requests with approximately 200 transactions per second and finally, in the last 3 seconds, it drops down the number of transactions per second until 0.

As for the response times, we can see from Figure 6.2, that qNotify in first 10 seconds struggles to serve requests, with an approximation of 300 milliseconds, due to unexpected load incoming, but in the next 50 seconds, qNotify handled all requested in an average response time of 100 milliseconds.

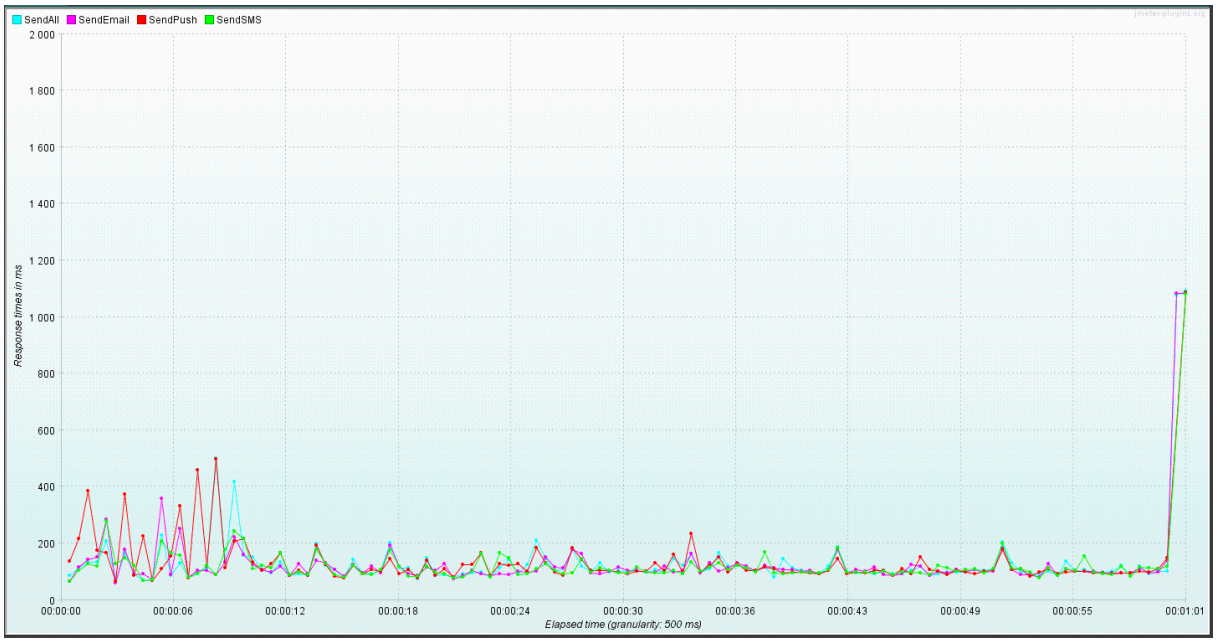

Figure 6.2. Test Plan 1 - Response Times with 100 Threads for 1 Minute (extracted from JMeter).

Although previous graphs provide information on qNotify behavior, precise values on Throughput, Response Time, and Apdex are presented in Table 6.3 and Table 6.4.

Table 6.3. Test Plan 1 - Result Statistics with 100 Threads (extracted from JMeter Dashboard).

Requests	Executions			Response Times (ms)						Throughput
Label	#Samples	KO	Error %	Average	Min	Max	90th pct	95th pct	99th pct	Transactions/s
Total	**51717**	**0**	**0.00%**	**116.14**	**37**	**7269**	**119.00**	**184.95**	**396.00**	**846.29**
SendAll	12888	0	0.00%	118.02	39	7269	130.00	194.00	687.77	211.75
SendEmail	12944	0	0.00%	111.77	37	7208	133.00	192.00	608.30	212.59
SendPush	12967	0	0.00%	122.80	38	7267	137.00	197.00	1147.32	212.22
SendSMS	12918	0	0.00%	111.97	38	7214	131.00	195.00	495.62	212.03

Table 6.4. Test Plan 1 – APDEX with 100 Threads (extracted from JMeter Dashboard).

Apdex	T (Toleration threshold)	F (Frustration threshold)	Label
0.993	**500 ms**	**1 sec 500 ms**	**Total**
0.992	500 ms	1 sec 500 ms	SendPush
0.993	500 ms	1 sec 500 ms	SendAll
0.994	500 ms	1 sec 500 ms	SendSMS
0.994	500 ms	1 sec 500 ms	SendEmail

As shown in Statistics table (Table 6.3), qNotify is able to handle up to 51.717 requests in 1 minute, without any error, with an average response time of 116.14 milliseconds and a Throughput of 846.29 Transactions per Second. As for the Apdex metric (exposed in Table 6.4), qNotify scored an excellent result of 0.993 (T = 500 and F = 1500).

These results prove that qNotify accomplished performance requirements in terms of Throughput, Response Time, and consequently Apdex score for scenarios S1 and S2.

Test Plan 1 – Thread Group 4

As in previous test, Test Plan 1 correspond to tests made for requests of type SendNotification, namely: SendPush; SendEmail; SendSMS; and SendAll. But this time tests were made under Thread Group 4 (200 Active Threads, with a starting delay of 0 seconds, requesting for a period of 60 seconds).

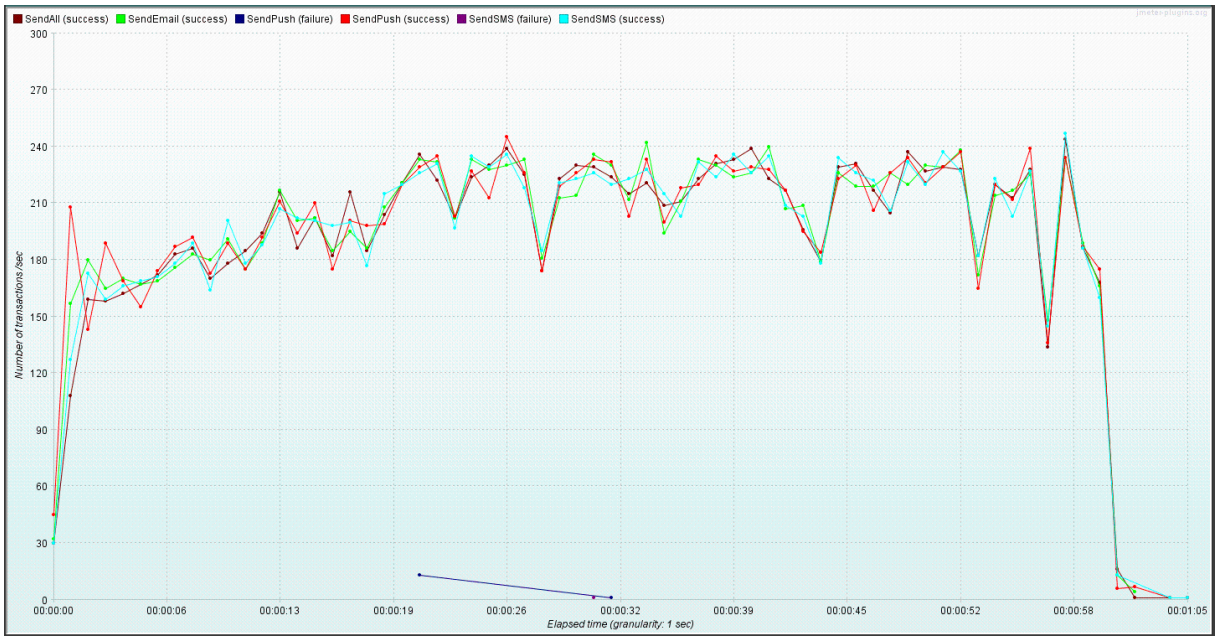

Figure 6.3. Test Plan 1 - Number of Transactions/Second with 200 Threads for 1 Minute (extracted from JMeter).

