University of Dublin

TRINITY COLLEGE

Better Notifications

Patrick Corr
B.A.(Mod.) Computer Science
Final Year Project April 2015
Supervisor: Prof. Owen Conlan

School of Computer Science and Statistics
O'Reilly Institute, Trinity College, Dublin 2, Ireland
Better Notifications is a modern web application which aims for consumers to receive contextual notifications from their mobile devices.

Research contained within this report delves into Semantic Analysis and User Modelling and how these techniques could potentially be used to realise the goal of contextual notifications in the age of ubiquitous computing, and how current systems are approaching the reduction of this cognitive load on users.
Declaration

I hereby declare that this thesis is entirely my own work and that it has not been submitted as an exercise for a degree at any other university.

Patrick Corr, April 21, 2015
Permission to Lend

I agree that the Library and other agents of the College may lend or copy this thesis upon request.

Patrick Corr, April 21, 2015
Acknowledgements

I would like to express my very great appreciation to Prof. Owen Conlan for introducing this area of research to me and for his help and guidance in the development of this project.

I should also like to thank my family and friends for their continued support throughout this project and, indeed, my years studying Computer Science.
## II Design and Implementation

### 4 Design

4.1 Design Goals .............................................. 19
4.2 Importance levels / Priorities – How Users Should Receive Messages .............................................. 19
4.3 Contextual Notifications - Landing Page .................. 20
4.4 User Model Page .............................................. 20
4.5 Settings .................................................. 21
4.6 Graph View .................................................. 22
4.7 How Data is Stored ........................................... 22
4.8 Accessing Data .............................................. 22
4.8.1 API design ............................................... 22
4.9 Data Generation and Live Notifications .................. 23
4.9.1 Pinger .................................................. 24
4.9.2 Notification Generator .................................. 24
4.10 User Interface Design ...................................... 24
4.10.1 Google Material Design ................................. 24

### 5 Implementation

5.1 Pinger .................................................. 26
5.2 Notification Generator ...................................... 26
5.3 Better Notifications Backend ............................... 27
5.3.1 API .................................................. 27
5.3.2 The Database ......................................... 30
5.4 Better Notifications Frontend .............................. 30
5.4.1 Notifications Screen ................................... 30
5.4.2 Model View ........................................... 32
5.4.3 Graph View ........................................... 34
5.5 Better Notifications Users ................................ 34
III Conclusion and Further Work 36

6 Conclusion 37
6.1 Limitations 38

7 Further Work 39
7.1 Improvements to the current system 39
7.2 User Modelling 40
7.3 Semantic Analysis 40
7.3.1 Context Analysis 40

Bibliography 41

Appendix A 44
List of Figures

2.1 Android Notification Architecture . . . . . . . . . . . . . . . . . . . . . 8

4.1 Better Notifications Database Schema . . . . . . . . . . . . . . . . . . . . 23
List of Tables

A.1 Android Notifications Priority Levels . . . . . . . . . . . 45
A.2 Project Relevant HTTP Requests Methods . . . . . . . 46
A.3 Project relevant HTTP Response Codes . . . . . . . . . 46
# List of Listings

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Definition of basic endpoints in Flask</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>Flask Templating</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>Notification JSON</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>SqlAlchemy to MySQL</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>Coffeescript to Javascript</td>
<td>15</td>
</tr>
<tr>
<td>6</td>
<td>Example API request</td>
<td>23</td>
</tr>
<tr>
<td>7</td>
<td>Pinger command line arguments</td>
<td>26</td>
</tr>
<tr>
<td>8</td>
<td>API endpoints</td>
<td>28</td>
</tr>
<tr>
<td>9</td>
<td>Validation Dictionary</td>
<td>29</td>
</tr>
<tr>
<td>10</td>
<td>One to Many relationship in Python SqlAlchemy</td>
<td>31</td>
</tr>
<tr>
<td>11</td>
<td>How times are displayed on the Notifications Page</td>
<td>32</td>
</tr>
<tr>
<td>12</td>
<td>HTML form</td>
<td>33</td>
</tr>
<tr>
<td>13</td>
<td>OpenID Providers</td>
<td>35</td>
</tr>
<tr>
<td>14</td>
<td>Short example of notifications sent by Notifications Generator</td>
<td>44</td>
</tr>
<tr>
<td>15</td>
<td>Random Colour Generator</td>
<td>47</td>
</tr>
</tbody>
</table>
Part I

Introduction
Chapter 1

Introduction

1.1 Introduction

A study by Telephonica Research showed a sample set of 15 people received on average 63.5 notifications per day and within minutes each of these notifications had been viewed, regardless of the persons context and whether the phone was on silent or not Pielot et al.[1].

This massive number of daily notifications can be termed 'Notification Overload' and causes important messages to be lost which, if it had arrived alone, could have been acted upon.

This report documents the design and implementation of an intuitive way to increase the arrival of contextual notifications and decrease the arrival of non contextual notifications.

Better Notifications aims to realise this goal using generated data and a web based application.

This report documents a project that should be considered a Proof of Concept. One which models a likely architecture which would be used by a system such as Android or the Apple iPhone. The project is built using modern software technologies and I will explain how each of these technologies fits together to make a very robust and scalable application.

1.2 Readers Guide

This chapter aims to give the reader an idea of the space that this project aims to fill

Chapter 2 gives an overview of the state of the art, and the background research that has been taken into consideration for this project.
Chapter 3 contains the design of the application as a whole without delving into the particulars, instead keeping focus on a more conceptual view of the system.

In Chapter 4 the systems implementation and a discussion of the choice of system architecture are described.

Chapter 5 will evaluate the usability of this project in both a quantitative and a qualitative manner. Both of these directions will be described in detail.

Chapter 6 gives the conclusion to my implementation and design and also how successful the research was.

