The Report Committee for Sashmit B. Bhaduri
certifies that this is the approved version of the following report:

sALERT: An Intelligent Information Alerting and Notification Web Service

APPROVED BY

SUPERVISING COMMITTEE:

sALERT: An Intelligent Information Alerting and Notification Web Service

by

Sashmit B. Bhaduri, B.S.C.S.

REPORT

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ENGINEERING

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2012

This report is dedicated to my parents for instilling within me a lifelong love for learning, as well as my sister for always supporting me in my educational endeavors.

Acknowledgments

I would like to thank my supervisor, Dr. Adnan Aziz, for his valuable feedback and insights for this project. I would also like to thank Dr. Daniel Miranker for graciously agreeing to act as my reader. In addition, I would like to thank my past professors for teaching me many wonderful things, and my classmates for imparting useful tidbits of knowledge, some of which have found their way into this report.

sALERT: An Intelligent Information Alerting and Notification Web Service

Sashmit B. Bhaduri, M.S.E.
The University of Texas at Austin, 2012

Supervisor: Adnan Aziz

Web services increasingly serve as large repositories and conduits of information. However, they do not always allow for the efficient dissemination of this information, particularly in a reactive way. In this report, I describe sALERT, a web-based application that allows for targeted information from various web services to be combined and cross-referenced in order to produce a system that is more convenient and more efficient in reactively disseminating information. This dissemination is performed using mobile notification mechanisms such as text messages, and information targeting is performed using data from social networks and geolocation sources. I present the design, implementation, and plans for future improvement for this service within this report.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Vision

It is a common need, in both the physical and virtual worlds, to be able to share information, content, and events with other people. A business, for example, may want to share information about special sales to customers, or an individual may want to share links quickly with friends. Similarly, it is common for people to seek out and search for such pieces of information from friends, businesses, or other entities that would publish them. It is natural to view this process as that of information creation and information discovery.

At the same time, there is often a need to either broadcast or receive information in a realtime or semi-realtime basis. People want to use their mobile devices increasingly often to receive this information. The mechanisms for performing these tasks online come in many forms, such as websites transmitting announcements over email subscription lists, members of social networks writing status messages to their followers and friends, and websites syndicating information over various feed formats. Some advanced websites may even be able to send out alerts upon key events being triggered, such as airline websites sending out flight status texts and emails to passengers upon exceptional circumstances. Other online presences do not actively try to broadcast information to their userbases, but are producers of information and events nonetheless.

I propose sALERT – a system that provides a means of unifying and augmenting these various information alerting mechanisms. By being able to intelligently link together characteristics of both consumers and information produced by publishers through a social network-data based recommendation system, the system allows for potentially more effective dissemination and discovery of information and information sources. By allowing mobile-friendly notification methods, services that had no way of sending out mobile alerts can suddenly do so. sALERT is extensible enough to progressively add more content sources, and as this occurs, users gain a centralized location to manage alerts and subscriptions to sources of alerts.

Social networks have become ever pervasive instruments for people to connect with others. For third parties that operate or interact with these platforms, they serve as rich sources of data and a valuable source of users. By interfacing with social network systems as user-information sources, sALERT allows information publishers to more easily find those who may be interested in the information they produce. I chose Facebook, currently the most popular social network, as the primary social network that the system interacts with. This network has a large ecosystem of third party developers producing services and applications oriented for the platform including everything from games to music players that have been made more social due to Facebook's platform and availability of users.

To summarize, my main goals in this project were to create a system in which users can be notified of information and events from various sources that correspond to their interests, to allow for mobile-friendly notifications for receiving these alerts, to allow for information publishers to send out information items and target them to specific users in this system, and lastly, to use social network-derived data to match these publishers and consumers so they can more easily discover each other.

The rest of this report is organized as follows: Chapter 2 focuses on planning the functionality of the proposed system by the exploration of user stories and interface mockups. Chapter 3 focuses on the design and implementation of the proposed system. Chapter 4 presents various results based on this implementation. Chapter 5 reaches conclusions about the proposed system and defines scoping for future work. Prior work is reviewed in Chapter 6.

Chapter 2

Specification

2.1 User Stories

In order to plan the functionality of the system, I created a series of user stories to qualify how users would use the system within the context of the broad level objectives stated in the vision. The development of user stories at the beginning of an overall software design process has become an important part of Agile-oriented software development methodologies such as Extreme Programming [34]. The stories presented in this section are in the form recommended by Mike Cohn [5], with a general template of "As a *role*, I want *goal*."

The stories themselves are divided into three types of user roles: *a*) information consumers, *b*) information publishers, and *c*) administrators. Each of these roles are described below, however, it is worth noting that a single person or entity may potentially function within the context of two or three roles simultaneously. In addition to a division by role, the user stories are further subdivided into general and specific sets of stories, corresponding to generalized and more concrete scenarios respectively. These user stories were an important part of gaining an intuition of the scope and functionality needed in the system.

2.1.1 Information Consumer

Table 2.1 delineates user stories that I created for an information consumer user role. For the purposes of these user stories, I conceptualized consumers as end users who use the system to consume information from alerts.

I recognized several different important concepts in these consumer-oriented user stories. These include the concepts of channels, or a conduit of information, and subscriptions, a stateful means of linking a consumer and publisher that is initiated by the consumer and then persisted by the system. When information would be pushed over the channel, it would delivered as a notification by the system to subscriber. As described by these stories, these channels may be derived from some remote Internet-based site, or may be local in nature. In addition, some of the stories also implied the need of some sort of recommendation system to help discover channels that much their interests. Finally, I extracted the need for multiple methods of delivering alerts to the end user from these stories.

2.1.2 Information Publisher

Table 2.2 summarizes a set of user stories that I created for the information publisher user role. A publisher could be an individual or organization who uses the system to send information to information consumers.

In general, these stories helped me define several new pieces of func-

Type	**Description**
Specific	As an *user*, I want to subscribe to a weather alert channel and receive a text when it is going to rain tomorrow.
Specific	As an *user*, I want the system to send me a text and email the 3rd of every month to remind me to pay my rent.
Specific	As an *user*, I want to subscribe to my local coffee shops channel and get alerts when they have flash sales.
Specific	As an *user*, I want to get alerts when businesses in my town that I havent heard of or used offer deals that I might be interested in.
Specific	As an *user*, I want to subscribe to a SlickDeals channel and receive alerts when there are cheap thumb drives on sale on my area.
Specific	As an *user*, I want to subscribe to a Craigslist channel and receive alerts when someone is selling SXSW tickets.
General	As an *user*, I want to be able to subscribe to various channels of my choosing.
General	As an *user*, I want to be able to be alerted to new items and offers from my subscribed channels.
General	As an *user*, I want to get alerts for new items from channels I am not subscribed to, but may be of interest to me.
General	As an *user*, I want to browse any special offers in my area regardless of whether I am subscribed to those channels.
General	As an *user*, I want to, on demand, get suggestions about what other channels I could subscribe to based on my interests.
General	As an *user*, I want to customize how I receive alerts on a per-channel basis.
General	As an *user*, I want to be able to setup filters on my subscribed channels.

Table 2.1: Consumer Oriented User Stories

tionality that are useful from the point of view of an information publisher, including the need to be able to manually create alerts to deliver as notification alerts to users, being able to find new users interested in the publisher's content, and being able to have some sort of tracking of published items for analytical purposes.

2.1.3 Admin

Table 2.3 gives examples of user stories that I created for the administrator user role. These users consist of privileged users who need to administrate the system. The primary new pieces of perceived functionality in these stories are additional analytics needing to be tracked by the system and administrator-level control of publisher created channels.

2.2 UI Mockups

In order to explore the user interface of a proposed system in relation to the user stories mentioned in Section 2.1, I prototyped the user interface by creating a series of mockups using a rapid prototyping tool, Balsamiq [31]. I did this before starting work on the actual implementation of the system. These mockups were useful for discerning what the different components of the end user interface needed to be, as well as finding the relationships and user-oriented action flow between them.

At the same time, mockups allowed me to quickly change and evolve concepts over multiple brainstorming sessions. Moreover, they were invaluable

Type	**Description**
Specific	As a *restaurant owner*, I want to send deals to my usual patrons on a regular basis.
Specific	As a *restaurant owner*, I want my employees to use the system to send out flash deals when they see the restaurant empty.
Specific	As an *electronics shop owner*, I want to reach new customers by sending out alerts when I get excess inventory of some product.
General	As a *publisher*, I want to find new customers by sending them offers they might be interested in.
General	As a *publisher*, I want to alert existing subscribed customers of new offers.
General	As a *publisher*, I want to have all pending offers be browsable and searchable by any potential consumers.
General	As a *publisher*, I want to have an easy to use interface for setting up offers.
General	As a *publisher*, I want to be able to customize the description of offers.
General	As a *publisher*, I want to be able to characterize offers such that they are matchable to users interests.
General	As a *publisher*, I want to set start and expiration dates for offers.
General	As a *publisher*, I want to track at a glance which of my offers are being followed up on through the service.

Table 2.2: Publisher Oriented User Stories

Description
As an *admin*, I want to track how many users are subscribed to what channels.
As an *admin*, I want to track how often users click on individual deals based on alerts.
As an *admin*, I want to enable or disable channels.
As an *admin*, I want to configure categories of channels.