Comparing graph results presented in Figure 6.3 and Figure 6.4, (under Thread Group 4), with graph results from Figure 6.1 and Figure 6.2 (under Thread Group 3), we conclude that qNotify lowered the number of transactions per second and worse, with this amount of requests incoming, qNotify was not able to satisfy all with success, leading to failures of some SendPush and SendSMS requests.

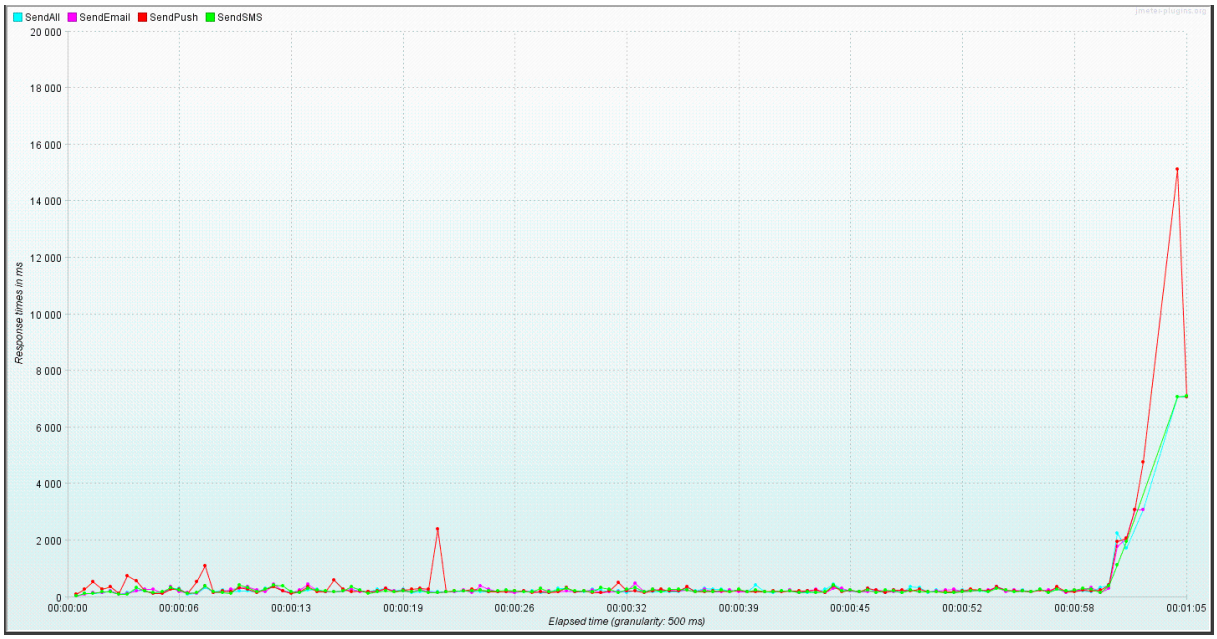

Figure 6.4. Test Plan 1 - Response Times with 200 Threads for 1 Minute (extracted from JMeter).

As we can conclude from Statistics table (Table 6.5), with 200 threads (Thread Group 4), the number of handled requests lowered to 49.637, the average response time 243.43 milliseconds, two times more than the previous test (with 100 threads), and worse, now qNotify produced 15 errors, meaning that 15 requests did not get a response at all. These results meet with the conclusion that qNotify reached is limit, and the maximum number of requests it can resolve is ~50K/minute without causing discomfort and with a 100% rate of request-response. Test results of Test Plan 1 with remaining Thread Groups (1, 2 and 5) are exposed in appendix.

Table 6.5. Test Plan 1 - Result Statistics with 200 Threads (extracted from JMeter Dashboard).

Requests	Executions			Response Times (ms)						Throughput
Label	#Samples	KO	Error %	Average	Min	Max	90th pct	95th pct	99th pct	Transactions/s
Total	**49637**	**15**	**0.03%**	**243.43**	**38**	**21039**	**416.00**	**600.00**	**1812.83**	**769.20**
SendAll	12336	0	0.00%	231.14	38	15671	405.00	563.00	1570.89	191.83
SendEmail	12433	0	0.00%	232.32	40	15210	403.00	546.30	1644.64	200.07
SendPush	12486	14	0.11%	279.76	39	21039	420.00	684.65	3234.26	193.52
SendSMS	12382	1	0.01%	230.19	39	21036	406.00	563.85	1637.17	192.34

Test Plan 2 – Thread Group 3

Test Plan 2 corresponds to tests made for requests of type GetNotifications, SetPermissions (Equivalent to SavePermissions), and Subscribe (Equivalent to SaveSubscription). These tests were performed under Thread Group 3 conditions to find if qNotify accomplishes performance requirements for scenarios S3 and S4.