Chapter 7 discusses the possible further work that could be undertaken to expand on this project, or implement it in industry.
Chapter 2

State of the art and Motivation

The constant stream of notifications we receive all day has caused many people to simply turn off these notifications and ignore all incoming messages. This is a contrast to those of us who receive a loud ring from our phones or computers every 10-15 minutes. In the former case the user receives all notifications at some point (tests show that they will still view the notification within minutes) but the notification may no longer be important by the time it is read. This is in stark contrast to what Weiser M. [2] foresaw when he discussed invisible ubiquitous computing\(^1\) in his paper "The computer for the 21st century".

In the latter case the user receives all notifications, however, the important messages can arrive within an onslaught of other messages that have no importance at the current time.

This is a big problem that seems to have no current solution.

This is exasperated by the sheer number of applications that are vying for a consumers attention. There is no awareness of an applications neighbouring applications and how these other applications want to get the attention of the user. This results in notification overload.

Further more this overload is causing users to pay more attention to their notifications than to the work at hand. A number of cases of this are discussed in Cutrell et al. [3]

There have been movements in the last year to increase the invisibility of our consumer products - think apple watch and wearable tech - the idea is to help the user when wanted, not when prompted.

Because of these problems a few systems have been released which will be discussed in this chapter. They are; Google Inbox, which is an

\(^1\) Ubiquitous computing is computing that is all around us or with us at all times.
email aggregator which processes messages before delivery so as to give user the emails in a categorised fashion; and Apples iPhone Do Not Disturb system, which offers users the ability to turn off notifications for their work day, it also allows certain phone calls and text messages through from ‘favourite’ contacts and repeated calls to your number.

Both are movements in the correct direction. They are not, however, powerful enough to ensure a user gets notifications that are important to them, and not receive notifications that are not important to them.

Research into Semantic Analysis and User Modelling will also be discussed in this chapter, as these areas are of great interest to how this research occurred and also to the future of this research.

2.1 Google Priority Inbox

Google Priority inbox is a “feature of Gmail that ranks mail by the probability that a user will perform and action on that email”[4]. It builds a per-user statistical model which is updated as frequently as possible.

The motivation behind this feature is that users receive tens or hundreds of emails a day, not unlike the motivation of this research. It states that this is not a new problem and cites Dabish et al. (2005)[5] for previous work on predicting an action on a message. Dabish et al. state that Importance has a modest impact on the likelihood that a user will make an action on an email, and also that on average a user kept half of all emails in their inbox and classified one third of these emails as important.

The Priority Inbox aims to perform this analysis without explicit labelling by the user, and based on a threshold a binary Important or Not Important is set for that email.

How this work can translate to Notifications will be discussed in Further Work section-7.3

2.2 Google Inbox

Google Inbox is a new product by Google that can be opted into by a Gmail user, after said user has been invited to join.

Google Inbox performs semantic analysis and smart processing of a Gmail users emails, categorising the users emails into topics such as Travel, Promotions, receipts etc.[6]
This means that while a user still gets notified as much as usual, they have easier access to important emails. It also give the option to dismiss and be reminded about emails that are received. [7]

The number of users that are in the Google inbox beta could not be shared at the moment [8]. However, new users are invited daily.

### 2.3 Apple Do Not Disturb

Do Not Disturb is accessed in the settings application in the iPhone and IOS devices. Using this product a user can define times when they are at work and do not want to be distracted. During this time only the consumers 'Favourites’, which a user sets, can call or text through to the user, and the user will be notified. The users phone will also ring if there is 'repeated call’ by an individual. [9]

### 2.4 Personalisation, User models and Personas

User models are abstract representations of user properties including their needs, preferences, knowledge, as well as physical, cognitive, and behavioral characteristics (Benyon and Murray)[10]. In a software system these are all represented by variables and a particular user model is instantiated by a particular declaration of these variables, Mohamad and Kouroupetrouglou (2014)[11]

Personas are fictional people or groups of people that are used to aid design of a product with these people in mind. The Persona may be "Computer novice" or "Computer savvy" etc. Pruitt and Grudin (2003)[12]

User models and Persona are related in that a Persona is a tool for a designer to model a specific target group. User models are a tool that aim to do the same thing but from a machines point of view. (Aquino and Filgueiras) [13]

The 2008 ChoiceStream Personalization Survey states 78% of consumers are interested in receiving personalized content[14]. This area has yet to be tapped in mobile notifications as a whole, only by the applications sending the notifications.

Kay J. (2006)[15] discusses the Scrutability of a User Model. In which, unlike Googles Priority Inbox, Users are given the ability to view and
update their own user model and the effect this model has on the user’s experience.

2.5 Sentiment analysis and Natural Language Processing

Sentiment analysis is the use of Natural Language Processing, and Semantic analysis to discover the subjective information in a body of text, Deerwester et al. (1990)[16].

It is used to group separate bodies of text together towards the concept of those bodies of text. Common sentiment analysis tasks, Cambria et al.(2013)[17], can be how happy or sad a body of text is, or how angry or pleased it is, or Politics, as in one of the first papers that researched this topic by Carbonel J. G. (1979)[18].

The idea of sentiment analysis grouping polar ideas such as, as Carbonel discussed, Liberal or Conservative ideologies gave motivation that the same could be done for important or unimportant notifications.

2.6 The Android Notification API

To perform this research without a taking into account of how notifications occur on the most popular smart phone operating system would be less beneficial research.

Figure-2.1 shows visually how notifications arrive in an android system, with the Better Notifications system placed as an reference to where this system would fit in.

This is based on the Android Developer Documentation.[19]

The Android Notification API has the facility for each notification to contain a "Priority" between 2 and -2 [20]. The table of notification priority is stated in Appendix A Table-A.1. These priorities define the action to be taken in the android system, and are set by the application sending the notification.