Table 2.3: Administrator Oriented User Stories

for developing a conceptual sense of the data that needed to be stored for specific types of objects. The mockups are intentionally expressed in a low fidelity form that is reminiscent of pencil and paper, a characteristic that has been proposed as advantageous and more effective than higher level designs [28].

In Figure 2.1 and Figure 2.2, a mockup of a new subscription page is presented, where each figure represents multiple tabs within the same screen. I viewed this screen as an interface where users could set up subscriptions to individual channels and define an output mechanism such as email or text messages for receiving alerts for this subscription. I envisioned three categories of channels within this page:

1. System Channels – Utility channels in which the system could generate alerts based on some trigger, such as sending an user an alert at a specific time every day.

2. Internet Channels – Channels for information sources that exist remotely on the Internet, such as information being pushed out from a website

using a mechanism such as a Really Simple Syndication (RSS) feed or a HTTP API.

3. Locals Channels – Channels for publishers associated with a geotagged address.

In Figure 2.3, a mockup of an information consumer-oriented screen is shown. I planned this interface to be where users can discover channels near the user, as well as browse the current items (called *offers* within the mockups) published by those channels. Moreover, I envisioned the map in the mockup to allow the user to see at a glance the geographical locations of each of the items. Ancillary pieces of information about each item were also put into the mockup as a way providing to the user contextual information, such as the distance between the publisher of the item and the user, as well as a rating of each item as voted by users. To illustrate real world usage of the system, the end user in the mockup is searching for Vietnamese Food within some distance of a zip code.

In Figure 2.4, a mockup of an information publisher-oriented screen is presented. This screen allows publishers to push new pieces of information into a channel that the user owns. The mockup is from the perspective of a Vietnamese-restaurant owner who wishes to notify subscribed users about a new offer from the restaurant. In addition, the screen allows the publisher to target users who are not subscribed to the channel by tagging a maximum distance and a set of terms associated with the item for recommendation purposes.

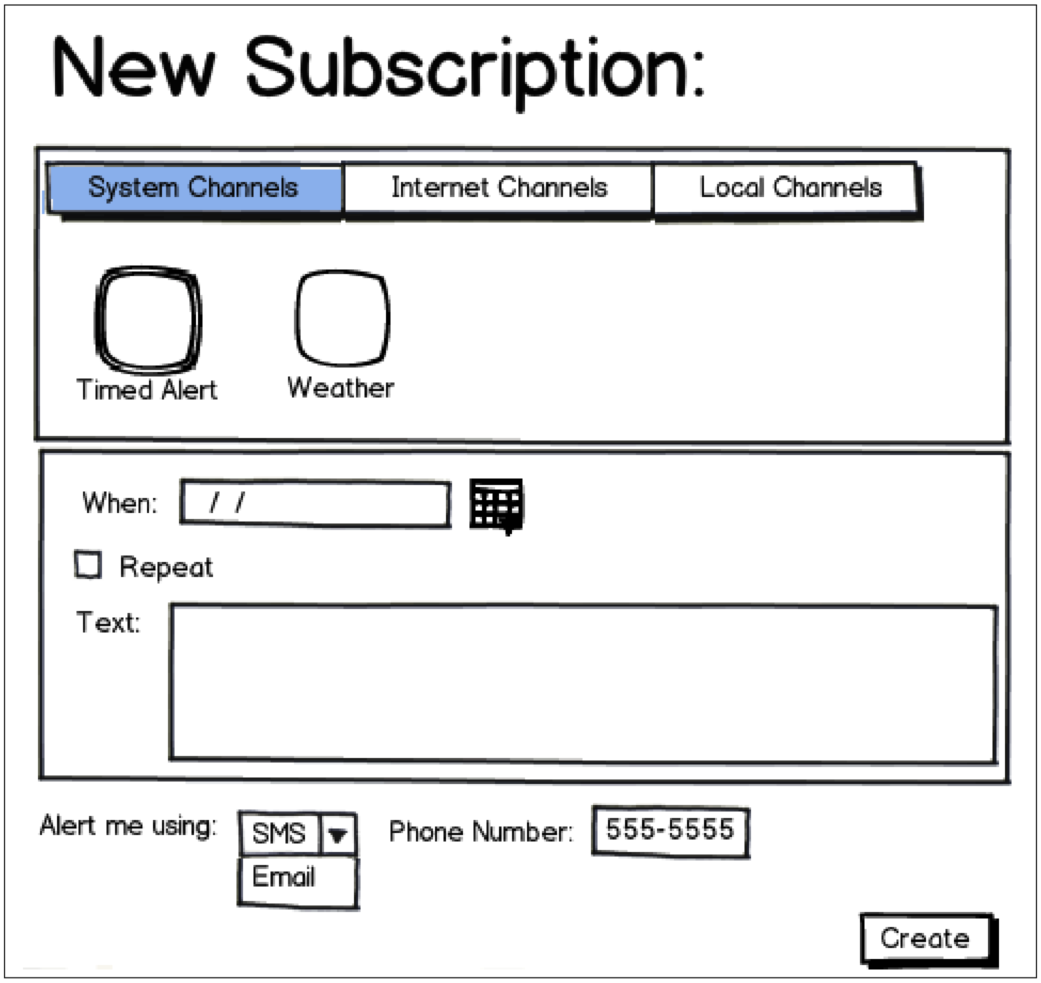


Figure 2.1: New Subscription Mockup

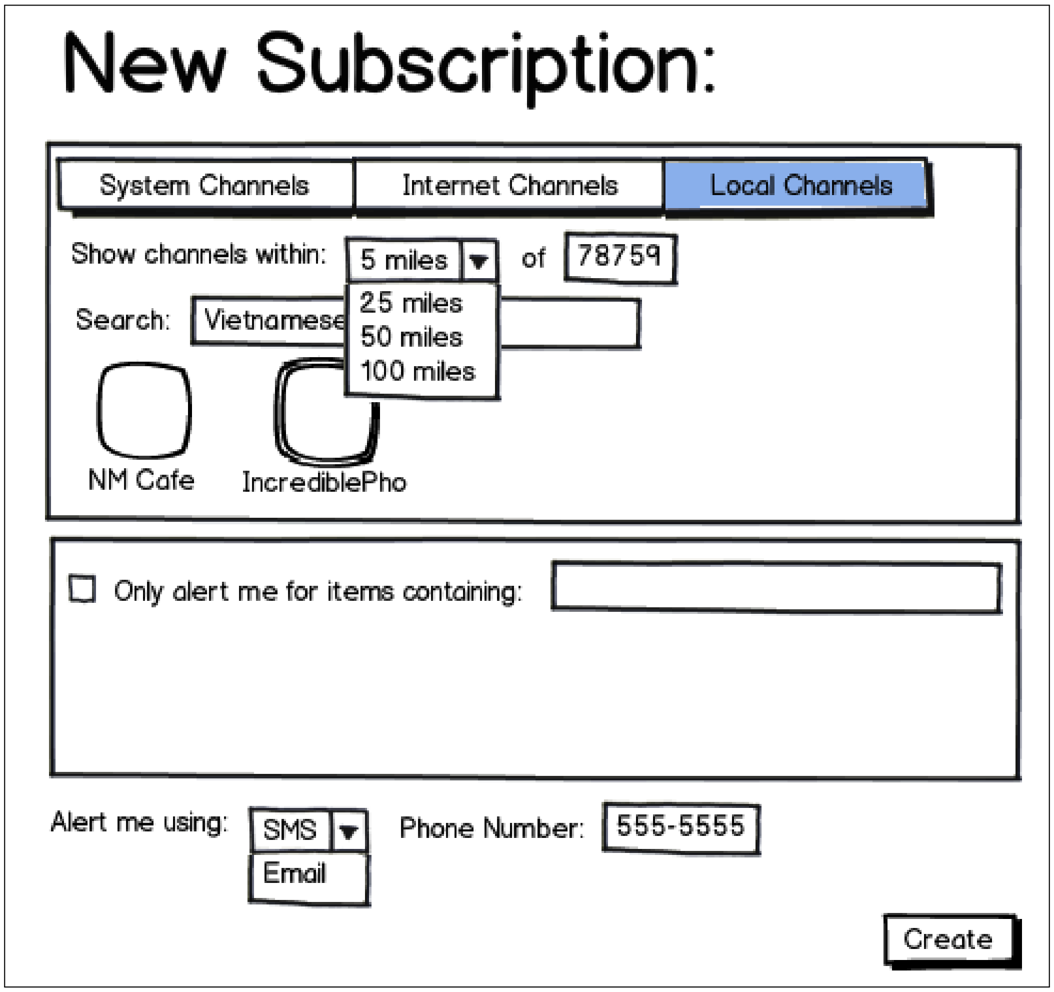
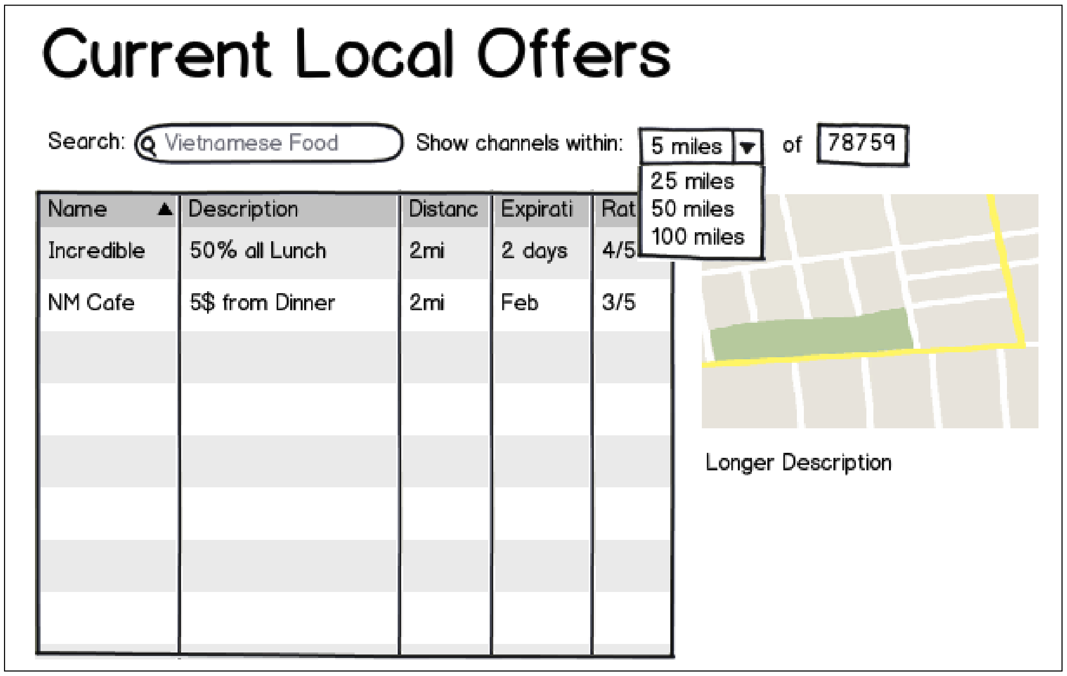