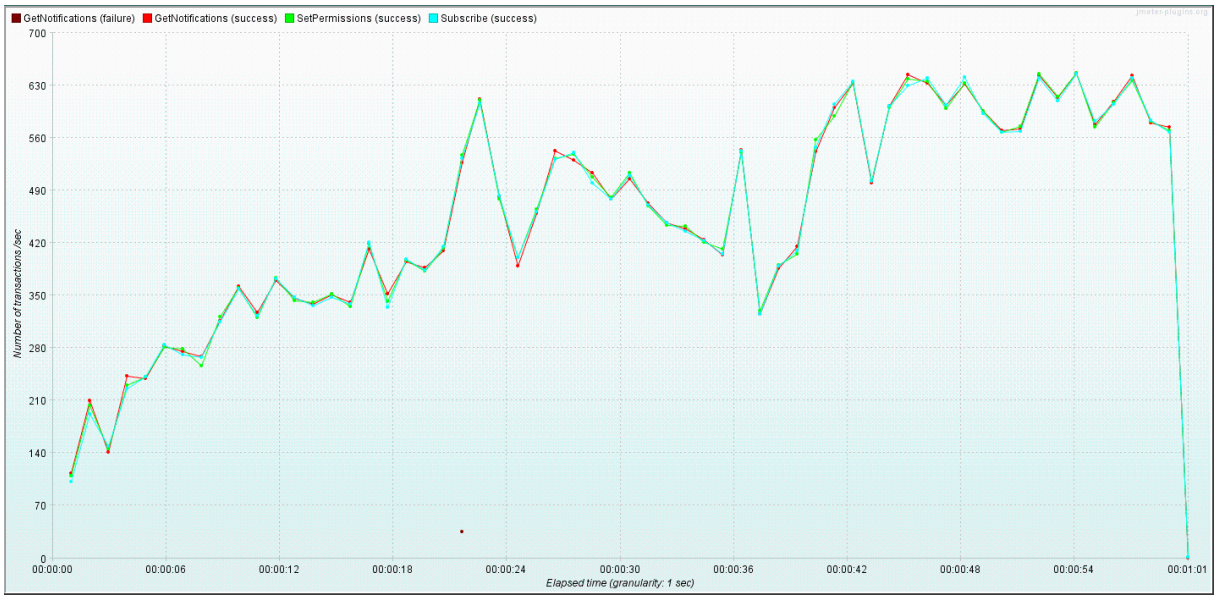

Figure 6.5. Test Plan 2 - Number of Transactions/Second with 100 Threads for 1 Minute – extracted from JMeter.

Figure 6.5 shows the Number of Transactions per Second during 1 Minute, and we can conclude from the graph that qNotify can handle more requests per second of Test Plan 2 (GetNotifications, SetPermissions, Subscribe), compared with requests of Test Plan 1 (SendPush, SendEmail, SendSMS, SendAll). In this test, it was registered peaks of +630 Transactions/Second from 42 seconds to 58 and the average number of transactions per second has also increased compared with the Test Plan 1 with 100 Threads.

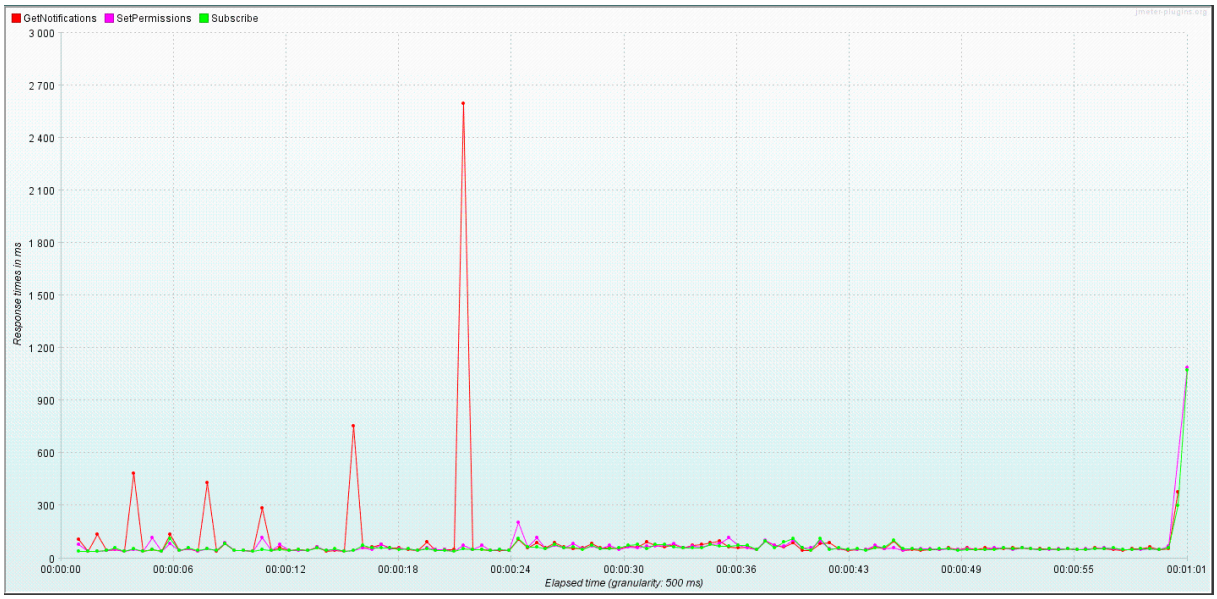

Figure 6.6. Test Plan 2 - Response Times with 100 Threads for 1 Minute – extracted from JMeter.

As illustrated in Figure 6.6, we also conclude that in average the response times during the whole period of 1 minute was also lower compared with response times of Test Plan 1 under Thread Group 3 conditions (with 100 Threads), although it was registered higher maximum values of response time, this detail happened only in GetNotifications request, and the reason why that happens, it's because when qNotify is accessing the database, it requires to find all notifications for the provided User, comparing with Subscribe and SetPermissions that just require to manage 1 ObjectId per request, GetNotifications might manage up to the number of notifications sent to that User.

Table 6.6. Test Plan 2 - Result Statistics with 100 Threads – extracted from JMeter Dashboard.

Requests	Executions			Response Times (ms)						Throughput
Label	#Samples	KO	Error %	Average	Min	Max	90th pct	95th pct	99th pct	Transactions/s
Total	**82669**	**36**	**0.04%**	**72.40**	**34**	**21064**	**66.00**	**82.00**	**310.00**	**1357.48**
GetNotifications	27589	36	0.13%	97.58	34	21064	68.00	87.00	347.00	459.01
SetPermissions	27557	0	0.00%	61.02	34	5750	69.00	87.95	349.00	453.64
Subscribe	27523	0	0.00%	58.55	34	7120	68.00	87.00	336.99	453.79

Table 6.7. Test Plan 2 – APDEX with 100 Threads for 1 Minute – extracted from JMeter Dashboard.

Apdex	T (Toleration threshold)	F (Frustration threshold)	Label
0.996	**500 ms**	**1 sec 500 ms**	**Total**
0.994	500 ms	1 sec 500 ms	GetNotifications
0.996	500 ms	1 sec 500 ms	SetPermissions
0.997	500 ms	1 sec 500 ms	Subscribe

From Test Plan 2 Statistics table (Table 6.6), we can conclude what was already expected, in fact qNotify is able to handle more requests of Test Plan 2 comparing to requests of Test Plan 1, the main reason for that result, it is because all requests from Test Plan 1 (SendPush, SendEmail, SendSMS, SendAll) are redirected to the same endpoint, the endpoint (SendNotification), with the provided channels. Meaning that instead of having 1 endpoint per type of request, we have 1 endpoint to 4 types of requests. Although such sacrifice of performance was needed, to build a consistent Notification system. Imagine the case: qDocs wants to notify a User from 2 channels (Email and Push), it would require to send 2 requests to qNotify, and for this reason it was decided that it should only send 1 request per Notification, and the notification is then sent to all provided channels.

