# Table of Contents

Introduction .......................................................................................................................... 3
Client Description .................................................................................................................. 3
Product Vision ....................................................................................................................... 4
Requirements ....................................................................................................................... 4
  Client Side .......................................................................................................................... 4
  Server Side ........................................................................................................................ 4
Outside Notification Service ................................................................................................. 5
Device Side .......................................................................................................................... 5
  Non-Functional Requirements ............................................................................................ 6
System Architecture ............................................................................................................. 6
Technical Design .................................................................................................................. 6
  Server-Side Design .......................................................................................................... 8
  iOS and Android App Design ............................................................................................ 11
Design and implementation decisions ................................................................................. 11
Results ................................................................................................................................ 12
Appendices .......................................................................................................................... 13
  Registering Your Application for APNs Document .......................................................... 14
Introduction

The system will be a universal push notification engine, which takes REST requests from clients and federates them into push notifications for iOS and Android systems. The engine then sends these requests to either the Google Cloud Messaging Service (GCM) or the Apple Push Notification Service (APNs). These services translate the notifications to the corresponding phone. This engine is essentially a package that allows push notification functionality to be easily built into iOS and Android applications. The project is to create a service that can take in a client request and output that request as a push notification on either an Android or iOS device. An addition to the project, which we completed, was to thread the notification executions on the server so they could run simultaneously.

We also created two graphical user interfaces for iOS and Android applications. For these GUIs, we create a field where users can enter their account number. There is also a send button that can show if your account number is valid or not. If users enter a valid account, the account number will be sent back to our main server. The server will receive the information from this user. Since each user has a unique account id, users can register several different devices under the same account number. For example, if a user has an iPad, an iPhone, and an Android phone, they can download Push Notifications System application for each device by using one account number, “123456”. Every time, the server pushes the notifications for account “123456”, all the devices (the iPad, iPhone, and Android phone) will receive this notification.

Client Description

Our client RadiantIQ is a self-funded cloud-based telecommunications software company whose solutions currently power millions of mobile devices and apps. Their customers take advantage of innovative web APIs to do things like sending SMS messages, buying phone numbers all around the world, and browsing the web from a mobile device. They work in a casual fun, fast-paced environment in Boulder. The main services of RadiantIQ are flexible OSS, powerful platforms, and simple APIs. RadiantIQ delivers a flexible OSS already connected to popular industry APIs and the ability to easily integrate to your specific vendors. For powerful platforms, RadiantIQ provides scalable voice, messaging, and mobility platforms built to serve as the core for mobile services. RadiantIQ exposes simple REST APIs for both OSS provisioning and real-time platform capabilities.
Product Vision

Our API was designed for RadiantIQ who needed a push notification API made so that their clients did not have to implement their own push notification service. This will allow their clients to design robust apps in a shorter amount of time. The API we created uses GCM and APNs for easy delivery of push notifications to devices and allows the client to have the peace of mind that their notifications will reach their targets. Our API gives clients easy implementation of a database that is able to save the device ID of any Android or Apple mobile device associated with the user account. This allows the client to send push notifications to all user devices with just an account id. Also, they will not need to worry about any problems sending push notifications to both Android and Apple devices at the same time because our API comes built in with that functionality. The API also has built in data fields where an account name can be entered and sent to the server. This allows the customers the ability to personalize a user interface easily with account registration functionality because the code for sending it to the server once a button is pressed is included in the API.

Requirements

The system has four main parts: the client side, server side, outside notification service, and device side. The client side will send a request for a push notification to the server-side, which will then send a push notification to the outside notification service (GCM or APNs). This service will then wait for the target device to be online and then send the notification to that device. The requirements for each part are laid out below:

Client Side

• RadiantIQ’s clients should send REST requests to RadiantIQ’s server

• The request format should be dependent on our design

• The client will be supplied with code to send this request based on a JSON file or a JSON object.

Server Side

• Receives messages from RadiantIQ’s clients in the form of REST requests
• Converts client REST requests into JSON. The format of the JSON will depend on whether the target device is Android or iOS.

• The JSON information is then sent to GCM or APNs.

• The service will also receive messages from GCM and APNs regarding notification success after a notification is sent.

• The service will receive JSON messages from user devices in order to register the device for an account.

• There will be an in-memory data structure that will hold notification and device IDs.

Outside Notification Service

The outside notification services are GCM (Google Cloud Messaging) and APNs (Apple Push Notification Service). These services already exist and we will not change their functionality; however, we will be interacting with them.

• GCM or APNs will then send the request to the specified device.