[^2]: http://www.idc.com/prodserv/smartphone-os-market-share.jsp
2.6 The Android Notification API

Figure 2.1 Android Notification Architecture
Chapter 3

Overview of modern web technologies

Better notifications has been built on a modern web stack. This chapter will discuss this stack so as to make this document accessible for anyone without a background in this area, this will also help the discussion of Design in Chapter 4 and the Implementation in Chapter 5.

The following list gives a run down of the stack and a road map to the rest of this chapter.

• Backend Technologies
  – Python
  – Flask
  – sql-alchemy
  – sql-lite
  – restful-architecture

• Frontend
  – Coffeescript
  – Sass
  – Html
  – Backbone.js

• Infrastructure
  – Heroku
3.1 Backend Technologies

3.1.1 Python

Python [21] was a good choice for this project for many reasons including

- It is a well known industry standard language that can be translated or even compiled into any language that a future implementer would like to use.

- Its robust and powerful, Python has a relatively small number source code lines, meaning it is less likely prone to issues, easier to debug and maintainable.

- Its libraries are well documented and intuitive to use.

3.1.2 Flask

Flask [22] is a Python "micro-framework". A micro-framework is one that gives only the bare essentials that are needed to write, in this case, a server.

Using a micro-framework like Flask means that the programmer has more control over what is contained in their project, and also that anything that they want to include must be explicitly connected and included.

This is in contrast to the other main python web framework, Django[23], which is a fully featured large framework, and gives the programmer all the components that they might want to build an application, regardless of whether these things are used or needed.

The most simple Flask application contains nothing but the creation of an application and a URL in that application that returns some HTML or other resource to the requester.

The code in Listing-1 show how once the server is accessed at the URL in the @app.route function, the user is then checked to see if they are logged in, the function index() is called, then the server will return the HTML stored in the file chosen by the render_template function.

The render_template function will compile the template that the file points to. An explanation of a template file is given in Listing-2.

In this example template file, the content stored in {% block content %} will replace the same block in the index.html and the HTML that will be sent to the requester will be the new constructed HTML.
3.1 Backend Technologies

```python
@app.route('/
@login_required
def index():
    return render_template('main.html')

@app.route('/user/
@login_required
def user():
    return render_template('user.html')
```

Listing 1 Definition of basic endpoints in Flask

```html
<!-- start index.html -->
<!doctype html>
<html class="no-js" lang="en">
<body>
    <div class="container">
        {% block content %}{% endblock %}
    </div>
</body>
</html>
<!-- end index.html -->

<!-- start main.html -->
{% extends "index.html" %}
{% block content %}
    <div id="notifications">
        <div class="notifications-header">
            <span>Contextual</span>
        </div>
    </div>
{% endblock %}
<!-- end main.html -->
```

Listing 2 Flask Templating

This makes it painless to create an application that has the same overarching design on all of its pages as each separate page is inserted into a block in the layout page.

Flask was chosen for this project as it is easy to scale to any number of running servers and can easily be configured. It is also very quick to get
applications running on flask, and there are many great libraries for the Restful design aimed for by this project.

### 3.1.3 API

An API (Application Programming Interface) is simply a specification of remote calls exposed to users, consumers or other loosely coupled applications so that these users can access data from somewhere, in this case: a Database.

### 3.1.4 REST and HTTP


The HyperText Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems.[25]

The key principles of REST involve separating your API into logical resources. These resources in the context of this project are Applications, Notifications, CustomTimes, and Overrides.

These resources are then manipulated using HTTP requests where the methods GET, POST, PUT, DELETE have a specific meaning. Each request returns a specific response code depending on how the server handled the request. Appendix A Table-A.2 and Table-A.3 describes the main HTTP requests and response codes used in this project.

An example of how these HTTP methods are mapped is:

GET user/1/applications - retrieves a list of all applications associated with that user

GET user/1/applications/1 - retrieves the users application associated with the applications id at the end of the url

POST user/1/applications - create a new application for the user

PUT user/1/applications/5 - Update application 5

DELETE user/1/application/5 - Delete application 5

REST and HTTP is industry standard for building APIs today. Using this API design means that this project can easily be translated into another systems code style, ie. Android.
3.1 Backend Technologies

3.1.5 JSON

JSON[26] is JavaScript Object Notation and is used in the API to send and receive resource data.

An example of a notifications JSON is shown in Listing-3

```
{ 'application': 'Twitter',
  'data': None,
  'datetime': '2015-04-03T17:17:24+00:00',
  'done': True,
  'id': 241,
  'notify_time': '2015-04-03T17:17:24+00:00'}
```

Listing 3 Notification JSON

Using JSON we can send and receive the entire state of a resource every time we perform a request. This means the server is 'Stateless', meaning that a request to the server is the only way for a resource to change state. User state is held only be the client, making the server faster and more secure.

3.1.6 SqlAlchemy

SqlAlchemy[27] is an Object Relational Mapper (ORM) for python and also a general database toolkit.

An ORM maps Python Class objects to relational database tables.

Consider a class Applications which has the class variables ID and Name. This will translate to creating a table "Applications" in the database schema, with columns 'ID' and 'Name'. The entire schema can be built and connected in this way.

Code can then be written in a Pythonic way to access databases, which does not use the same keyword arguments as traditional SQL.

As an example view the database requests in Listing-4 from the Better Notification application as both SQL and SqlAlchemy
# python#
```
times = DBsession.query(CustomTime).filter(
    CustomTime.user_id == uid).order_by(
    CustomTime.start_time
).all()
```

# mysql#
```
SELECT * FROM customtimes WHERE user_id = <id>
    ORDER BY start_time;
```

Listing 4 SqlAlchemy to MySQL
3.2 Frontend Technologies

The frontend is the client side code, the code that is downloaded when a user goes to the Better Notifications URL.