Furthermore, we conclude that qNotify accomplished performance requirements in terms of Throughput, Response Time, and consequently Apdex score for scenarios S3 and S4, showing a

throughput of 82K Transactions/minute (1357.48 Transactions/second), an average response time of 72.40 milliseconds and an excellent Apdex score of 0.996 (as seen from Table 6.7).

Test results of Test Plan 2 with Thread Group 4 (200 Threads) are exposed in the appendix. The reason why they are not here, it is because they only prove what Test Plan 1 already proved, that the number of handled requests does not improved like expected and qNotify produced more errors comparing with Thread Group 3 (100 Threads).

6.1.5. Results Discussion

This subsection discusses performance results previously presented, according to the following metrics: Throughput; Response Time; and Apdex Score.

Throughput Conclusion

qNotify is in fact, able to handle up to 50K requests of type (Send Notification), and 82K requests of type (GetNotifications, SetPermissions, Subscribe), both for a period of one minute, without producing errors, and returning for every request a correct response. These results are very positive, given the required 30K requests of each type in overload capacity.

Concluding, qNotify presents for SendNotification request type a throughput of 50K transactions per minute (833 Transactions/second) and for GetNotifications/SetPermissions/Subscribe types of requests, qNotify presents a throughput of 82K transactions per minute (1357 Transactions/second).

Response Time Conclusion

Regarding response time, qNotify presents an average response time of 116.14 milliseconds for Send Notification type of requests and 72.40 milliseconds for requests of type (GetNotifications, SetPermissions, Subscribe), both during a period of one minute. These results show that qNotify perform better than the required average response time of 200 milliseconds presented in scenarios S1 and S3 with the system under normal operation mode, and 300 milliseconds presented in scenarios S2 and S4 with the system under overload operation mode.

Apdex Conclusion

As for the Apdex metric, qNotify scored an excellent result of 0.993, for Send Notification type of requests, during a period of one minute, and also an excellent result of 0.996 for GetNotifications/SetPermissions/Subscribe types of requests. This metric shows that the application that makes requests to qNotify (qDocs in this context), will not feel any frustration when making requests, because qNotify returns a response always within 500 or less milliseconds.

From results previously presented, we conclude that qNotify not only fulfill performance requirements established in subsection 6.1.1, as it also surpasses the three measured metrics required for each scenario.

6.2. Security

Regarding security, it was established that qNotify shall only provide service (handle requests) from authorized applications and all messages exchanged between qNotify and authorized applications must be Confidential, Integral, and Authenticated.

6.2.1. Confidentiality, Integrity and Authenticity

In order to provide Confidentiality, Integrity and Authentication in exchanged messages, it was decided that qNotify will perform under TLS Protocol, meaning that qNotify will communicate with other applications through HTTPS requests, and for that qNotify require a Digital Certificate issued by a Certification Authority (CA).

TLS solve the problems of Confidentiality, Integrity and Authentication, because both parties (qNotify and qDocs in this context) require to be Certificated by CA's, and the Certificate contains among other data, the Public Key of the application that was certified (certificates are the key for the Handshake phase of TLS).

In the TLS Protocol Handshake Phase, first qDocs send a list of algorithms it supports along with a qDocs nonce, qNotify chooses algorithms from the list, and sends to qDocs the choice, the digital certificate, and qNotify nonce.

qDocs will then verify the digital certificate and confirms if qNotify is who claims to be, obtains qNotify public key from the certificate, and then generates the Pre Master Secret (PMS), qDocs encrypts the PMS with qNotify public key and sends it to qNotify.

qNotify decrypts the PMS with his own private key and both qDocs and qNotify independently compute the Encryption and MAC Keys from the PMS. Both qDocs and qNotify send a MAC of all exchanged messages.

After the handshake phase, qNotify and qDocs exchange data under the TLS Record Protocol. Record Protocol breaks data in fragments, added the MAC with each fragment, and then Encrypt the Data Fragment + MAC.

This security requirement (Confidentiality, Integrity and Authentication of messages) was not tested, because qNotify is not deployed in a certificated server at the date of this document.

6.2.2. Authorization

Now that TLS was explained, and assuming both applications are communicating under this protocol, qNotify is still a webservice and will respond to any HTTPS request coming from any application. To prevent this from happening, it was added the authorization property to qNotify.

“Authorization defines an entities’ privileges to the different resources and services of a system and limits interactions with resources according to the assigned privileges” [37].

As explained in Chapter 4, in order to serve requests only from allowed applications, it was required to add a new endpoint to qNotify, RegisterApp (Table 4.1). This endpoint allows to register an application with provided attributes: “name” and “secret”. It is public for now, to promote qNotify growth with the engagement of any other application that requires Notifications services.

As for the “name” attribute, it is stored in plaintext, while the “secret” is first applied to a Key Derivation Function (KDF), with SHA-256 algorithm, with a Salt length of 64 bit and the number of iterations applied is 10.000, and only then the result of the KDF is stored in qNotify database in AuthorizedApp Model (Figure 6.7 exposes the example of qDocs registration object stored in qNotify AuthorizedApp Collection).

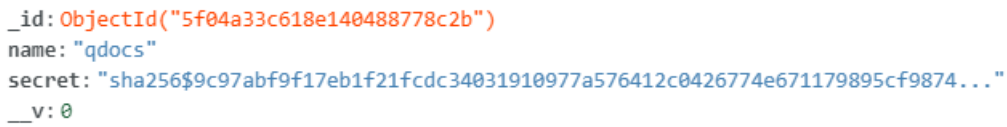

Figure 6.7. qDocs Registration Object extracted from qNotify AuthorizedApp Collection.

It was also added to qNotify a validation function that intercepts all requests for any endpoint except RegisterApp endpoint (because this one is public for everyone that wants to register, for then use qNotify). The validation function check if the application that requested is allowed to access the endpoint, and if it is, the request is handled properly in the specific endpoint logic, if it don’t, the flow is interrupted with a status code 401 Unauthorized Access.

The validation process first checks if the request headers contain both an application “name” and a “secret”, otherwise reject the request. If it does (contain “name” and “secret” in headers), it checks if the application is registered in the database, if it is not it rejects the request, and only after these validations approval, the validator verifies if the “Secret” in request header corresponds to the Hashed Salt secret stored in the database for that correspondent application. In case of all validation’s approval, the request is then handled properly in the corresponding endpoint.

According to [38], the KDF applied to the password, increases key strengthening and provides Rainbow Table Attacks protection, which is very useful if an attacker gains access to the database, it

won’t be able to obtain the password. KDF also mitigates brute force attacks by having a sliding computational cost, the higher the number of iterations applied, the harder it is to perform the brute force attack (in this case it is applied 10.000 iterations with a 64 bit salt).

6.2.3. Scenarios and Results

Within the scope of qNotify regarding to security requirements it was designed one scenario to test authorization correctness:

Security Scenario:

- Source: Unregistered Application
- Stimulus: Performs a “SendNotification” request
- Artifact: qNotify
- Environment: Under Normal Operations
- Response: Request is Rejected with a return Code 401, Unauthorized
- Response Measure: With a probability of correct judgment of 100%

The above scenario was tested and Figure 6.8 presents the information that is displayed to whom attempts to access resources without authorization. As we can conclude from the figure, any application that is not registered in the system, is not allowed to perform requests to qNotify.