Device Side

• Two boilerplate apps will be designed (one for iOS and one for Android) to incorporate the push notification system. They will have the following functionalities:

  • Registering the device for push notifications with the corresponding service (GCM or APNs)

  • Sending account and device IDs to our server for registration purposes

  • Choosing how to handle push notifications when the app is active
Non-Functional Requirements

- Programming will be done in java 8
- JSON will be used for server side coding for Android and iOS
- GCM service will be used for android messaging
- APNs service will be used for iOS messaging

System Architecture

The system architecture shown in figures 1-3 has two versions: the proof of concept version and the production version. The proof of concept version is the version that our team will be delivering to RadiantIQ and the production version is the version that RadiantIQ will have before taking the system into production. The difference between the two is that the proof of concept design has a HashMap to store device information while the production version has a database. The typical flow is that a client sends a REST request to the server containing a JSON object. The server then parses that object into either Android or iOS notifications and then sends them to either GCM (Google Cloud Messaging) or APNs (Apple Push Notification Service) which then sends the notification to the proper device. The phone lookup is to transform the account id sent by the client into device ids that GCM and APN can use. The notification cross reference is a possible addition that could allow for common notifications to be stored on RadiantIQ’s servers so they wouldn’t have to be sent with the JSON request.
PROOF OF CONCEPT ARCHITECTURE

Client → Radiant IQ Server → GCM
Rest (http) → Hash(Phone Lookup) → Notification CX
Constants → JSON → JSON + IDs → APN

Figure 1

PRODUCTION ARCHITECTURE

Client → Radiant IQ Server → GCM
Rest (http) → JSON → JSON + IDs → APN
Phone Lookup Database
Notification Cross Reference Database

Figure 2
**Technical Design**

**Server-Side Design**

The server implements a RESTful web service, which can be queried using JSON requests. It uses the jersey framework to design the servlet that the various pieces run on. The server essentially consists of three parts: a receiver for push notification requests, a receiver for device registrations, and a thread pool to send the push notifications. The general flow of the server is shown in Figure 4.
Figure 4
**Push Notification Receiver**

The push notification receiver is the meat of the web service design. It accepts incoming requests from RadiantIQ’s clients and transforms those requests into push notifications. This service accepts a JSON request sent over an HTTP connection. A sample JSON request is shown in Figure 5:

```json
{
    "androidAppId": "AIzaSyAueUKYwLxbSz6SSn2uXedUgcCXs8RsQ0U",
    "senderId": "APP_ID_HERE",
    "iosAppIdPassword": "radiantiq",
    "pushNote": [
        {
            "devicelds": [
                {
                    "deviceld": "3032495386"
                }
            ],
            "message": "This is a push notification"
        }
    ]
}
```

*Figure 5*

After parsing the JSON, the receiver converts the JSON into either an AndroidNotification or an IOSNotification. It then sends those notifications to the thread pool to be executed.

**Notification Thread pool**

The notification thread pool is necessary because RadiantIQ expects to receive around 100 million push notifications per month. If there was only a single thread executing all of these notifications, there wouldn’t be enough time in the month for all of them to be sent and the consumers wouldn’t receive them in a timely manner. The thread pool accepts the notifications from the receiver and puts them into a queue. If there are any threads that have finished, it will run the next notification in the queue on the thread. This way RadiantIQ can send hundreds of notifications at once and their consumers can receive them in a timely manner.

**Device Registration Receiver**

The device registration receiver accepts incoming JSON requests from mobile devices in order to register them for push notifications. When the user of an app that implements push notifications first opens the app, the user is asked if they want to receive push notifications. If they do, their device registers them on the proper service (Google Cloud Messaging or Apple
Push Notification Service) with a unique device id. The user will then have to make an account for that app so the app can keep track of them. Once the user registers for the account, the device will send a message to the device registration receiver with the account id and the device id. The receiver will then put this data into a container. Currently the container is a HashMap on the server but the end goal for RadiantIQ is to build a database to store this information. This storage allows one account to be associated with many devices. This means that one request from a client can lead to push notifications on all of the intended recipient’s devices.

iOS and Android App Design

There is an iOS app and an Android app associated with this project; the end goal for this app is to be turned into an API so RadiantIQ’s clients can easily fold this service into their apps. Currently, the app allows users to accept push notifications and to register for an account. Once they register for an account, their information will be sent to the device registration receiver. The app also registers the device for push notifications on the corresponding service (APNs for Apple and GCM for Google). This allows our server to send push notifications to that device. These apps seem simple but they will take care of almost all the leg work of creating a push notification system for the clients that use them.

Design and implementation decisions

- The problem was we needed a service to receive for android that was both free and widely used. We decided to use GCM rather than possible alternatives even though it has speed issues.
  - We decided to use GCM because other similar services are either too expensive or even less reliable. GCM has some speed issues from time to time but it is integrated deeply in the Android OS which means less battery drain and more reliability than other alternatives.

- The problem here we needed communication code that would be able to be used for multiple types users and devices. We decided to use both JSON and XML for communication.
  - Because RadiantIQ has many different clients that will be using this service, we decide to use both JSON and XML for communication. This helps because most app developers typically pick one or the other for communication and if we support both, that means potential clients would be able to easily add our system into their apps. Additionally, JSON is faster and easier to parse so it is the better of the two but some companies only interface with SOAP applications (which only uses XML) so providing XML abilities allows for them to interface with our application without adding new functionality.
• The problem was we needed a web service to send the push notifications. We decided to use a RESTful web service.
  o The first reason for using REST over SOAP is that RadiantIQ has many different clients and REST allows for different types of communication (XML, JSON in our case). This makes it more likely that the clients won’t have trouble integrating this service into their systems. Additionally, REST services are more reliable which is necessary when receiving around 100 million requests a month.