Figure 2.2: New Local Subscription Mockup

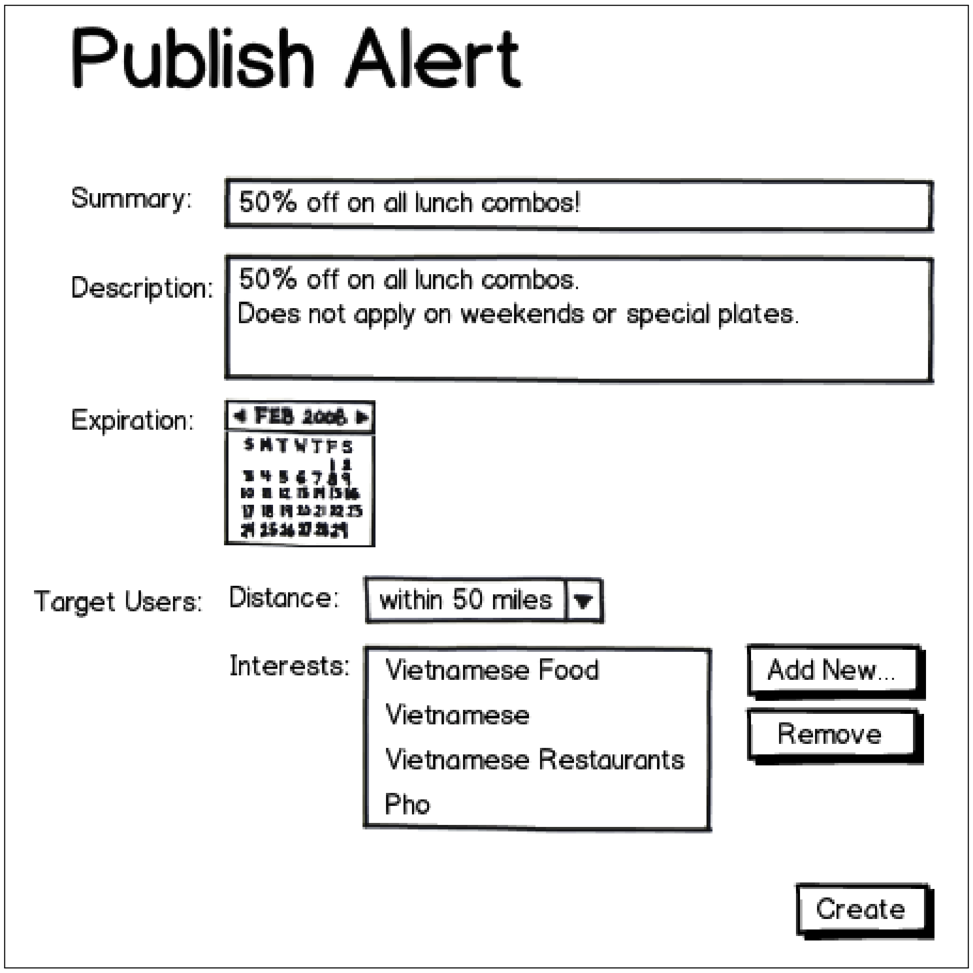


Current Local Offers

Search: Vietnamese Food   Show channels within: 5 miles ▼ of 78759

Name ▲	Description	Distanc	Expirati	Rat
Incredible	50% all Lunch	2mi	2 days	4/5
NM Cafe	5$ from Dinner	2mi	Feb	3/5

Figure 2.3: Current Local Offers Mockup



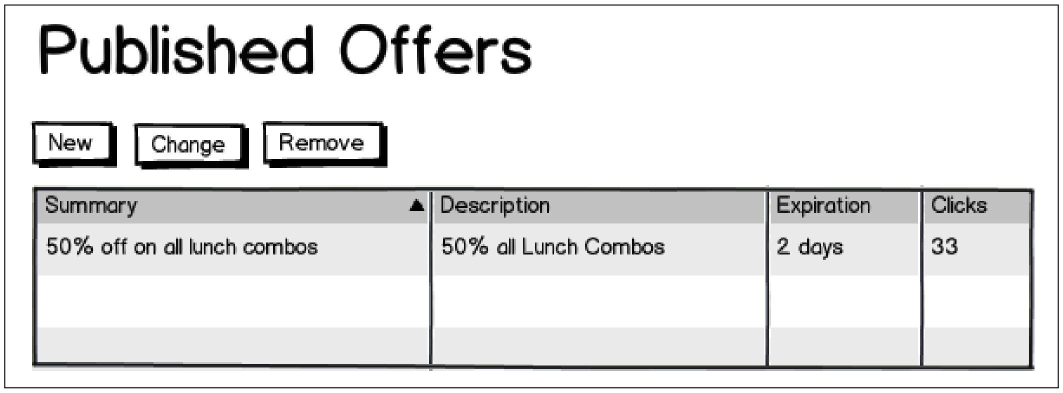
Summary: 50% off on all lunch combos!

Description: 50% off on all lunch combos.
Does not apply on weekends or special plates.

Distance: within 50 miles

Interests: Vietnamese Food, Vietnamese, Vietnamese Restaurants, Pho

Figure 2.4: Publish Alert Mockup

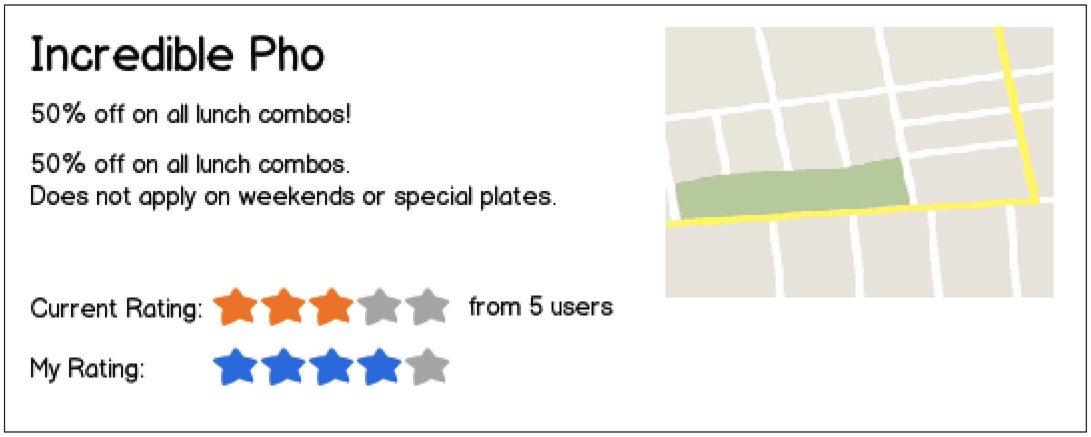


Published Offers

New | Change | Remove

Summary	Description	Expiration	Clicks
50% off on all lunch combos	50% all Lunch Combos	2 days	33

Figure 2.5: Published Offers Mockup



Incredible Pho

50% off on all lunch combos!

50% off on all lunch combos.
Does not apply on weekends or special plates.

Current Rating: from 5 users

My Rating:

Figure 2.6: View Alert Mockup

In Figure 2.5, a mockup of a screen is presented whose aim is to allow the information publisher to display, at a glance, all of the items that they publish from their owned channels. The number of clicks is displayed for each item to allow the publisher to assess popularity of each item.