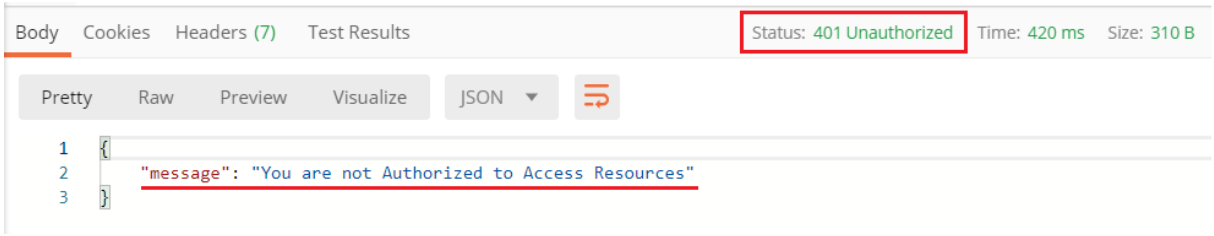

Figure 6.8. Unauthorized Send Notification Request Example – extracted from Postman.

The reason why it was not performed more scenarios and further evaluation of them is that part of the security depends on TLS Protocol which will be enabled when qNotify possess a Digital Certificate.

6.3. Interoperability

Interoperability is "the capability to communicate, execute programs, or transfer data among various functional units" such that the user needs "little or no knowledge of the unique characteristics of those units" [39].

This section presents interoperability aspects of qNotify within itself, such as Push, Email and SMS components, and with other systems outside, as one of the objects of this thesis, qDocs Platform. Since interoperability is a key quality of this thesis, some aspects were already explained in Chapters 4 and 5, and for that reason, this section focuses on the choices made.

6.3.1. Requirements, Scenarios and Results

To explain which interoperability aspects were taken before and during qNotify development, it was design and tested quality attribute scenarios based on qNotify interoperability requirements:

- qNotify should handle requests from other applications (in this work qDocs Platform was the targeted application) without knowing which application is requesting before runtime (this property is applied because qNotify must serve other applications)
- The application that is consuming Notifications services must discover qNotify before runtime (meaning that qDocs must know who is qNotify and its address)
- qNotify must always return a proper response, while during successful operations, incorrect input, unauthorized access and even when an error occurs

Considering the above requirements, it was made the following scenarios to evaluate interoperability quality attribute:

First Interoperability Scenario (S1):

- Source: qDocs Platform
- Stimulus: Request to Send Notification with User, Message Content and Channels Information in Body along with Authorization Credentials in Headers
- Artifact: qNotify
- Environment: System Known Before Runtime
- Response: qNotify Validates Request Authorization, then Delegates the Notification Sending Process to the Respective Components (Push, Email, SMS), Save Notification, and Returns an OK Code.
- Response Measure: 99% of the information exchanged correctly processed

Second Interoperability Scenario (S2):

- Source: qDocs Citizen Consults Notifications History
- Stimulus: UserId Sent in Request along with Authorization Credentials in Headers
- Artifact: qNotify
- Environment: System Known Before Runtime

- Response: qNotify Validates Request Authorization and Returns the Notifications List for the provided User along with an OK Code.
- Response Measure: 99% of the information exchanged correctly processed

Above interoperability scenarios were already tested during performance tests in Section 6.1, for every performance result it was provided the statistics with percentage of information exchanged correctly (Table 6.3 and Table 6.6).

From those tables, we conclude that interoperability scenarios S1 and S2 were achieved, given that in 51K Send Notification requests, 100% of the information was exchanged correctly, and in 27K GetNotifications requests, 99.87% were exchanged correctly, only with 0.13% resulting in incorrect processed request.

6.3.2. External (Webservices)

qNotify Server as explained in Section 2.3, was developed in Node.js, and for that reason, it was used Express, as it is called the standard server structure for Node.js web applications and APIs. As explained in Chapter 5, qDocs and qNotify exchange information throughout standard communication protocol and with standard data type messages.

During the design phase of qNotify, it was scheduled meetings with stakeholders to clarify which was the operational requirements for exchanging data, how much and what type of information is required to be exchanged for the system to perform its functions, and it was decided that qNotify for all requests except RegisterApp, requires the Authorization Credentials from who is requesting, provided in request Headers, along with the UserId in the request Body for qNotify to locate the targeted User in context. Depending on the type of request, it is also required to provide more information, for example, Send Notification request, also requires to be provided the Message Content, Channels List, and the User PhoneNumber and Email, in order for qNotify to send the notification through the provided channels list.

For qNotify web service, it was decided that for this stage that it would bring more benefits to use REST as the standard communication protocol, due to the following REST Advantages over SOAP:

- REST messages contain fewer characters comparing with SOAP making message size also lower, leading to a performance difference favoring REST, this performance difference is relevant when considering systems that exchange a large number of messages with other systems (as qNotify does);
- REST requires the use of HTTP and assumes direct point-to-point communication, which is very important in terms of security as explained in Section 6.2;
- REST also has a smaller learning curve and it is the closest to other web technologies philosophy.

As for the data exchanged in messages, it was decided to use JSON instead of XML for the following reasons:

- JSON is significantly less verbose than XML, because XML necessitates opening and closing tags and JSON uses name/value pairs concisely delineated by “{“ and “}” for objects, “[“ and “]” for arrays, “,” to separate pairs, and “:” to separate name from value;
- With the same amount of information, JSON is almost always significantly smaller, which leads to faster transmission and processing. Also, it is noticed that JSON is serialized and deserialized drastically faster than XML;
- JSON is a subset of JavaScript, so code to parse and package it fits very naturally into JavaScript code, making it easier to display information to User (such as notifications history).

6.3.3. Internal (Components)

For any application to interoperate with qNotify to consume service, first requires to register through RegisterApp endpoint, and only after registration successfully it is allowed to make requests to qNotify/Protected API endpoints (Table 4.2).

As for the SendNotification endpoint, there are three components in charge of delivering the notification (Push, Email, and SMS) which are used according to the notification channels list. Push and Email components make use of open source libraries for Node.js, the Web Push, and Nodemailer, respectively. The Web Push library allows to (by making use of Push API methods) handle Push Notifications subject matter in a consistent way and without cost fees (comparing with Push Notification Services such as the FCM, OneSignal [40], Amazon SNS [41], and many other Proprietary Services).

About the Nodemailer library, allows us to send emails via SMTP, also without associated costs, and decoupled from big Stacks. The SMS component is the only one that makes use of a Proprietary Service, the Nexmo. Nexmo allows us to send SMS with a 0,005 € cost per SMS [27], but there is a trial version that allows us to use it for free during a limited time. This functionality was tested and determined to be working correctly.