The frontend of my application was heavily influenced by the design of Google Inbox. It uses Javascript and CSS that are compiled by the languages CoffeeScript and Sass.

All this is centered around working with a library called Backbone.js which makes live updating of the system quick and easy and also makes it cleaner to access restful architectures, like the Better Notifications server.

3.2.1 Javascript

JavaScript[28] is a programming language, based on the Ecmascript standard, and is used to make web pages interactive. It is built into web browsers.

3.2.2 Coffeescript

Coffeescript[29] is “a little language that compiles into Javascript”.

An example of Coffeescript and the Javascript it compiles into is show in Listing-5.

```javascript
// CoffeeScript //
opposite = true
number = -42 if opposite

// JavaScript //
var number, opposite;
opposite = true;
if (opposite) {
    number = -42;
}
```

Listing 5 Coffeescript to Javascript

Using Coffeescript was a design choice in this system to use the most up to date technologies.
3.2.3 CSS

CSS[30] is a style sheet language used for describing the look and formatting of a document written in a markup language, in this case HTML.

3.2.4 Sass

Sass[31] is a CSS extension language that adds variables and nested styles to standard CSS.

When compiled it is the exact same as hand written CSS, ie. there is no performance upgrades, however it is a lot cleaner to write the CSS through Sass as brand colours and common styles can be shared by multiple elements easily.

3.2.5 Backbone.js

Backbone.js [32] "gives structure to web applications by providing models with key-value binding and custom events, collections with a rich API of enumerable functions, views with declarative event handling, and connects it all to your existing API over a RESTful JSON interface."

Backbone offers a lightweight way to structure Web applications that involve lots of Javascript. In Backbone data is expressed as Models, which can be created, validated, destroyed and synced with a server. When a model is changed an event, or function, is called on the model and this can cause the ‘view’ (what the user sees) to change.

These models are all grouped together in what’s called a Collection. This Collection is the list of resources taken from the API.

3.3 Infrastructure

The Better Notifications application is designed to be scalable and easy to deploy. To ease in that a system called Heroku is used.

3.3.1 Heroku

Heroku is a platform as a service (PaaS) that enables developers to build and run applications entirely in the cloud.[33]

To push the app onto the Heroku server we use the version control system Git [34] and use Heroku as our server for the source code. When
the code is deployed it is built by following a set of steps defined by the developer and within minutes the application is running on a Heroku server.
Part II

Design and Implementation
Chapter 4

Design

4.1 Design Goals

The goal in this design is to ensure that a user can receive their notifications when wanted, and also be able to view a backlog of the notifications and see, live, how the changing of their settings, their user model will effect the reception of their notifications.

To this end the design needs to cover each of these design goals:

1. The user must be able to view and be notified of their contextual notifications.
2. The user must be able to view a sample set of their previous notifications.
3. The user must be able to add, remove and update Applications, Custom times, and Overrides easily.
4. The effect of changing these settings should be immediately visible.
5. User should be able to set "Holiday time", which is non repeating and longer than 1 day, and priority for this.

4.2 Importance levels / Priorities – How Users Should Receive Messages

The initial steps in the design of this system are to correctly identify the how the user would like to receive notifications.
This system uses a Importance Level or Priorities system to define a threshold above which users will receive notifications, this is based on the Google Priority Inbox and Android Notification priorities.

The priority system was designed to be as user friendly and simple to set up as possible. To achieve this simplicity the only acceptable form of priorities was to set integers as the priorities and the threshold.

As notifications come through, the priority of the application sending is attached to the notification, this priority is then compared to the priority level of the current and future times. When the notification is of a higher precedence than the time, the user is notified.

4.3 Contextual Notifications - Landing Page

The main function of this application is to simulate a notification experience for a user. For this reason the front, or index, page of the web application should contain, and be focused on, the contextual notifications that a user receives. This should be set out in the same style as the Android notifications system, which would be a good example of how most people receive notifications.

Since we want only the contextual notifications for a user, this page has only one table, which contains all notifications that have not been dismissed and that have a time that is past the current time, ie. no notification that has a notify time in the future will be shown.

The displayed time is offset from the current time as is want in current systems eg. a notification with a notify_time of 09:00 at the time 09:15 will display a time of 15 min’s.

This page is of a clean design to keep focus on the important data, the notifications. The logic and other background information should not be shown on this page.

4.4 User Model Page

In contrast to the previous section, the User models page should contain as much information as possible that can help a user to view the status of their current setup. This screen is analogous to the settings screen in an Android system, where this information would be set if this system were implemented for Android.
There are two running applications on this page. The Graph View (Section-4.5) and the Settings View (Section-4.6).

4.5 Settings

There are 3 main resources that a user has control over customising: Applications, Custom Times, and Overrides.

Accessing these values over the API should be simple, and since the API will be designed in a Restful way this should not be a problem. Implementing these end-points on the frontend however, needs some more work. Since each of these resources are quite different they cannot have the same frontend implementations, however, they must still form a uniform user experience.

Applications

Applications are the applications that are being monitored by the better notifications system. These are initially set by users.

Each application has a priority level from 1 to 7, with default being 1.

Custom Times

Custom Times are set by a user to specify times, eg. Working, Nighttime, Holidays. These should repeat daily and so should only receive a start time and end time without dates.

If a number of ‘days’ is set then the custom time is considered a holiday and notifications below that holidays priority will not come through.

Overrides

Overrides consist of keywords that prompt a previously low level notification to pass through the filter. While overrides in this way are simple, they need to have a sensible API and database schema, as the implementation of them with further work into semantic and context analysis would need to perform quickly.
4.6 Graph View

Users need to be able to see the effects of how they change their model. Enough of a time scale should be available, and currently this is set to a week.

The graph must show how users received notifications over a period of time so that a user can view and reflect on what times inappropriate messages came in. As a user tweaks their user model these notifications should shift and organise themselves into a layout that suits the model.