In Figure 2.6, a mockup of a consumer-centered screen for viewing information about a specific item is shown. I visualized this screen to be what would be linked to by the system by any notification mechanisms ultimately supported, such as emails and text messages. I envisioned that the screen allow the user to see the location of an item (if applicable) on a map, as well to both view the current rating of an item and rate it themselves.

Chapter 3

Implementation

3.1 Introduction

In this chapter, I describe the implementation of a prototype system centered on the user stories and mockups presented in Chapter 2. I focused the implementation towards the creation of a minimally viable system that implemented most of the functionality described in the previous chapter while serving as the basis for future work, described in Section 5.3. Both deviations and extensions to the system proposed in Chapter 2 will be described as appropriate.

Figure 3.1 shows a high level overview of the logical components of the implementation, as well as the relationships and pieces of data shared between them. Each of these components will be described in depth within this chapter.

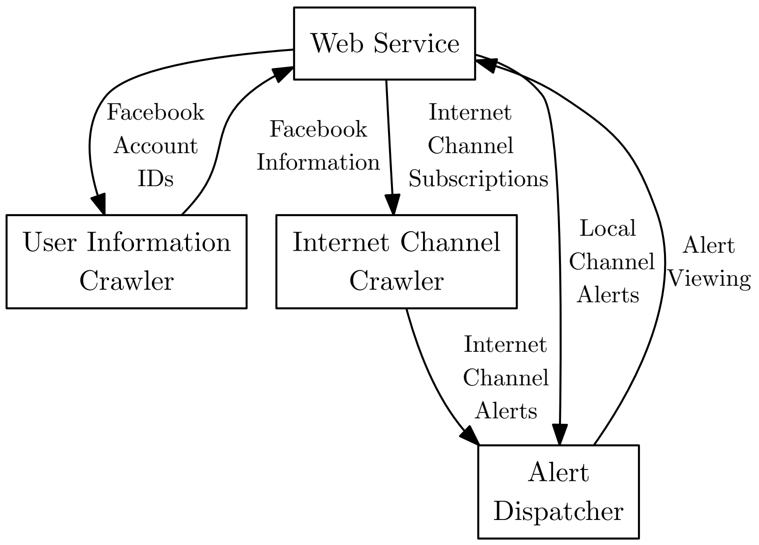


Figure 3.1: Logical Components

3.2 Technologies

The technologies used within the implementation are summarized in Table 3.1. Generally speaking, those technologies used by a specific logical component are described in their respective sections. Technologies and idioms that are used more pervasively will be described in this section.

3.2.1 Cloud Services

I used Amazon Elastic Compute Cloud (EC2) to host the various components of the sALERT system and serve as a cloud infrastructure provider. EC2 is an example of a cloud-based Infastructure as a Service (IaaS), or delivery of computer infrastructure as a service based on some usage-based payment scheme [36], usually in a scalable way.

Type	**Technology**	**Logical Components Used In**
Cloud Infrastructure	Amazon EC2	All
Web Application Framework	Django	All
Web Server	Apache	Web Service
Asynchronous Message Passing	RabbitMQ	Web Service, Alert Dispatcher
Task Queueing	Celery	Web Service, Alert Dispatcher
Database	MySQL	All (if configured)
Database	SQLite	All (if configured)
Database	MongoDB	Internet Channel Crawler
Web Service API	Twitter API	Internet Channel Crawler
Web Service API	Facebook API	Web Service, User Information Crawler
Web Information Feed	Craigslist	Internet Channel Crawler
Misc Library	Feedparser	Internet Channel Crawler
Clientside JavaScript Library	jQuery	Web Service

Table 3.1: Summary of Technologies Used

I used EC2 in a fairly naive way in the current implementation, using it mostly as a Linux-based hosting solution. I deployed an Amazon Machine Image (AMI) with a stock version of Canonical Ltd's Ubuntu Linux Cloud Edition on a single micro EC2 instance, and I installed and configured any supporting software, including much of the software components listed in Table 3.1 by hand. This approach serves well for the construction of a system implementing initial functionality; it is not particularly optimized for scalability. Nonetheless, EC2 is part of a large umbrella of cloud-related services offered by Amazon, collectively referred to as Amazon Web Services (AWS). EC2, in addition to the technologies in AWS, allow for the rapid scaling up of server needs as needed. Some ideas for using more of EC2 will be described in Section 5.3.4.

3.2.2 Web Application Frameworks

Web frameworks are libraries that assist the development of web-based services and applications. Web frameworks have become extremely popular in recent years amongst Web application programmers, especially in the form a newer generation of frameworks based on a Model View Controller (MVC) architecture, long used in traditional software development, but shunned until relatively recently for Web development [25]. These frameworks have helped increase developer productivity when creating web-centered applications by offering a large set of functionality while promoting modularity. Traditionally, dynamic content on the web entailed Common Gateway Interface (CGI) or servlet style programming, which typically exposed a limited set of functions to the application writer, such as print statements [11]. In these older models, the web server would pass requests to another functional block to actually handle and serve the request. The requests could either be passed to an external program or, for efficiency, run within the same process space as the web server using server-specific plugins and extensions. These CGI scripts and servlet programs would then access external systems such as databases as needed. Web frameworks subsume this model, and even use CGI and servlet technologies for communication with a web server, but provide a richer set of programming interfaces for the developer. Thus, web frameworks can be thought of as an evolution of older CGI and pure servlet style web programming.

In this project, I used the Django web framework. Django is an MVC architecture that is composed of a multitude of Python-based application support modules. Django heavily espouses the concept of Don't Repeat Yourself (DRY) [12] and has a large ecosystem of developers and third-party extensions. Similar to other modern MVC-based web frameworks such as Ruby-on-Rails [25], Django includes the ability to provide an abstraction of data in a higher level language as opposed to database-specific languages, commonly referred to as an Object Relational Model (ORM), and includes URL dispatching services as well as rich HTML templating features. Additionally, modern web frameworks also tend to have features to assist with techniques that have become popular recently, including the transfer of data in JavaScript Object Notation (JSON), handling of Asynchronous Javascript and XML (AJAX) calls, etc. I extensively used many of these facilities within the implementation, not only for the web-based frontend of the system, but the backend components as well.

A more complete introduction to the core concepts behind Django with a particular focus on its MVC architecture is presented in Appendix A.

3.3 Data Models

In Figure 3.2, a simplified graph of the Django data models used by multiple components of the implementation is illustrated. Models provided by Django itself, and Django-oriented libraries that I used such as Celery and `django_facebook` are not listed, with the exception of the “User” model from Django’s authentication framework, which is key to representing several types of relationships between different models. The operation of key fields and relationships will be elaborated on in subsequent sections.

3.4 Web Service

The web service component of sALERT comprises the user-facing frontend of the system. In addition to a few general concepts used by the web service, each of the screens of the web-based interface are examined in this section.

3.4.1 Authentication

The web service uses Facebook as a primary user authentication source, and the landing page of the web interface results in little else other than a login link. Facebook allows authentication with external third party web services

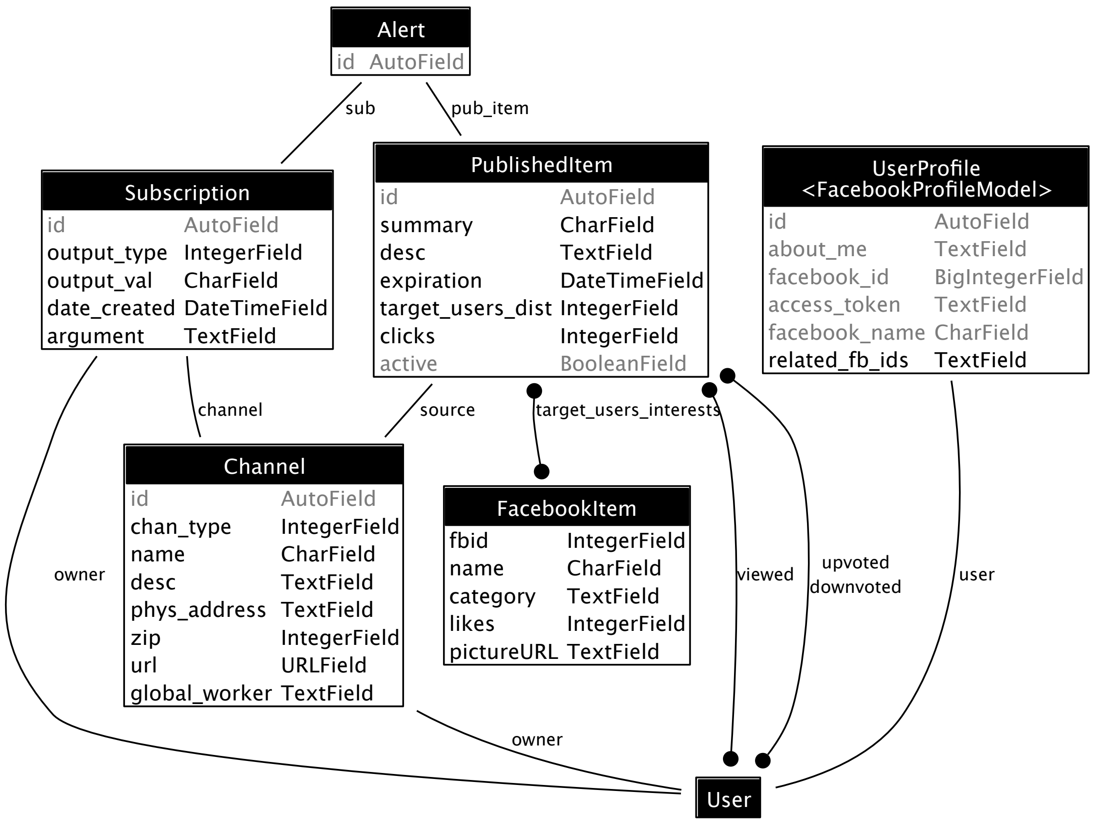


Figure 3.2: Models

exclusively using an Open Authorization (OAuth2) based protocol. OAuth2 is currently a draft specification [10], however, it has already been widely adopted to handle authentication and third-party application authorization for a variety of web services.