7. Conclusion

In this research it was developed a notification system called qNotify, with the ability to deliver messages via Push Notification, Email and SMS channels. qNotify allows any application (Server Application) to register in the system (qNotify), and only after the registration process succeeds, the Application can perform requests to qNotify, such as the following: Subscribe User for Notifications; Send Notification to User; Get User Notifications History; Change/View User Channels Permissions.

The integration of qNotify with the qDocs platform allows qDocs to notify users whenever it needs, requiring only to perform a “SendNotification” request to qNotify, providing the UserId, the Content (Title, Message) and the Channels from which the message will be delivered. In qDocs context it were implemented and demonstrated the following notification use cases: User Association to Curator with Specific Roles; User Disassociation from Curator; Shared Document Accepted; Invitation to Participate in Document with Specific Roles; Invitation to Participate in Document Cancelled; and Document Request. It was also implemented and demonstrated two alternatives for qDocs users to consult their notifications history, from the Messages menu panel and from the Notifications icon. Both perspectives show the history of messages to user, with the difference that from the Notifications icon the user has also access to the number of notifications unseen and unopen yet, displayed near the icon. Finally, it was also implemented and demonstrated the ability to Users to View and Change their channels (Push, Email and SMS) permissions as they desire.

In this research it was also evaluated qNotify according to Performance, Security and Interoperability quality attributes. Regarding performance quality, the tests showed that qNotify is able to handle up to 50K requests of type (SendNotification), and 82K requests of type (GetNotifications, SetPermissions, Subscribe), both for a period of one minute, without producing errors, and returning for every request a correct response. qNotify also presented an average response time of 116.14 milliseconds for requests of type (SendNotification), and 72.40 milliseconds for requests of type (GetNotifications, SetPermissions, Subscribe), both during a period of one minute. qNotify also scored an excellent Apdex result of 0.993, requests of type (SendNotification), during a period of one minute, and also an excellent result of 0.996 for GetNotifications/SetPermissions/Subscribe types of requests.

As for Security quality, qNotify provides service only for authorized applications, that prove their authenticity with credentials (“name” and “secret”), passed in request header, along with the request body. qNotify also assumes that it will only perform his functions when deployed in a certificated server, possessing a digital certificate issued by a CA, to exchange HTTPS requests with other applications, to ensure Confidentiality, Integrity and Authenticity in exchanged messages.

Regarding to interoperability quality, qNotify do not require to discover any application that will request for his (qNotify) services before runtime, instead qNotify is able to provide service to any authorized application during runtime. As for the applications that request for qNotify services, these required to

discover qNotify before runtime, they need to perform requests to qNotify specific endpoints with specific parameters to each request.

Finally, regarding the requirements defined in this thesis (Section 1.2), we conclude that all of them were achieved by solving the problem with qNotify. We decided to build our own notification service to avoid the problems of monetary cost and to not depend on other proprietary services, not to mention that some proprietary services increase the price with the usage of service and with the growth of our own product.

7.1. Future work

During this work, more specifically in the design and implementation phase of the solution, some ideas came up, from the people involved in this work, to envision the best possible result of a notification system integrated in qDocs. It is important to clarify that none of these ideas undermine the work presented in this thesis. Follows some aspects to be considered in future work:

Security: Authorization requires more investigation, since the approach used (shared secret) solves the problem for now, but to make it more secure and consistent, it should be migrated to an OAuth 2.0 authorization mechanism.

Scalability: This quality attribute needs to be implemented in a future work of qNotify, to achieve full potential. qNotify should be assisted by a load balancer to distribute the load to qNotify instances, and more tests shall be made to determine the scalability of the system with more instances running.

qNotify/Client: Build a front-end interface for applications to: Register their applications with a proper user interface, to connect to the RegisterApp endpoint; Consult their user's subscriptions; Consult notifications sent to each user; Consult their user's permissions.

qNotify/Server: qNotify must allow Applications to define Users Subscriptions by Group, to enable Applications to notify all Users that are subscribed in a certain group, without providing the user details.

Functionalities: Explore and implement more qDocs notifications Use Cases, such as: Defining qDocs to send notifications to Users giving a specific data in time; Notify multiple users at once; Notify users when documents are approved by the curator.

Production: All work presented in this thesis, including the qNotify server and the evaluation made for it, was performed in a development instance. qNotify should be deployed into production in future work.

References

[1] E. Isikligil, S. Samakay and D. Kilinç, "A Prototype Framework for High Performance Push Notifications, International Journal of Computer Applications," May 2017.

[2] M. Mulligan, P. Nykänen and J. Toijala, "Web Services Push Gateway," *Nokia Corporation, Espoo (FI),* 7 August 2007.

[3] Medium, "6 Reasons Why to Use Web Push Notifications on Your Website," [Online]. Available: https://medium.com/yeello-digital-marketing-platform/6-reasons-why-to-use-web-push-notifications-on-your-website-b142363370f5 . [Accessed May 2020].

[4] I. Podnar, M. Hauswirth and M. Jazayeri, "Mobile push: Delivering content to mobile users, In 22nd International Conference on. IEEE," 2002.

[5] MDSS Meta Documents Systems Lda, "qDocs, Citizen Centric Document Technology, white paper, v1.4," December 2018.

[6] World Wide Web Consortium (W3C), "Push API," 4 February 2015. [Online]. Available: https://www.w3.org/TR/push-api/. [Accessed May 2020].

[7] WHATWG, "Notifications API, Living Standard," [Online]. Available: https://notifications.spec.whatwg.org/. [Accessed May 2020].

[8] Internet Engineering Task Force (IETF), "RFC 8030 - Generic Event Delivery Using HTTP Push," [Online]. Available: https://tools.ietf.org/html/rfc8030. [Accessed May 2020].

[9] Internet Engineering Task Force (IETF), "RFC 8292 - Voluntary Application Server Identification (VAPID) for Web Push," [Online]. Available: https://tools.ietf.org/html/rfc8292. [Accessed May 2020].

[10] World Wide Web Consortium (W3C), "Service Workers Nightly," [Online]. Available: https://w3c.github.io/ServiceWorker/. [Accessed May 2020].

[11] MDN Web Docs, "Web technology for developers, Web API," [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Notification/tag. [Accessed June 2020].

[12] A. Silva, J. Saraiva and J. Menezes, "Citizen-Centric and Multi-Curator Document Automation Platform: The Curator Perspective," *28th International Conference on Information Systems Development, AIS,* 2019.

[13] D. Caramujo, "A Citizen-Centric and Multi-Curator Document Automation Platform: The qBox and Further Interoperablity Aspects, Master Thesis," Instituto Superior Técnico, Universidade de Lisboa, 2019.

[14] Udemy, "Official WebPage," [Online]. Available: https://www.udemy.com/. [Accessed May 2020].

[15] "NodeJS Official Web Page," [Online]. Available: https://nodejs.org/en/about/. [Accessed May 2020].

