qNotify – Notification System and Integration with a Document Automation Platform

Leonardo Pereira Troeira
leonardo.troeira@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal
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.

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]. 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 [4]. 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.
2. Background
This chapter introduces Push Notifications and qDocs Platform, as they are the main subjects discussed in this research.

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 [5]. As exposed in Figure 1, 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 [6].

![Push Notification Workflow](image)

**Push service** is a third-party component that acts as an intermediary ensuring reliable and efficient delivery of push messages to a Client App. 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. ServerApp requests push delivery by sending a message to push service, and then the push service delivers the message with a push event.

**Voluntary Application Server Identification (VAPID)** is a technique that allows an application server to identify itself to a push service. 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. 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 [7].

**A Service Worker (SW)** is required to be installed in the user agent, for a client application to support push notifications. 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 [8].

**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.

**A Push Subscription** is a message delivery context established between the user agent and the push service on behalf of a web application. Whenever an application requires to send push messages to a client, it needs to create a push subscription specific to that client. 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.

2.2. 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 [4].

qDocs general architecture is defined by the integration of several applications, namely (as suggested by Figure 2): Docs/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 [9].

![qDocs General Architecture](image)
3. qNotify Design and Implementation
This chapter presents qNotify. Notify is a webservice that allows qDocs platform as well as any other application, the ability to send notifications to users.

3.1. qNotify Architecture
As suggested in Figure 3, 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 requires to be subscribed for push notifications. To perform a correct push subscription, the following steps may happen (as suggested in Figure 3): (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.

After the user is subscribed for push notifications, the Application can send Push, Email and/or SMS notifications (using qNotify), following these steps: (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.

Users can consult their notifications history as follows: (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.

qNotify allows users to receive notifications through Push, Email and SMS channels. Users can define their permission channels, as follows: (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.

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). For an 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.

3.2. qNotify Domain Model
As suggested in Figure 4, regarding qNotify domain model, the AuthorizedApp class represents any Application that is registered in the system and consequently authorized to use qNotify services. 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 with the number of unseen/not open notifications.

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

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 5.2), 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.

3.3.2. User Push Endpoints Synchronization

This subsection presents three problems and respective solutions regarding User Push Endpoints, that emerged during qNotify development.

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. **qNotify Integration with qDocs**
This chapter presents qNotify integration with qDocs platform. Section 4.1 presents qDocs architecture with qNotify integration, section 4.2 describes the functionalities and use cases implemented in qDocs context with qNotify integration.

4.1. **qNotify – qDocs Architecture**
This section presents the architecture of qDocs Platform after the integration with qNotify. As suggested in Figure 5, qDocs communicate with qNotify via qNotify/API interface.
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 3), as for the qDocs Platform, takes place where the ServerApp was.

![Figure 5. qNotify Architecture Integrated with qDocs (ArchiMate diagram).](image)

4.2. **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.

4.2.1. **Manage User Push Subscriptions**
The first time a User authenticates in qDocs, it is prompted with an “Allow Notifications” pop-up, 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 3.3.2), qNotify will take that same service worker and save in the database the Push Subscription with the current UserId.

4.2.2. **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 6). To it so, the User only need to select in each channel if allow or deny notifications, and finally select Save Permissions.
4.2.3. Display Notifications History to User

Users can view their notifications history 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, deliver date, the subject, and the message content (Figure 7). Push Notifications also contain an URL attribute, specifically defined to each type of notification, to redirect the user to the specific Document, Curator, or 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 8, 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.

4.2.4. 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 (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; Roles the Citizen was added in that specific Curator; 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.

4.2.5. Template Messages Construction
As already mentioned before, qNotify was developed as an isolated webservice 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 [10]. Each notification was implemented in both qDocs supported languages, Portuguese and English.

4.3. 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 4.2.4, 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 4.2.2). 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 4.2.3). Both ways display the notifications history to the user, with the difference that the second alternative (Notifications icon) provide the feature of unseen and unopened notifications number displayed near notifications icon (as seen in 4.2.3).

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

5.1. Performance
This section presents and discusses performance results obtained for Test Plan 1 and Test Plan 2, under Thread Group 3 conditions, considering the following metrics: Throughput, Response Time, and Application Performance Index (Apdex).

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).

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

<table>
<thead>
<tr>
<th>Request</th>
<th>Min</th>
<th>Mean</th>
<th>Max</th>
<th>SD</th>
<th>90%ile</th>
<th>95%ile</th>
<th>99%ile</th>
<th>Median</th>
<th>Throughput</th>
<th>Response Time</th>
<th>Apdex</th>
<th>Throughput</th>
<th>Response Time</th>
<th>Apdex</th>
</tr>
</thead>
<tbody>
<tr>
<td>SendPush</td>
<td>1531</td>
<td>1364</td>
<td>1536</td>
<td>43</td>
<td>1421</td>
<td>1435</td>
<td>1459</td>
<td>1437</td>
<td>15.00</td>
<td>15.00</td>
<td>3.00</td>
<td>20.00</td>
<td>3.00</td>
<td>20.00</td>
</tr>
<tr>
<td>SendEmail</td>
<td>1436</td>
<td>1436</td>
<td>1436</td>
<td>0.00</td>
<td>1436</td>
<td>1436</td>
<td>1436</td>
<td>1436</td>
<td>16.00</td>
<td>16.00</td>
<td>3.00</td>
<td>20.00</td>
<td>3.00</td>
<td>20.00</td>
</tr>
<tr>
<td>SendSMS</td>
<td>1531</td>
<td>1531</td>
<td>1531</td>
<td>0.00</td>
<td>1531</td>
<td>1531</td>
<td>1531</td>
<td>1531</td>
<td>16.00</td>
<td>16.00</td>
<td>3.00</td>
<td>20.00</td>
<td>3.00</td>
<td>20.00</td>
</tr>
<tr>
<td>SendAll</td>
<td>1531</td>
<td>1531</td>
<td>1531</td>
<td>0.00</td>
<td>1531</td>
<td>1531</td>
<td>1531</td>
<td>1531</td>
<td>16.00</td>
<td>16.00</td>
<td>3.00</td>
<td>20.00</td>
<td>3.00</td>
<td>20.00</td>
</tr>
</tbody>
</table>

Table 2. Test Plan 1 – APDEX with 100 Threads (extracted from JMeter Dashboard).