To perform OAuth2-based authentication when the user clicks the "Login with Facebook" link on the landing page, the following steps occur:

1. The Facebook JavaScript API's OAuth2-based login/authorization popup is invoked.
    (a) If the user is not currently logged into Facebook, a username and password is asked for by the popup.
    (b) The user is asked, via the popup, if they want to authorize the sALERT application to access a certain set of data, called a *scope*.
2. Assuming the popup successfully closes, an *authorization code* is received and sent to a Django view.
3. This view retrieves an *access token* by using a client ID and client secret special to sALERT that have been previously generated when registering the application with Facebook.
4. A new Django User model object is created with the email address of the Facebook account just authorized, if one does not exist already.
5. The User model object's login function is called, which makes it available to all Django views, including views implemented for the system.

6. The access token is stored for later usage by the User Information Crawler component.

Much of this is abstracted out by the usage of a combination of Django's built in user authentication system which offers an extensible authentication backend system, and a modified version of `django_facebook`, which handles many of the OAuth2-specific handling logic. OAuth permissions that are asked for include a user's likes, interests, groups, events, checkins, activities, email, birthday, and other profile information, however not all of these are used at the moment. I modified the latter to account for ongoing changes in the Facebook OAuth2 API.

3.4.2 Geolocation

Several parts of the sALERT web service either require the user's current location or need to calculate distances between two geographical points or zip codes. The methodologies and algorithms that I used will be described in this subsection.

The system uses zip codes coupled with street addresses natively as a means of storing object locations (e.g, for local channels). Additionally, it exposes zip codes to the user when the user is filtering in various user interfaces within the web interface (e.g, finding objects that are within n miles from around a zip code). Addressing locations in this coordinate system is both intuitive to the end user and supported natively by the Google Maps API used in the web interface.

However, in order to compare the distance between two locations, it is more convenient mathematically to convert locations into latitude/longitude space. In the current implementation, only the zip code is transformed into a latitude/longitude for distance matching purposes, and not the street address itself. I used information derived from the US Census Bureau [1], from where the geographical center of each zip code in the United States is retrieved. Since this only works in the United States, and there is some inherent error in using the geographical center of a zip code rather than a true latitude/longitude of a street address, some better strategies are proposed for future work in Section 5.3.1. In addition to a zipcode $\rightarrow$ latitude/longitude transformation, I also implemented a reverse transform by finding the zip code with the minimum distance between its geographical center and a given latitude/longitude. This is also an approximation, but suffices for the cases it was used for.

To find distances between two coordinates in latitude/longitude space, I used the Haversine formula, listed in Equation 3.1. This method computes the great circle distances between two latitude/longitude pairs (ϕ_1, ψ_1 and ϕ_2, ψ_2) using a constant radius r that is a spherical approximation for the Earth's true geometry [39].

$$d = r\,\mathrm{hav}^{-1}\left(\sqrt{\mathrm{hav}(\phi_2 - \phi_1) + \cos(\phi_1)\cos(\phi_2)\,\mathrm{hav}(\psi_2 - \psi_1)}\right) \tag{3.1}$$

which may be rewritten, using more common trigonometric functions as opposed to *versed* trigonometric functions that are sometimes used for navigational calculations:

$$d = 2r \arcsin\left(\sqrt{\sin^2\left(\frac{\phi_2 - \phi_1}{2}\right) + \cos(\phi_1)\cos(\phi_2)\sin^2\left(\frac{\psi_2 - \psi_1}{2}\right)}\right) \quad (3.2)$$

In addition to implementing coordinate space transforms, the web service must also attempt to find the user's current location. For this, I implemented support for the HTML5 Geolocation API, currently a World Wide Web Consortium (W3C) candidate specification [35]. This API provides a simple interface to allow websites to retrieve a user's current geographical location in latitude/longitude space from location-aware browsers, typically after a user authorizes this action on a per-website basis. The actual source of information for this geolocation data is dependent on browser and device. It may be based on a GPS or cell tower information if available, for example, on a phone, or it may be based on triangulation based on Wi-Fi networks or inferred from a IP address [7]. Because of this, the accuracy of the data can vary wildly based on circumstance and what data sources are available. For example, the best IP-based methods, may achieve median errors of 690-700 meters for certain types of US-based IPs (assuming no proxy or VPN is used), while 200-1000 meters accuracy has been achieved using cell tower information and 10-20 meters can be had using Wi-Fi databases, with GPS being regarded as the most accurate data source [41].

3.4.3 User Recommendations

In addition to users subscribing to channels to receive alerts, the system also shows recommended items to the user. This is ultimately performed and presented within the web service. Users are matched up with recommendations using two methods: geographical proximity and interest-based matching. Currently, user recommendations only occur for items that are tied to local channels, and not Internet-based sources, although more advanced recommendation techniques may address this in the future. The current methodologies serve to provide a basis for future improvements, mentioned in Section 5.3.1.

In the first case, users are shown recommended alerts based on geographical proximity. When a publisher creates an item, they have the opportunity to provide a targeted maximum user distance. The idea in this case is that different types of items and channels may have different domain-specific distances that potentially interested users may be willing to travel within. For example, a professional sports team may want to send alerts to a very wide radius across a metropolitan area, but a local neighborhood bar may only want to target users within the confines of a few miles.

The system, in several location aware user interface pages, attempts to retrieve a user's current latitude/longitude and then maps it to a zip code using the methodology in Section 3.4.2. This is then compared with every item to see if it meets the item's minimum distance matching criteria, and if so, a recommendation is generated. Additionally, the end user has an opportunity to change the location the system believes they are at by changing the zip code in the location area pages. These are persisted across multiple interface screens by storing the current tracked location in a HTTP session cookie.

The second method of recommended item matching is performed by comparing a user's Facebook graph information in the form of Facebook Graph Object identifiers (see Section 3.5) with a set of targeted interests as defined by an information publisher when an item is created. For a given user, this Facebook graph information may include their likes, interests, activities, and a limited set of demographic information such as the languages they speak, their location, hometown, etc. For a given user U and an item I, recommendations are then generated on the basis of:

$$\mathsf{Ids}_{\mathsf{C}} = \mathsf{Ids}_{\mathsf{U}} \cap \mathsf{Ids}_{\mathsf{I}} \tag{3.3}$$