There was a wide range of libraries that could be used to implement this feature, since this is only one part of the project that needed a graph engine, I chose one that was relatively light weight and customisable.

The simplest and most user friendly design for this graph view is a simple time-line.

It is very important that all notifications for the current setting should be viewed and traversed as easily as possible and this is possible with the Vis.js library.[35]

4.7 How Data is Stored

The database schema, simply, consists of users where each user is the one in a one to many relationship with, Applications, Overrides, Times and Notifications. Applications are then the one in a one to many relationship with Overrides.

The database schema is shown graphically in Figure-4.1

4.8 Accessing Data

The data in the database is always accessed via the API which provides a wrapper that can be used by any other application. This provides security and validation over the data being accessed in the database.

4.8.1 API design

If you view the Database schema from left to right, the API endpoints can be deduced simply. This design makes it obvious both how the database schema is implemented and how to access certain values in the database.
4.9 Data Generation and Live Notifications

To simulate the kinds of notifications that would be received by users two other pieces of infrastructure separate from the Better Notifications code base will need to be built. In keeping with the Android design, Better Notifications should simply accept notifications and display them to the user after being updated. This means that the generator and action of sending notifications should be entirely separate, not unlike Google Cloud Messaging Service \cite{36} or Apple Push Notifications. \cite{37}

---

\footnote{Curl website: curl.haxx.se}
4.10 User Interface Design

4.9.1 Pinger

Pinger is a command line application that sends a ‘Ping’ or a GET request to a URL every N milliseconds. Pinger is used to prompt a Notification Generator to generate a new notification and receives a response when this is done.

Using Pinger means that the Better Notification application can receive Notifications in the same form as would be received on a mobile device.

Setting it to ping continuously at a random time between 5 and 15 minutes simulates a usual day of notifications. Pielot et al. (2014)[1]

4.9.2 Notification Generator

Notification Generator, when prompted, will create a notification for a random or specified application and based on the application the body and main text of the notification will be created and then sent to the Better Notification system.

The generated messages are formed from data taken from example application notifications to keep them relevant. A snapshot of these notifications is in Listing-14 in Appendix A.

These application messages are stored in a Python dictionary file. Some of these messages contain language of urgency text such as "emergency". This is important so that user specified overloads can be fired.

4.10 User Interface Design

On an android or apple device the design of the interface would obviously be set as per the companies design guidelines2,3.

In keeping with the Android design I consulted the Google Material Design guidelines. [38]

4.10.1 Google Material Design

Google’s material design sets a template for how Android applications, in particular, should be designed. This also expands to cover web applications and general design.

---

2 Apple IOS design: "developer.apple.com/library/ios/documentation/UserExperience/Conceptual/MobileHIG/"
3 Android Design Guidelines: developer.android.com/design/index.html
Based on bold and contrasting colours an entire framework for the design of your app or web site can be created.
Chapter 5

Implementation

There are 3 parts to the Better Notifications system. These are:

1. Pinger
2. Notification Generator
3. Better Notifications

5.1 Pinger

Pinger is implemented as a Python command line application. It can be run with a command in the form stated in Listing-7.

```bash
./pinger.py -s <server_address> -p <server_port> -d <data>
```

Listing 7 Pinger command line arguments

Pinger is a continuously running application which calls `time.sleep` for N seconds at a time and after this time performs a GET request to any server and endpoint that is specified. It has very good performance as it it built in python and the `sleep` action suspends the thread for the time specified. This means that polling or continuous computation are not needed.

5.2 Notification Generator

The Notification Generator is an application that is built to act like the Android System and its many applications sending notifications. The
system has 3 endpoints, all of which generate one or many notifications and sends them to the Better Notifications notifications endpoint.

The Notification Generator uses the Python HTTP module, Requests [39]. We perform every POST request to the Better Notifications server using this library which creates a HTTP request from python dictionary’s and objects.

The POST requests that the Notifications generator performs are the same as ones that can be written manually, or ones that the Better Notifications frontend performs. But in this case, the data is randomly generated from data contained in python dictionaries in the samples module.

5.3 Better Notifications Backend

5.3.1 API

The Better Notifications API is built using Flask and the Flask library Flask-Restful.

The code for the API is in a file called ‘resources.py’. There are two python classes for most resources. The first is a class for individual resources and the second is for list resources that contain many resources at one endpoint.

There are three resources that can be accessed by the API, they are Notifications, Times and Applications.

The endpoints for the API are all defined in app.py using the code:

The code shown displays that each endpoint is assigned a Class which is defined in resources.py, a string which identifies the resources URL with any arguments to the function surrounded by <>s and given a type, and a name for the endpoint, which flask can use as an identifier internally.

The 4 HTTP verbs, GET, POST, PUT, and DELETE, can be used to access all endpoints. The following list explains the typical order of code for each method.

For each endpoint:

1. A GET request will access the database and fetch data from it, return a HTTP error code 404 if a resource does not exist, then "marshal" or bundle together the fetched data into a JSON object and return this JSON to the requester.
api.add_resource(NotificationListResource,
    '/user/<string:uid>/notifications',
    endpoint="notifications")
api.add_resource(NotificationResource,
    '/user/<string:uid>/notifications/<string:id>',
    endpoint="notification")
api.add_resource(TimeListResource,
    '/user/<string:uid>/times',
    endpoint="times")
api.add_resource(ApplicationListResource,
    '/user/<string:uid>/applications',
    endpoint="applications")
api.add_resource(ApplicationResource,
    '/user/<string:uid>/applications/<string:id>',
    endpoint="application")

Listing 8 API endpoints

2. A POST request will parse and validate (ensure the data is in the correct format) the data sent by the requester, build an object relevant to the endpoint with this data, add the object to the Database section and then commit the Database transaction. The resulting object will then be marshaled and returned to the requester.