[16] TJ Holowaychuk, StrongLoop and others, "Express, fast, unopinionated, minimalist web framework for Node.js," [Online]. Available: https://expressjs.com/. [Accessed May 2020].

[17] MongoDB Inc., "MongoDB Official Documentation," [Online]. Available: https://docs.mongodb.com/. [Accessed May 2020].

[18] Google, "Angular Official Web Page," [Online]. Available: https://angular.io/. [Accessed May 2020].

[19] Microsoft, "Typescript Official Web Page," [Online]. Available: https://www.typescriptlang.org/. [Accessed May 2020].

[20] Microsoft, "Introduction to ASP.NET Core, Official Documentation," [Online]. Available: https://docs.microsoft.com/pt-br/aspnet/core/?view=aspnetcore-3.1. [Accessed May 2020].

[21] G. Albertengo, F. Debele, W. Hassan and D. Stramandino, "On the performance of web services, google cloud messaging and firebase cloud messaging," *Department of Electronics and Telecommunications, Politecnico di Torino Corso Duca Degli Abruzzi, 24, Turin, 10129, Italy,* February 2018.

[22] L. Richardson, S. Ruby and D. Hansson, RESTful Web Services, vol. 1, O'Reilly Media, 2008.

[23] J. Snell, D. Tidwell and P. Kulchenko, Programming Web Services with SOAP, O'Reilly Media, 2009.

[24] R. Santos, "qDocs – A Citizen-Centric Electronic Document and Record Management System, Master Thesis," Instituto Superior Técnico, Universidade de Lisboa, 2016.

[25] "Nodemailer Official Web Page," [Online]. Available: https://nodemailer.com/about/. [Accessed May 2020].

[26] TechTarget, "SMTP (Simple Mail Transfer Protocol), Official Web Page," [Online]. Available: https://whatis.techtarget.com/definition/SMTP-Simple-Mail-Transfer-Protocol. [Accessed June 2020].

[27] Vonage, "Nexmo SMS API Official Web Page," [Online]. Available: https://developer.nexmo.com/messaging/sms/overview. [Accessed May 2020].

[28] Wikipedia, "Software as a service," [Online]. Available: https://en.wikipedia.org/wiki/Software_as_a_service#Distribution. [Accessed June 2020].

[29] GitHub, "Web Push library for Node.js," [Online]. Available: https://github.com/web-push-libs/web-push#api-reference. [Accessed May 2020].

[30] Sparx Systems, "Enterprise Architect," [Online]. Available: https://sparxsystems.com/products/ea/. [Accessed May 2020].

[31] Medium, "Switching to ES6," [Online]. Available: https://medium.com/predict/switching-to-es6-part-2-string-interpolation-and-template-literals-2f1b0ee56740. [Accessed June 2020].

[32] TechTerms, "TechTerms: Throughput Definition," [Online]. Available: https://techterms.com/definition/throughput. [Accessed June 2020].

[33] J. Colantonio, "Performance Testing Response Time," TEST GUILD, [Online]. Available: https://testguild.com/performance-testing-response-time/. [Accessed June 2020].

[34] New Relic, "New Relic Documentation," [Online]. Available: https://docs.newrelic.com/docs/apm/new-relic-apm/apdex/apdex-measure-user-satisfaction. [Accessed May 2020].

[35] TechTarget, "Search IT Operations, Official Web Page," [Online]. Available: https://searchitoperations.techtarget.com/definition/Application-Performance-Index-Apdex. [Accessed May 2020].

[36] The Apache Software Foundation, "Apache JMeter," [Online]. Available: https://jmeter.apache.org/. [Accessed May 2020].

[37] R. Reussner, J. Mayer, J. Stafford, S. Overhage, S. Becker and P. Shroeder, Quality of Software Architectures and Software Quality, First International Conference on the Quality of Software Quality Architectures, 2005.

[38] C. Ribeiro, A. Zúquete, M. Correia, M. Pardal and R. Chaves, "Security in Computer Networks and Systems," in *Department of Information Systems and Enginnering, SIRS Course, Instituto Superior Técnico, Universidade de Lisboa*, 2019.

[39] MITRE, "Systems Engineering Guide, Develop and Evaluate Integration and Interoperability," [Online]. Available: https://www.mitre.org/publications/systems-engineering-guide/se-lifecycle-building-blocks/systems-integration/develop-and-evaluate-integration-and-interoperability-iampi-solution-strategies. [Accessed June 2020].

[40] OneSignal, Inc., "OneSignal Official Web Page," [Online]. Available: https://onesignal.com/. [Accessed May 2020].

[41] Amazon.com, Inc, "AWS SNS," [Online]. Available: https://aws.amazon.com/pt/sns/?whats-new-cards.sort-by=item.additionalFields.postDateTime&whats-new-cards.sort-order=desc. [Accessed May 2020].

Appendix A – Performance Results with Thread Group 1 & 2

Thread Group 1

Requests	Executions			Response Times (ms)						Throughput	Network (KB/sec)	
Label	#Samples	KO	Error %	Average	Min	Max	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**12651**	**0**	**0.00%**	**47.28**	**35**	**1132**	**53.00**	**62.40**	**97.00**	**210.73**	**65.34**	**114.93**
SendAll	3160	0	0.00%	47.95	35	1122	53.00	62.95	101.00	52.84	16.31	28.74
SendEmail	3164	0	0.00%	46.98	35	1123	53.00	65.00	96.00	52.83	16.41	28.84
SendPush	3166	0	0.00%	47.86	35	1132	53.00	64.00	100.00	52.74	16.38	28.79
SendSMS	3161	0	0.00%	46.34	35	403	52.00	59.00	93.38	52.81	16.40	28.83

Thread Group 2

Requests	Executions			Response Times (ms)						Throughput	Network (KB/sec)	
Label	#Samples	KO	Error %	Average	Min	Max	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**11120**	**0**	**0.00%**	**53.83**	**34**	**1146**	**72.00**	**82.00**	**108.00**	**185.14**	**716.28**	**71.88**
GetNotifications	3710	0	0.00%	68.02	50	1146	81.00	94.00	117.00	61.77	685.62	15.86
SetPermissions	3706	0	0.00%	47.10	34	349	56.00	67.00	96.00	61.83	15.34	18.05
Subscribe	3704	0	0.00%	46.35	34	376	54.00	63.00	94.95	61.85	15.40	38.11

Requests	Executions			Response Times (ms)						Throughput	Network (KB/sec)	
Label	#Samples	KO	Error %	Average	Min	Max	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**1195**	**0**	**0.00%**	**49.89**	**35**	**223**	**64.00**	**70.00**	**99.00**	**19.92**	**77.12**	**7.73**
GetNotifications	399	0	0.00%	62.16	51	223	68.00	76.00	111.00	6.65	73.82	1.71
SetPermissions	398	0	0.00%	43.74	35	96	48.00	52.00	90.03	6.67	1.65	1.95
Subscribe	398	0	0.00%	43.74	35	101	49.00	57.05	90.09	6.67	1.66	4.11