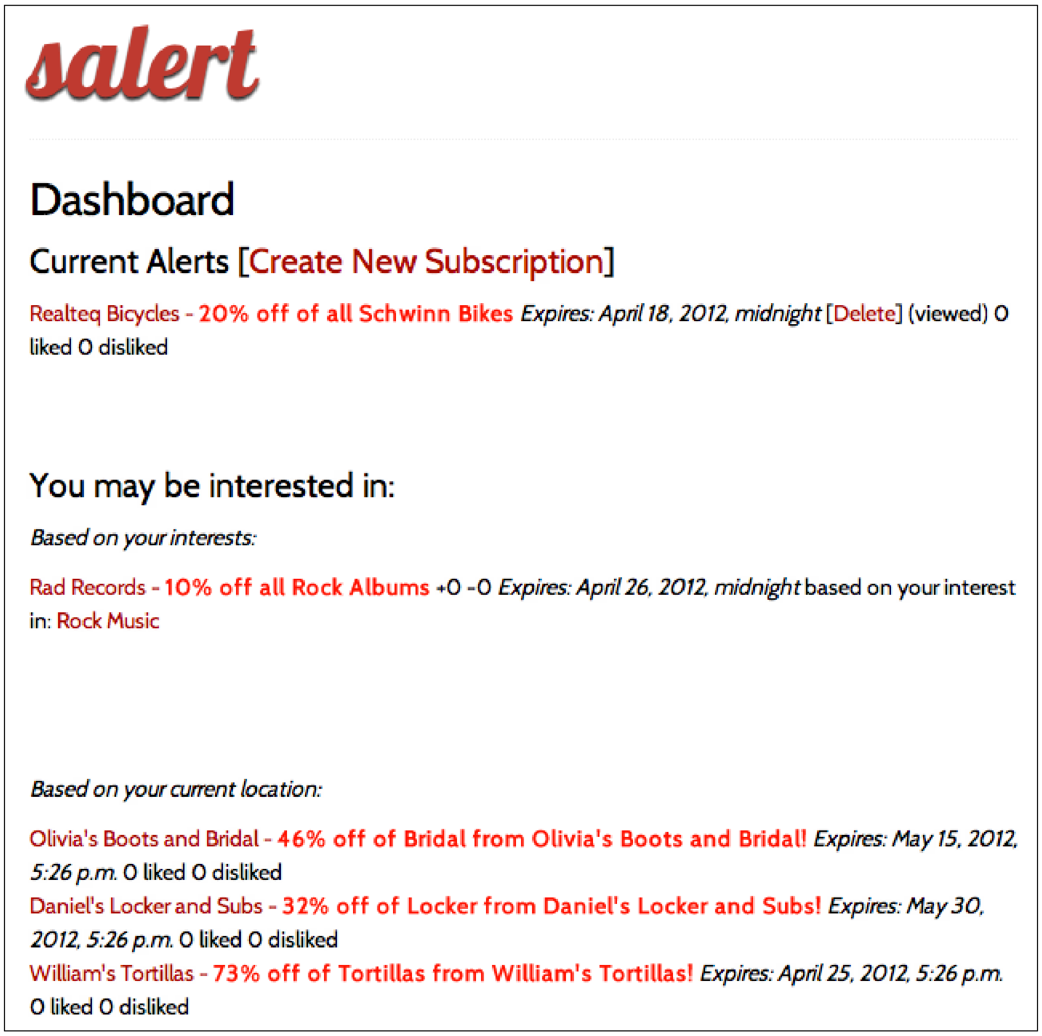
Where Ids correspond to Facebook Graph Object identifiers. If $\mathsf{Ids}_{\mathsf{C}} \neq \emptyset$, a recommendation is generated for item I, and is annotated with $\mathsf{Ids}_{\mathsf{C}}$ to allow the user to know why the recommendation was made. This style of filtering data for purposes of generating recommendations using a user's older preferences has been referred to as *keyword-based filtering* [38]. This method allows information publishers to specifically target users who have specific interests or demographic information. For example, someone who is using the system to notify people about a book club meeting may want to create an item whose targeted interest includes the Facebook Graph Objects for that book, the author, and the book's genre. This recommendation model is limited due to the enormity of objects represented by the Facebook Graph as users may have interests that are similar but not identical to a publisher's targeted graph objects. This implementation serves as a first order implementation that may be improved in various ways. Chapter 6 discusses some related work on recommendation engines in general.

3.4.4 Web User Interface Pages

In this section, each of the major pages within the web service are described and compared with the pre-implementation mockups.

In Figure 3.3 is a screenshot of the *dashboard* page of the web interface. This page serves to both show current alerts for the channels the user has subscribed to, as well as recommendations of items based on interests (along with which of the user's interests triggered the recommendation) and based on location. I did not envision this page during the design process, and thus I did not mock it up, but it was clear to me that a central interface for displaying these pieces of information was useful. The dashboard also serves the additional useful purpose of being the screen that is shown after login occurs, and therefore may be thought of the main page for users.

Figure 3.4 presents a screenshot of the *New Subscription* page of the web interface and corresponds to the mockups represented in Figure 2.1 and Figure 2.2. In contrast to the mockups, I did not implement System Level-type channels in the final implementation. This is proposed as future work in Section 5.3.2. In addition, I did not implement filtering based on a textual string, which may be helpful with a larger number of channels. To simplify the interface, I scrapped multiple tabs as shown in the mockup in favor of showing and hiding form elements using dynamic JavaScript techniques depending on what channel was selected. In particular, Internet-based channels accept an extra argument field.



salert

Dashboard

Current Alerts [Create New Subscription]

Realteq Bicycles - **20% off of all Schwinn Bikes** *Expires: April 18, 2012, midnight* [Delete] (viewed) 0 liked 0 disliked

You may be interested in:

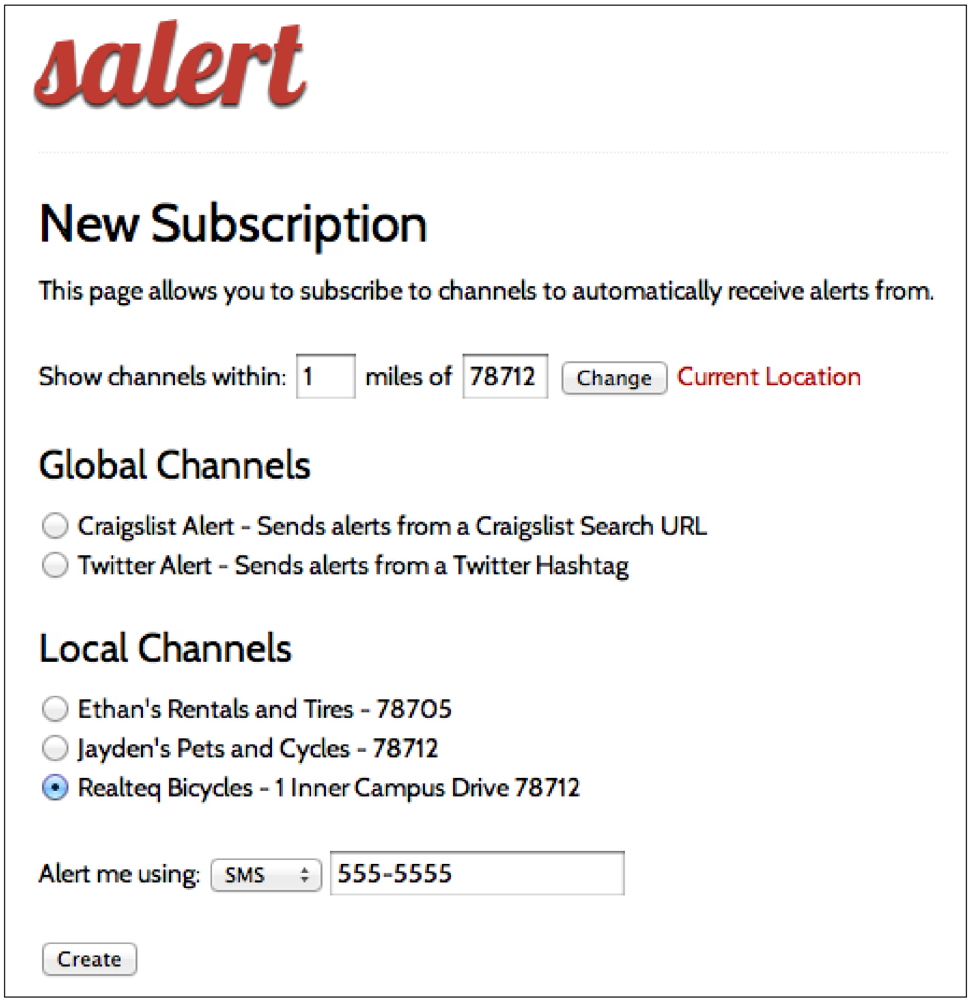
Based on your interests:

Rad Records - **10% off all Rock Albums** +0 -0 *Expires: April 26, 2012, midnight* based on your interest in: Rock Music

Based on your current location:

Olivia's Boots and Bridal - **46% off of Bridal from Olivia's Boots and Bridal!** *Expires: May 15, 2012, 5:26 p.m.* 0 liked 0 disliked
Daniel's Locker and Subs - **32% off of Locker from Daniel's Locker and Subs!** *Expires: May 30, 2012, 5:26 p.m.* 0 liked 0 disliked
William's Tortillas - **73% off of Tortillas from William's Tortillas!** *Expires: April 25, 2012, 5:26 p.m.* 0 liked 0 disliked

Figure 3.3: Dashboard Page


salert

New Subscription

This page allows you to subscribe to channels to automatically receive alerts from.