• The problem here was we needed to store the device-ids and device types somewhere to make our job easier. We decided that the device-ids and device types will be stored locally (for now in a hash-map but later in a database).
  o Although it was tempting to make the clients store their own device ids and device types (due to the huge amount that we will be receiving) we decided to store them locally because in some applications, the clients will not need to know what devices they are sending to. They will just need to know what accounts they are sending to. For that reason, we are building a map between accounts and device IDs and types.

• The problem was that there are many android version the newest one being 22, so we needed a minimum version that would allow us to reach the greatest number of users. We decided to target Android version 8.8 as our minimum target because that would reach the greatest number of user devices.
  o Android version 8.8 allows us to cover almost 100% of user devices, this make it the best option since we want the greatest amount of users to be able to use our app. If we had targeted a more updated version, we would not of been able to reach that amount of coverage and would have excluded some users from using it.

Results

The goal of the project was to design a mobile API that would allow for easy implementation of Android and IOS push notifications. It is currently working as intended on both types of mobile devices. We tested the API via individually made mobile apps for each phone type, for android we first tested the above app on an emulator but later moved on to testing it on an actual device. The device that was used was the Samsung Galaxy S6 edge which is the latest phone you can get so it uses the latest API. For the iOS we put the app directly on to a phone, the phones that were used were the iPhone 6 and iPhone 6 plus. The application was tested on iOS 8 and the yet unreleased iOS 9. So as far as testing devices there were no hard pressed constraints because we had the most up to date mobile devices. There were however limitations on the IDE’s used and the servers used to send push notifications to devices. For the IDE’s we had to
exclusively use Android Studio for Android, which can be clunky, slow, and unreliable at times. For Apple, the IDE used was XCode, which used objective C, this was also the only option but Apple’s IDE is easy to use and functions smoothly. The servers we were required to use were GCM for android and APNs for Apple, these servers were the only options to send a push notification to a device from a server. One functionality we wanted to add that we were not able to due to time constraints was intent call. This would allow us to use phone calls as push notifications, this was an extra thing that could have been added in, but was not a major concern to the project as a whole and is not part of the definition of done. Additionally, we wanted to add functionality to send requests over xml as well as JSON but Apple’s certificate can’t be sent over xml so instead we worked on making it easy for the clients to send JSON requests so they wouldn’t need xml. Overall, the project will allow RadiantIQ’s clients to easily add push notifications to their mobile applications on both iOS and Android platforms. Additionally, RadiantIQ should just have to deploy the built .war file on whichever server they want to run the application on and their push notification service will be up and running.
Registering Your Application for APNs

Creating the certificate and key

1. Go to the member center (https://developer.apple.com/membercenter/index.action) and click on Certificates, Identifiers, and Profiles.
2. Choose identifiers and the click the plus key in the upper right corner.
3. Follow the instructions making sure to check push notifications in the App Services section.
4. Open Keychain Access
   2. Fill out the information requested and choose save to disk
5. Back in the member center, choose the app id you created in step 3 from the app id list and click the edit button. Click the create certificate button for the correct option (probably development).
6. Click continue and upload the signing request you made in step 4.
7. Download the certificate and double click it in keychain to “install” it.
8. Right click and export the certificate.
9. Expand the drop down and right click the private key and export it to the same directory ad the certificate.

Creating the provisioning profile

10. In the iOS Dev Center, go to the provisioning profiles section and select development.
11. Click the plus button and choose iOS development.
12. Click continue and select your app id from the list.
13. Download the provisioning profile and drag it into Xcode to add it to your app.
Creating the combined p12 file

14. You should have three files:
   - A certificate signing request (CSR)
   - The private key (p12 file)
   - The certificate (.cer file)

15. You need to convert the certificate and private key into pem files
   1. Run the following command in terminal:

   openssl x509 -in CERTIFICATE.cer -inform der -out CERTIFICATE_OUT.pem

16. Next you need to convert the private key into a pem file
   1. Run the following command in terminal:

   openssl pkcs12 -nocerts -out PRIVATE_OUT.pem -in PRIVATE_KEY.p12

17. You don't need to do anything with the CSR right now but if you keep it, when you have to renew your certificate, you won't have to create a new private key

18. Now you need to combine the two files
   1. Run the following command in terminal (you will be prompted for the key password so have it on hand):

   cat CERTIFICATE_OUT.pem PRIVATE_OUT.pem > COMBINED.pem

19. Finally, you need to convert the pem file back into a p12 file for use with our system
   1. Run the following command in terminal (you will be prompted for the password you mad for COMBINED.pem):

   openssl pkcs12 -export -in CERTIFICATE_OUT.pem -inkey PRIVATE_OUT.pem - out FINAL.p12

20. Use FINAL.p12 as the key file to send to our servers.

Note: a good reference to use if you need more information:
http://www.raywenderlich.com/32960/apple-push-notification-services-in-ios-6-tutorial-part-1