Requests	Executions			Response Times (ms)						Throughput	Network (KB/sec)	
Label	#Samples	KO	Error %	Average	Min	Max	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**1329**	**0**	**0.00%**	**44.92**	**36**	**310**	**50.00**	**53.00**	**89.70**	**22.15**	**6.87**	**12.08**
SendAll	332	0	0.00%	44.46	36	110	49.00	52.00	91.68	5.55	1.71	3.02
SendEmail	332	0	0.00%	45.51	36	310	50.00	53.00	88.67	5.55	1.72	3.03
SendPush	333	0	0.00%	44.80	36	98	50.00	56.00	90.66	5.55	1.72	3.03
SendSMS	332	0	0.00%	44.91	36	221	49.00	53.00	95.34	5.55	1.72	3.03

Appendix B – Performance Results with Thread Group 4 & 5

Thread Group 5

Requests	Executions			Response Times (ms)						Throughput	Network (KB/sec)	
Label	#Samples	KO	Error %	Average	Min	Max	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**48266**	**902**	**1.87%**	**636.58**	**36**	**47004**	**568.00**	**1430.80**	**21036.00**	**590.71**	**209.94**	**316.16**
SendAll	11803	4	0.03%	204.57	36	21039	364.00	499.00	1879.96	185.12	57.26	100.66
SendEmail	12156	265	2.18%	755.35	36	27434	442.00	1317.15	21023.00	183.98	66.88	98.25
SendPush	12232	437	3.57%	979.71	36	21278	434.00	1572.70	21238.00	186.91	74.25	98.39
SendSMS	12075	196	1.62%	591.71	37	47004	415.00	1172.80	21036.00	148.39	51.92	79.69

Requests	Executions			Response Times (ms)						Throughput	Network (KB/sec)	
Label	#Samples	KO	Error %	Average	Min	Max	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**94560**	**337**	**0.36%**	**331.93**	**35**	**103171**	**342.00**	**1112.95**	**15089.00**	**914.31**	**234.52**	**352.46**
GetNotifications	31696	154	0.49%	392.34	35	103171	333.00	1062.00	7157.79	306.47	79.26	78.03
SetPermissions	31512	109	0.35%	319.25	35	62375	333.00	1084.00	7596.59	324.65	83.20	94.15
Subscribe	31352	74	0.24%	283.60	35	54478	332.00	695.00	7100.97	342.38	87.17	210.14

Thread Group 4

Requests	Executions			Response Times (ms)						Throughput	Network (KB/sec)	
Label	#Samples	KO	Error %	Average	Min	Max	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**90324**	**35**	**0.04%**	**134.07**	**35**	**38661**	**103.00**	**352.00**	**1542.99**	**1263.79**	**314.66**	**489.01**
GetNotifications	30176	17	0.06%	151.89	35	38661	109.00	347.95	1435.00	422.21	104.91	107.97
SetPermissions	30112	7	0.02%	127.07	35	33082	107.00	349.00	1778.98	455.13	113.14	132.42
Subscribe	30036	11	0.04%	123.19	35	37243	106.00	350.00	1715.99	446.78	111.65	274.78

Requests	Executions			Response Times (ms)						Throughput	Network (KB/sec)	
Label	#Samples	KO	Error %	Average	Min	Max	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**49637**	**15**	**0.03%**	**243.43**	**38**	**21039**	**416.00**	**600.00**	**1812.83**	**769.20**	**239.06**	**417.90**
SendAll	12336	0	0.00%	231.14	38	15671	405.00	563.00	1570.89	191.83	59.20	103.97
SendEmail	12433	0	0.00%	232.32	40	15210	403.00	546.30	1644.64	200.07	62.13	108.83
SendPush	12486	14	0.11%	279.76	39	21039	420.00	684.65	3234.26	193.52	60.62	105.15
SendSMS	12382	1	0.01%	230.19	39	21036	406.00	563.85	1637.17	192.34	59.77	104.61

Appendix C – qNotify API Documentation

URL	Title	Type	Endpoint	Headers	JSON Parameters	Return

http://localhost:3000/	Save Subscription	POST	savesubscription	“name”: [string],
				“secret”: [string]
					{
					“UserId”: [string],
					“Subscription”: {
					“endpoint”: [string],
					“expirationDate”:
					[DateTime],
					“keys”: {
					“p256”: [string],
					“auth”: [string]
					}
					}
					}
						{
						“message”: [string]
						}

	Unsubscribe	POST	unsubscribe	“name”: [string],
				“secret”: [string]
					{
					“UserId”: [string]
					}
						{
						“message”: [string]
						}

	Get Subscription	POST	subslist	“name”: [string],
				“secret”: [string]
					{
					“UserId”: [string],
					“Subscription”: {
					“endpoint”: [string],
					“expirationDate”:
					[DateTime],
					“keys”: {
					“p256”: [string],
					“auth”: [string]
					}
					}
					}
						{
						“endpoint”: [string],
						“expirationDate”: [DateTime],
						“keys”: {
						“p256”: [string],
						“auth”: [string]
						}
						}

	Send Notification	POST	sendnotification	“name”: [string],
				“secret”: [string]
					{
					"User": {
					"UserId": [string],
					"Email": [string],
					"PhoneNumber": [string]
					},
					“Content”: {
					“title”: [string],
					“message”: [string],
					“url”: [string],
					“ttl”: [integer],
					“icon”: [string]
					},
					“Channels”: {
					“Push”: [boolean],
					“Email”: [boolean],
					“SMS”: [boolean]
					}
					}
						{
						“message”: [string]
						}

	Get Notifications	POST	notificationslist	“name”: [string],
				“secret”: [string]
					{
					“UserId”: [string]
					}
						[{
						"User": {
						"UserId": [string],
						"Email": [string],
						"PhoneNumber": [string]
						},
						“Content”: {
						“title”: [string],
						“message”: [string],
						“url”: [string],
						“ttl”: [integer],
						“icon”: [string]
						},
						“Channels”: {
						“Push”: [boolean],
						“Email”: [boolean],
						“SMS”: [boolean]
						}
						}]

	Set Unseen	POST	notificationseen	“name”: [string],
				“secret”: [string]
					{
					“NotificationId”: [string]
					}
						{
						“message”: [string]
						}

	Save Permissions	POST	savepermissions	“name”: [string],
				“secret”: [string]
					{
					“UserId”: [string],
					“Push”: [boolean],
					“Email”: [boolean],
					“SMS”: [boolean]
					}
						{
						“message”: [string]
						}

	Get Permissions	POST	permissionslist	“name”: [string],
				“secret”: [string]
					{
					“UserId”: [string]
					}
						{
						“Push”: [boolean],
						“Email”: [boolean],
						“SMS”: [boolean]
						}

	Register App	POST	registerapp	None	{
					“name”: [string],
					“secret”: [string]
					}
						{
						“message”: [string]
						}