Show channels within: 1 miles of 78712 Change Current Location

Global Channels

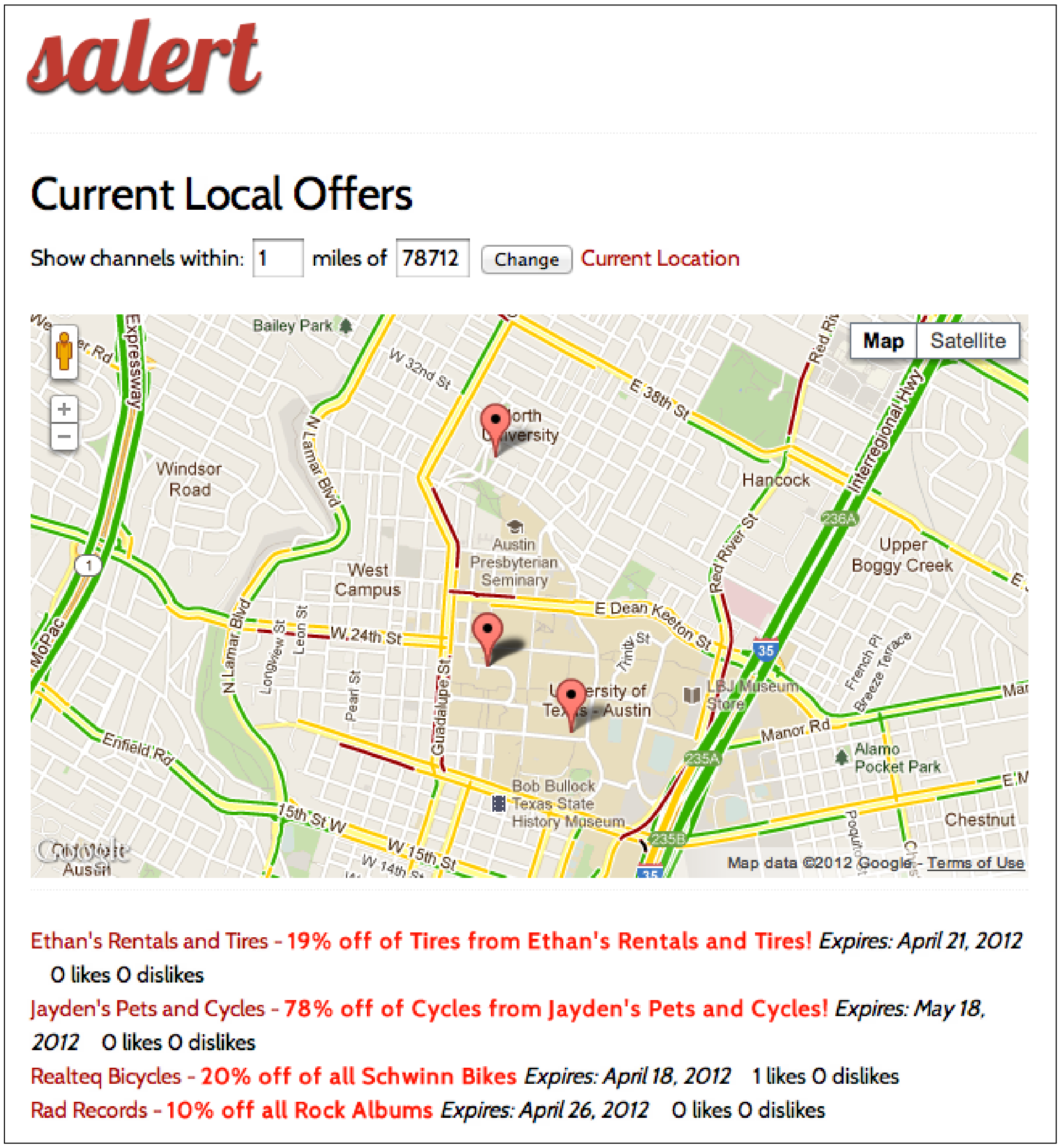
- ○ Craigslist Alert - Sends alerts from a Craigslist Search URL
- ○ Twitter Alert - Sends alerts from a Twitter Hashtag

Local Channels

- ○ Ethan's Rentals and Tires - 78705
- ○ Jayden's Pets and Cycles - 78712
- [✓ Realteq Bicycles - 1 Inner Campus Drive 78712]

Alert me using: SMS 555-5555

Figure 3.4: New Subscription



salert

Current Local Offers

Show channels within: 1 miles of 78712 Change Current Location

Ethan's Rentals and Tires - **19% off of Tires from Ethan's Rentals and Tires!** *Expires: April 21, 2012* 0 likes 0 dislikes

Jayden's Pets and Cycles - **78% off of Cycles from Jayden's Pets and Cycles!** *Expires: May 18, 2012* 0 likes 0 dislikes

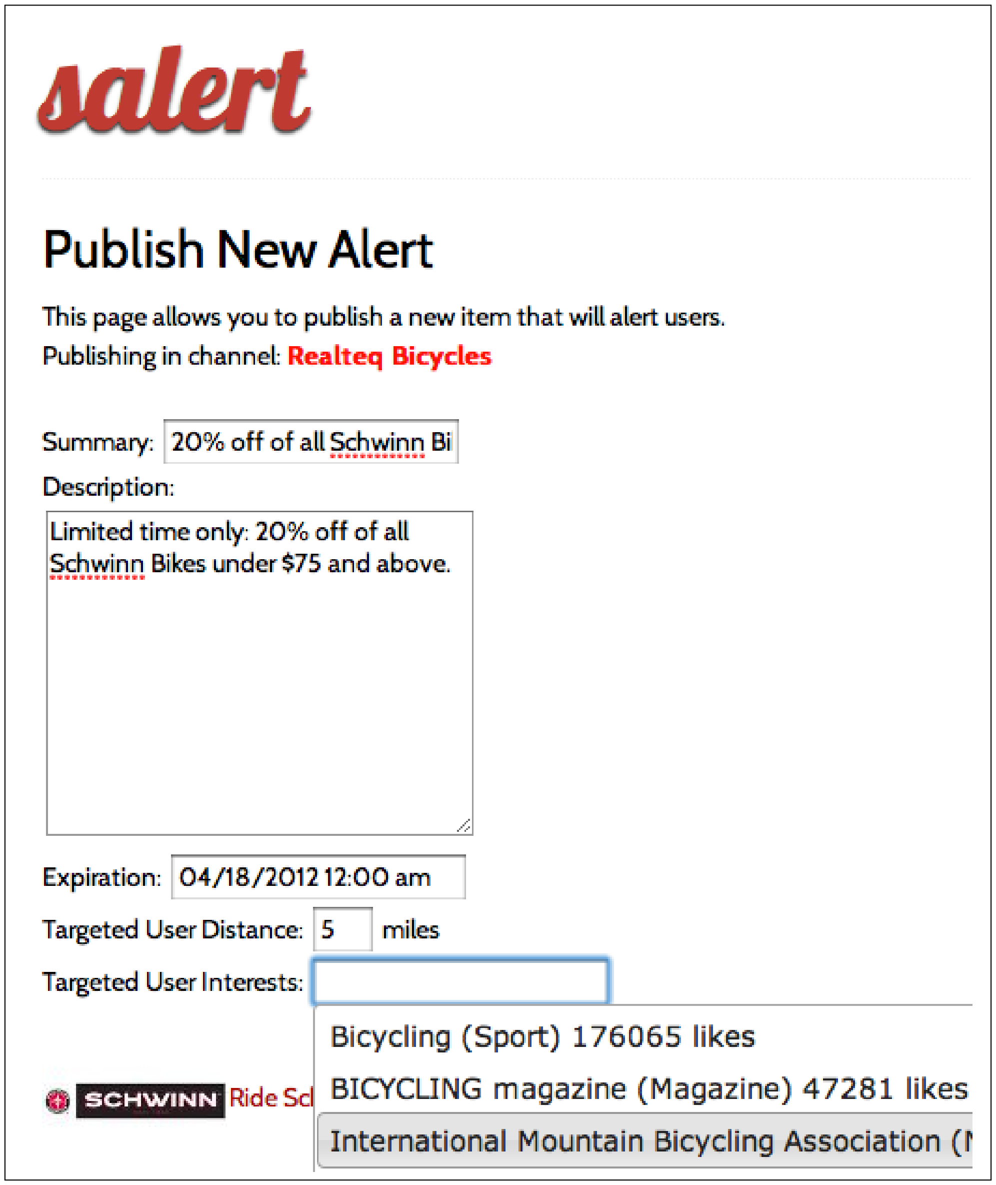
Realteq Bicycles - **20% off of all Schwinn Bikes** *Expires: April 18, 2012* 1 likes 0 dislikes

Rad Records - **10% off all Rock Albums** *Expires: April 26, 2012* 0 likes 0 dislikes

Figure 3.5: Local Items

In Figure 3.5 is a screenshot of the *Local Items* page of the web interface and corresponds to the Figure 2.3 mockup. The implementation is a fairly faithful representation of the mockup, with a key usability change to make the map a much more central piece of the overall composition of the page. In order to assist users in browsing the map, when the mouse is hovered on top of an item, the item corresponding to it in the list below is highlighted. Like the New Subscription page, I did not implement textual searching.