<table>
<thead>
<tr>
<th>Apdex</th>
<th>T (Stimulation threshold)</th>
<th>F (Frustration threshold)</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.003</td>
<td>500 ms</td>
<td>1 sec 500 ms</td>
<td>Total</td>
</tr>
<tr>
<td>0.992</td>
<td>500 ms</td>
<td>1 sec 500 ms</td>
<td>SendPush</td>
</tr>
<tr>
<td>0.993</td>
<td>500 ms</td>
<td>1 sec 500 ms</td>
<td>SendEmail</td>
</tr>
<tr>
<td>0.994</td>
<td>500 ms</td>
<td>1 sec 500 ms</td>
<td>SendSMS</td>
</tr>
</tbody>
</table>

As shown in Statistics Table 1, 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 2, qNotify scored an excellent result of 0.993 (T = 500 and F = 1500).

**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) under Thread Group 3.

Table 3. Test Plan 2 - Result Statistics with 100 Threads – extracted from JMeter Dashboard.

<table>
<thead>
<tr>
<th>Request</th>
<th>Average</th>
<th>P50</th>
<th>P90</th>
<th>P99</th>
<th>T</th>
<th>Lavel</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetNotifications</td>
<td>21.14</td>
<td>34.16</td>
<td>58.1</td>
<td>82.9</td>
<td>72.4</td>
<td>50K</td>
</tr>
<tr>
<td>SetPermissions</td>
<td>21.05</td>
<td>34.16</td>
<td>58.1</td>
<td>82.9</td>
<td>72.4</td>
<td>50K</td>
</tr>
<tr>
<td>Subscribe</td>
<td>21.05</td>
<td>34.16</td>
<td>58.1</td>
<td>82.9</td>
<td>72.4</td>
<td>50K</td>
</tr>
</tbody>
</table>

Table 4. Test Plan 2 – APDEX with 100 Threads for 1 Minute – extracted from JMeter Dashboard.

<table>
<thead>
<tr>
<th>Apdex</th>
<th>T(Expectation threshold)</th>
<th>P(Frustration threshold)</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.996</td>
<td>500 ms</td>
<td>1 sec 500 ms</td>
<td>Total</td>
</tr>
<tr>
<td>0.954</td>
<td>500 ms</td>
<td>1 sec 500 ms</td>
<td>GetNotifications</td>
</tr>
<tr>
<td>0.956</td>
<td>500 ms</td>
<td>1 sec 500 ms</td>
<td>SetPermissions</td>
</tr>
<tr>
<td>0.957</td>
<td>500 ms</td>
<td>1 sec 500 ms</td>
<td>Subscribe</td>
</tr>
</tbody>
</table>

From Test Plan 2 Statistics presented in Table 3, we can conclude that 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 showed 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 4).

**Throughput Conclusion**

qNotify presents for “SendNotification” type of requests 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 “SendNotification” type of requests and 72.40 milliseconds for requests of type “GetNotifications”, “SetPermissions”, “Subscribe”, both during a period of one minute.

**Apdex Conclusion**

As for the Apdex metric, qNotify scored an excellent result of 0.993, for “SendNotification” type of requests, during a period of one minute, and also an excellent result of 0.996 for “GetNotifications”, “SetPermissions”, and “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.

**5.2. 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.

**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).

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.

**Authorization**

Assuming both applications are communicating under TLS 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.

As explained in Chapter 3, in order to serve requests only from allowed applications, it was required to add a new endpoint to qNotify, RegisterApp. 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 9 exposes the example of qDocs registration object stored in qNotify AuthorizedApp Collection).

Regarding qNotify external structure, it was in incorrect processed request. exchanged correctly, only with 0.13% resulting 27K GetNotifications requests, 99.87% were information was exchanged correctly, and in 51K Send Notification requests, 100% of unauthorized access and even when an error occurs. From Table 1 and Table 3, we conclude 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. Regarding qNotify external structure, it 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 4, qDocs and qNotify exchange information throughout standard communication protocol (REST) and with standard data type (JSON) messages. 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. Regarding qNotify internal structure, qNotify delivers message through three components (Push, Email, and SMS) which are used according to the user channels permissions. Push and Email components make use of open source libraries for Node.js, the Web Push, and Nodemailer, respectively. The SMS component is the only one that makes use of a Proprietary Service, the Nexmo.

6. 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 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.
qNotify integration with qDocs platform allows qDocs to notify users whenever it needs. 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 which also includes 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, qNotify presents a throughput of 50K transactions per minute (833 Transactions/second) with an average
response time of 116.14 milliseconds for each request. As for Security quality, qNotify provides service only for authorized applications, that prove their authenticity with credentials (“name” and “secret”). qNotify also assumes that it will only perform his functions exchanging 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.

For future work, regarding 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. It is also part of future work to 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 must also 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. It is also included in future work, to 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. 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