3. A PUT will fetch a particular object from the database, return a HTTP error code 404, if it does not exist, parse and validate the requester’s sent data, update the object with this data, add it to the database, save the database transaction, then marshal the updated object to be sent back to the requester.

4. A DELETE request will simply find if a row defined in the endpoint is in the database, and if it is, it deletes it, and commits the database transaction. A delete returns no data except a HTTP response code.

Since this list explains how most endpoints work and are accessed, the next subsections will discuss interesting implementation details of some resources and the API as a whole.
5.3 Better Notifications Backend

Notifications

The Algorithm used by Better Notifications is contained within the Notifications List endpoint. For this reason all other tables need to be accessed by this endpoint.

The algorithm is as shown:

1. Access the overrides and decide if this notification is overridden. If it is then skip all next steps and notify.

2. Find the applications priority by accessing the application database by the applications name.

3. Go through each defined custom time. move the notifications notify_time on until it is of higher priority than a time, or there is no time defined. This also checks holiday days.

Marshaling and Validation

An example of a validation dictionary is shown in Listing-9. Since the HTTP request and data that comes to the API is string data, a way of ensuring the data fits into the database schema is needed. The example validation takes strings and ensure that they can be translated into the python types Integer, String, and DateTime. It also gives a list of the keywords that can be used in the request, ie. id, application, data, etc.

```
notification_fields = {
    'id': fields.Integer,
    'application': fields.String,
    'data': fields.String,
    'done': fields.String,
    'datetime': fields.DateTime(dt_format='iso8601'),
    'notify_time': fields.DateTime(dt_format='iso8601'),
}
```

Listing 9 Validation Dictionary

This same data is used to marshal a response into JSON, and models the shape of the JSON object.
5.4 Better Notifications Frontend

### Times, ISO 8601 and Unix Time

Since times are an integral part of this project the implementation of them in this project should be discussed.

In the above section (Section-5.3.1) the validation field contains a \texttt{dt\_format} of \texttt{iso8601}. ISO 8601 \cite{iso8601} is an international standard for the representing dates and times in numbers. An example date and time is 2015-04-19T12:59:23+00:00 where the month is before the T, the time after the T, and the timezone at the end.

Unix time or Epoch time is the number of seconds that have passed since Thursday, 1 January 1970 at 00:00.

An example of translating Unix time to ISO 8601:

\begin{verbatim}
\end{verbatim}

Unix time is used a lot in the Better Notifications system as time is represented as integers, meaning addition and subtraction of times is painless. This Unix time can then be translated to ISO 8601 time with ease.

### 5.3.2 The Database

The Database schema is defined in the Design chapter Section-4.1.

To build the database a Python framework called SqlAlchemy was used. Using this language the table relations and foreign keys are defined in the python code.

An example of how the Users table and the applications table are connected in a one to many relationship is shown in the Listing-10. The \texttt{relationship} in the User class means that the table applications can be accessed from the class User. This would not be easily possible in pure SQL development. The \texttt{backref} means that by performing Application.creator we can get the users name from each application. This relationship object makes it easier to access related data in tables.

### 5.4 Better Notifications Frontend

#### 5.4.1 Notifications Screen

The main screen on the Better Notifications applications contains only the user’s contextual notifications.
class User(Base):
    __tablename__ = 'users'

    applications = relationship('Application',
        backref="creator", lazy='dynamic')

class Application(Base):
    __tablename__ = 'applications'

    user_id = Column(Integer, ForeignKey('users.id'))

Listing 10 One to Many relationship in Python SqlAlchemy

Most of the computation is done on the server side, so when a user accesses this page the Coffeescript Collection is called to retrieve its data from the API list endpoint it is assigned to. This is the NotificationsList endpoint. The Coffeescript Render function is then called on the collection.

When Render is called on the collection it, in turn, calls the render function on each of the Notifications that are contained in the Collection. The Notification then gets displayed if it has not been marked completed or if it has a notify time past the current time.

To be rendered to the page, the empty table, which will house the applications, is selected in Coffeescript by jQuery[41], which is a simple Javascript library that makes . jQuery selects this table by its 'id' which is a CSS Selector¹ we then append the notifications to the table by a HTML template that we define in the templating language and Javascript library, underscore.js[42], which work like the Flask Templates described in Section-2.

The result is that the table in the centre of the screen is updated with a list of contextual notifications.

To render the notifications the time since notification must be computed, the table is then organised by this time. We use the code in Listing-11 to

¹Each element in a web page can contain one or more CSS selectors. These are used to specify the style on this particular section of HTML. These can be Classes or IDs. Class selectors can appear often in a page and gives the same style to each time an HTML element has this Class. An ID con only occur once on a page, and so when we select a HTML element by an ID selector, we know that we have a particular element. Using these selectors we can append more HTML inside these elements.
do this. This shows only days if the notification has been there for more than a day, the same design is realised in hours and minutes.

```javascript
var d = new Date(datetime);
var now = new Date();
ago = (now.getTime()-d.getTime()).toString();
ago = ago/1000;
seconds = Math.floor(ago % 60);
ago = Math.floor(ago/60);
mins = ago % 60;
ago = Math.floor(ago/60);
hours = ago % 60;
days = Math.floor(ago/24);
if (days >= 1) {
   print (days + " dy ago");
} else if (hours >=1) {
   print (hours + " hr ago");
} else {
   print (mins + " min ago");
}
```

Listing 11 How times are displayed on the Notifications Page

### 5.4.2 Model View

The models part of the user screen is more complicated to populate than the notifications screen as not only will the user want to see their Applications, Overrides and Times, but they will want to perform CRUD operations on them (Create, Read, Update and Delete). To do this we need some more Coffeescript functions. These functions will essentially be the same for each resource.

To implement the Creation of a resource 4 steps are necessary

1. A button the user presses to create a new resource.
2. A form that when this button is clicked is added to the Web page at the foot of the table
3. To update the collection and render the new resource.
4. Changes must be persisted to the database.
To perform this an HTML button is added to each table, and assigned a CSS id selector. A 'event listener' is then added to this HTML button.

An Event Listener is a Javascript/Coffeescript function which is called when an event happens. This event could be the pressing of a button or the hovering of a mouse over a HTML Element.

In this case when the HTML button is pressed we attach a HTML form to the bottom of the table, an example of which is shown in Listing-12.

```html
<form id="time_form">
  <input type="text" name="name"
         id="time_name" value="" />
  <input type="time" name="start time"
         id="time_start" value="" />
  <input type="time" name="end time"
         id="time_end" value="" />
  <input type="text" name="Importance"
         id="importance" value="" />
  <input type="button" value="Submit"
         id="createTime">
</form>
```

Listing 12 HTML form

This form allows the user to specify a new custom time, with the type of each entry being specified in the type field. When the Button specified in

```html
<input type="button" value="Submit" id="createTime">
```

is pressed another event listener is called, which will perform the saving of this resource to the database are rendering it on the front end.

When submit is pressed:

1. A new object is created with the fields specified in the form.

2. The render function for this object is called, appending the object to the table.

3. The object is added to the Backbone Collection.

4. The save function is called on this object. Backbone takes the fields of the object, and sends them to the API the same way the API is accessed from other applications.
5.4.3 Graph View

The Graph is built using a Javascript Library called Vis.js. This has been customised to give random colours to each of the items on the time line.

The graph view first gets all notifications into a collection, the same way that the Notifications Screen table does, however, instead of rendering these Notifications into HTML, they are added to a Javascript array that can be parsed by Vis.js.

The custom times that have been specified by the user then need to be fetched and put into a collection too.

Vis.js then takes all of these notifications, and times and places them onto the graph, depending on their notify times, and the times that have been specified.

To colour the nodes on the graph a different colour but close to the brand colour of blue in Better Notifications a quick small random colour generator was written which is showed in Appendix A Listing-15

5.5 Better Notifications Users

Users are implemented using the OpenID[43] of any client which the user decides to use, Yahoo, Google, LinkedIn etc.

Using OpenID is a relatively painless procedure as user passwords do not need to be handled by the Better Notifications application.

When a user first accesses the application they will be re-routed to a login page. This page contains only a text field and a checkbox to be kept logged in. When the user types in their OpenID and clicks submit they are then re-routed to the OpenID provider to login to their system. Once this is done the user is finally redirected to Better Notifications and they can now use the application.

If this user was not previously a in the user database table, they will be added after a successful login.

When the user logs in, their details are saved to a Flask global variable called ‘g’. This variable contains the users logged in state and is accessible to the Flask templates, thus it is accessible to the Javascript code.

A list of acceptable OpenID providers is displayed in Listing-13.
5.5 Better Notifications Users

```python
app.config['OPENID_PROVIDERS'] = [
    {'name': 'Google', 'url':
        'https://www.google.com/accounts/o8/id'},
    {'name': 'Yahoo', 'url':
        'https://me.yahoo.com'},
    {'name': 'AOL', 'url':
        'http://openid.aol.com/<username>'},
    {'name': 'Flickr', 'url':
        'http://www.flickr.com/<username>'},
    {'name': 'MyOpenID', 'url':
        'https://www.myopenid.com'}]
```

Listing 13 OpenID Providers
Part III

Conclusion and Further Work
Chapter 6

Conclusion

The research contained in this report proposed to investigate a better and more contextual way of users receiving notifications, designed in a way that would allow the system to be integrated into an existing system and also be extended with new technologies such as Sentiment Analysis and User Modelling. A successful implementation in this regard is one which can easily be implemented into an existing system and one that can show obviously the benefits of having contextual notifications through user updated models.

On testing the application it was found that when overrides, applications and times were defined the graph updated accordingly to show the notifications do come through depending on the importance level that was deduced. Also in the live notifications feed, only notifications that were context sensitive were shown and notifications that have been dismissed were not shown. This indicates that the essential part of the project worked and that if this project was implemented in this way on another system, there would be no extra logic to be implemented.

This system used explicit user defined models. In research conducted among user testers, this was deemed to be sufficient, as many users wanted only to stop notifications from some applications coming in during the work or college day, and during holidays. The users felt that, while the system would benefit from machine generated user models to automatically stop notifications coming in, they did not feel it necessary for the system to do this for it to be marketable.
6.1 Limitations

The need to input all overrides for each application is not user friendly. It is also not robust as the override is based on key words in a notifications body. This means that if an overrides key word was the word "emergency" any notification that comes through with that word will be of the highest priority. This is discussed in further work.

The current implementation is simply a web interface and so a definite limitation is that the notifications are generated and are not modelled on how each different user received notifications.

Pages are not live, this is a definite limitation as the graph does not update live when the resources are changed. Users discussions showed that this was unintuitive.
Chapter 7

Further Work

To expand on the current solution that i have built there are a few simple features that can be added, which I discuss in section-7.1. There is also the addition of User Modelling and Semantic analysis which i have discussed throughout this report.

7.1 Improvements to the current system

All applications should be defined when a new notifications is received from an unknown application. This is a simple feature that could be implemented easily on the server, where the Name of the notifications Application is searched for in the database. This is quick as the application name is an index in the application table and would only marginally reduce the performance of the system.

The main Notifications page and the graph view on the models page are not live and do not show notifications as they arrive. To create this performance is not trivial, it is however possible to do so using Web sockets, which keep a port open to the server while the page is open. The page could also poll the server to receive new information as often as every 30 sec/1 min. Both of these solutions increase load on the server, and would not be necessary if implemented in an Android system.

Notifications are not generated depending to a particular user. The obvious solution to this is to give the notifications generator a snapshot of previous notifications to model all subsequent notifications.
7.2 User Modelling

User modelling should be researched. In the same way that Googles Priority Inbox defines importance for each user as a threshold above which something is important. Implementing User modelling was kept in mind during the design of this system and was a major motivation toward the end product. All resources could easily be expanded to keep custom user data.

7.3 Semantic Analysis

Semantic Analysis is discussed in my background reading. Using semantic analysis the overall importance could be inferred from each notification.

7.3.1 Context Analysis

Context Analysis using semantic analysis of a set of notifications would mean that, as messages come in, any message that stands out from a set of previous notifications from the same application or sender could potentially stand out from these other notifications and heighten the notifications importance.
Bibliography


A BibTex file is provided.


[38] Google design guidelines.


Appendix A

```python
apps = {
    "Dropbox": ["Paddy has updated the folder 'important/stuff'"],
    "Gmail": ["Paddy: Hey! ", "Mom: hi!", "Bro: Emergency, please call"],
    "Facebook": ["Message from Tony: lorem ipsum", "Ciara Maguire and Eleanor McSweeney posted in Portuguese Route!", "Sean Treacy and Brandon Boyle have their birthday today.", "Jaye Lowe, Niamh Hennessy and 4 others posted in VDP.", "Mnn Rdmnd and Eoin O'Dwyer posted in Santiago 2015"],
    "Twitter": ["@nsno: ugh", "@shayan: hey buddy"],
    "Viber": ["Stephen: hey, you around tonight?"]
}
```

Listing 14 Short example of notifications sent by Notifications Generator
<table>
<thead>
<tr>
<th>Priority</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX</td>
<td>Use for critical and urgent notifications that alert the user to a condition that is time-critical or needs to be resolved before they can continue with a particular task.</td>
</tr>
<tr>
<td>HIGH</td>
<td>Use primarily for important communication, such as message or chat events with content that is particularly interesting for the user. High-priority notifications trigger the heads-up notification display.</td>
</tr>
<tr>
<td>DEFAULT</td>
<td>Use for all notifications that don’t fall into any of the other priorities described here.</td>
</tr>
<tr>
<td>LOW</td>
<td>Use for notifications that you want the user to be informed about, but that are less urgent. Low-priority notifications tend to show up at the bottom of the list, which makes them a good choice for things like public or undirected social updates: The user has asked to be notified about them, but these notifications should never take precedence over urgent or direct communication.</td>
</tr>
<tr>
<td>MIN</td>
<td>Use for contextual or background information such as weather information or contextual location information. Minimum-priority notifications do not appear in the status bar. The user discovers them on expanding the notification shade.</td>
</tr>
</tbody>
</table>

Table A.1 Android Notifications Priority Levels
The GET method means retrieve whatever information (in the form of an entity) is identified by the Request-URI. If the Request-URI refers to a data-producing process, it is the produced data which shall be returned as the entity in the response and not the source text of the process, unless that text happens to be the output of the process.

The POST method is used to request that the origin server accept the entity enclosed in the request as a new subordinate of the resource identified by the Request-URI in the Request-Line.

The PUT method requests that the enclosed entity be stored under the supplied Request-URI. If the Request-URI refers to an already existing resource, the enclosed entity SHOULD be considered as a modified version of the one residing on the origin server.

The DELETE method requests that the origin server delete the resource identified by the Request-URI.

<table>
<thead>
<tr>
<th>Verb</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>The GET method means retrieve whatever information (in the form of an entity) is identified by the Request-URI. If the Request-URI refers to a data-producing process, it is the produced data which shall be returned as the entity in the response and not the source text of the process, unless that text happens to be the output of the process.</td>
</tr>
<tr>
<td>POST</td>
<td>The POST method is used to request that the origin server accept the entity enclosed in the request as a new subordinate of the resource identified by the Request-URI in the Request-Line.</td>
</tr>
<tr>
<td>PUT</td>
<td>The PUT method requests that the enclosed entity be stored under the supplied Request-URI. If the Request-URI refers to an already existing resource, the enclosed entity SHOULD be considered as a modified version of the one residing on the origin server.</td>
</tr>
<tr>
<td>DELETE</td>
<td>The DELETE method requests that the origin server delete the resource identified by the Request-URI.</td>
</tr>
</tbody>
</table>

Table A.2 Project Relevant HTTP Requests Methods

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>OK</td>
<td>The request has succeeded.</td>
</tr>
<tr>
<td>201</td>
<td>Created</td>
<td>The request has been fulfilled and resulted in a new resource being created.</td>
</tr>
<tr>
<td>400</td>
<td>Bad Request</td>
<td>The request could not be understood by the server due to malformed syntax.</td>
</tr>
<tr>
<td>401</td>
<td>Unauthorized</td>
<td>The request requires user authentication</td>
</tr>
<tr>
<td>404</td>
<td>Not Found</td>
<td>The server has not found anything matching the Request-URI.</td>
</tr>
<tr>
<td>405</td>
<td>Method Not Allowed</td>
<td>The method specified in the Request-Line is not allowed for the resource identified by the Request-URI</td>
</tr>
<tr>
<td>500</td>
<td>Internal Server Error</td>
<td>The server encountered an unexpected condition which prevented it from fulfilling the request.</td>
</tr>
</tbody>
</table>

Table A.3 Project relevant HTTP Response Codes
getColor: ->
  # Builds a hex color string close to the blue
  # colour used in the project.
  red = Math.floor((Math.random()*30))
  green = Math.floor((Math.random()*20 + 159))
  blue = Math.floor((Math.random()*50 + 200))
  if red < 16
    '#0' + red.toString(16) + green.toString(16) + blue.toString(16)
  else
    '#1' + red.toString(16) + green.toString(16) + blue.toString(16)

Listing 15 Random Colour Generator