Figure 3.6 presents a screenshot of the *Publish New Alert* page of the web interface. This page is an implementation of the mockup presented in Figure 2.4, and is, overall, very similar to it. The Targeted User Interests field is an auto-completing input field that uses Facebook's Social Graph search API using an AJAX call to let publishers find Social Graph items corresponding to a search substring. When a user selects an item from the dropdown box, a FacebookItem model object is created, and the item's unique Facebook Graph identifier as well as various pieces of metadata from the JSON returned by the API, including a picture (if applicable), category, and number of likes are stored. The number of likes and category are displayed to help the information publisher pick the interests that have the widest coverage and are most applicable. This is important due to the sheer expansiveness of Facebook Graph data. The total number of elements represented within the Facebook Graph is unpublished and non-trivial to estimate due to the non-linearity of its identifier space, but the nodes and edges of this graph includes over 900 million users and 125 billion friendships between these users. In addition, an average of 3.2 billion likes and


salert

Publish New Alert

This page allows you to publish a new item that will alert users.
Publishing in channel: Realteq Bicycles

Summary: 20% off of all Schwinn Bi
Description:
Limited time only: 20% off of all Schwinn Bikes under $75 and above.
Expiration: 04/18/2012 12:00 am
Targeted User Distance: 5 miles
Targeted User Interests:

Bicycling (Sport) 176065 likes
BICYCLING magazine (Magazine) 47281 likes
International Mountain Bicycling Association (I

Figure 3.6: New Alert Page

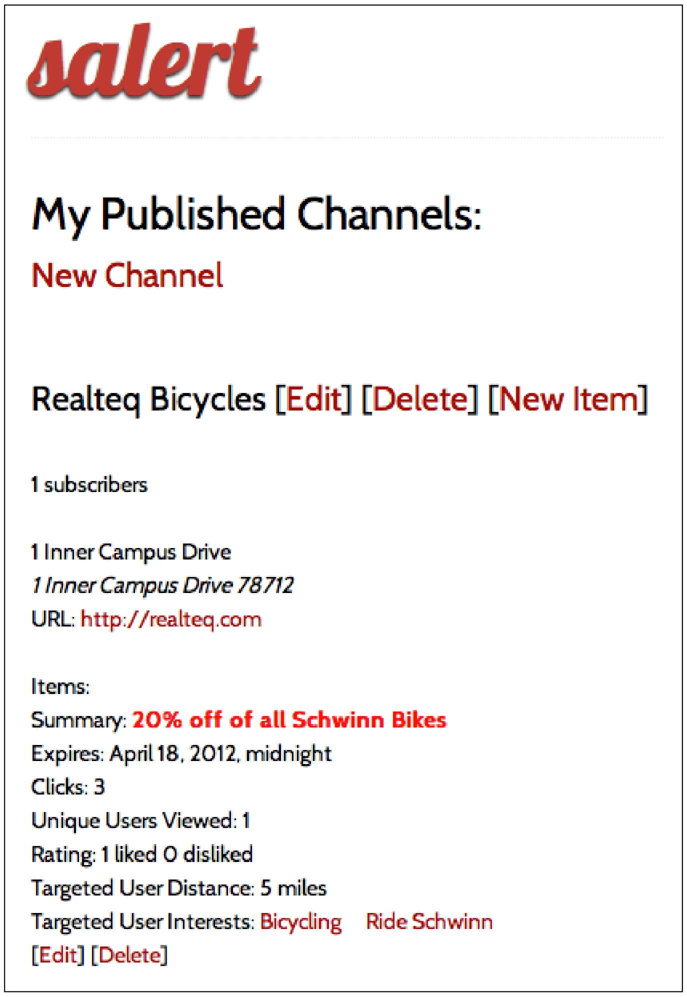
comments and 300 million photos were uploaded every day within the first quarter of 2012 ending March 31, 2012 [17]. Not all of this data, however, is publicly searchable by all users.

Figure 3.7 is a screenshot of the *My Channels* page of the web interface. This is an implementation of the mockup presented in Figure 2.5. As opposed to the mockup, which presents information in just an item-oriented view, information is shown about channel in the actual implementation as well, including a number of subscribers. In addition, a complete set of information about each item is presented at the cost of user interface compactness.

Figure 3.8 shows a screenshot of the *View Alert* page of the web interface and is an implementation of Figure 2.6. The implementation is very similar to the mockup, with the main exception that instead of a graphical cumulative rating, a numerical one is shown instead. Ratings are implemented using a upvote/downvote system where any single user has one vote per published item that they may change at any time.

Figure 3.9 is a screenshot of the *My Subscriptions* page of the web interface. This page allows the user to manage their subscriptions and see where alerts are being sent to, either through email or Short Messaging Service (SMS). This was another interface that I did not envision during the mockup phase.

Figure 3.10 is a screenshot of the *New Channel* page of the web interface. This allows information publishers to create and specify properties for a new



My Published Channels:

New Channel

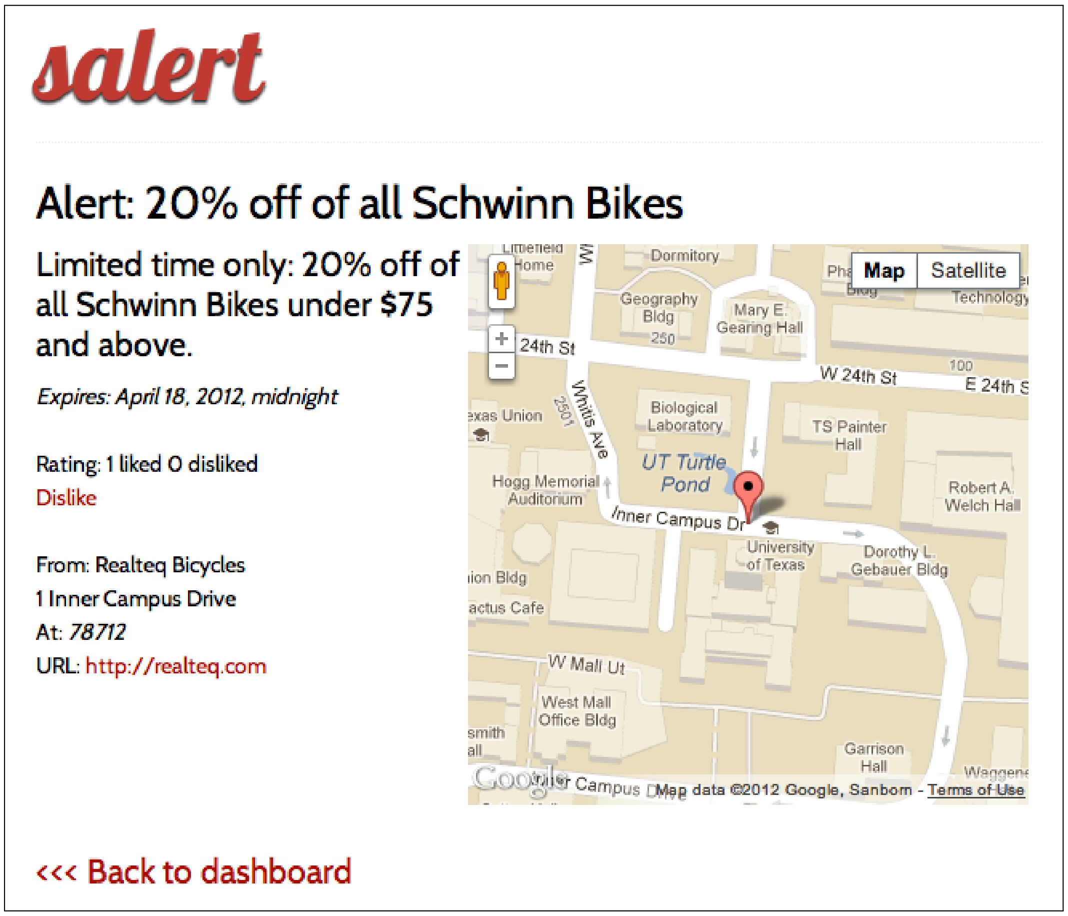
Realteq Bicycles [Edit] [Delete] [New Item]

1 subscribers

1 Inner Campus Drive
1 Inner Campus Drive 78712
URL: http://realteq.com

Items:
Summary: **20% off of all Schwinn Bikes**
Expires: April 18, 2012, midnight
Clicks: 3
Unique Users Viewed: 1
Rating: 1 liked 0 disliked
Targeted User Distance: 5 miles
Targeted User Interests: Bicycling Ride Schwinn
[Edit] [Delete]

Figure 3.7: My Channels



salert

Alert: 20% off of all Schwinn Bikes

Limited time only: 20% off of all Schwinn Bikes under $75 and above.

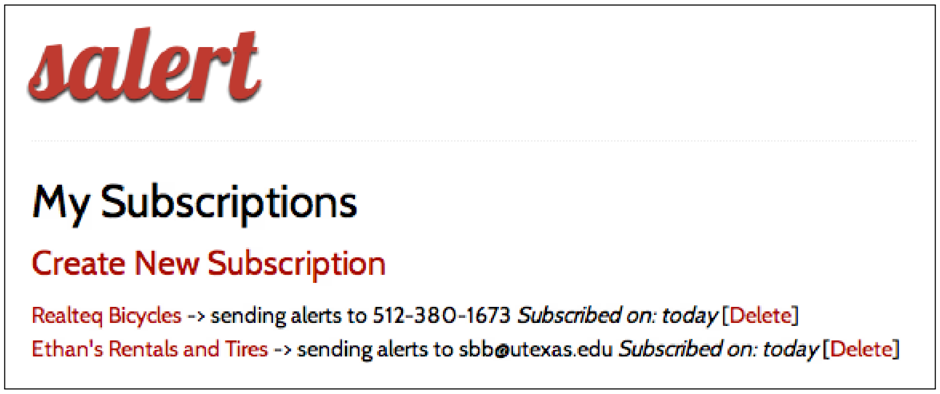
Expires: April 18, 2012, midnight

Rating: 1 liked 0 disliked
Dislike

From: Realteq Bicycles
1 Inner Campus Drive
At: *78712*
URL: http://realteq.com

<<< Back to dashboard

Figure 3.8: View Alert Page


salert

My Subscriptions

Create New Subscription

Realteq Bicycles -> sending alerts to 512-380-1673 *Subscribed on: today* [Delete]
Ethan's Rentals and Tires -> sending alerts to sbb@utexas.edu *Subscribed on: today* [Delete]

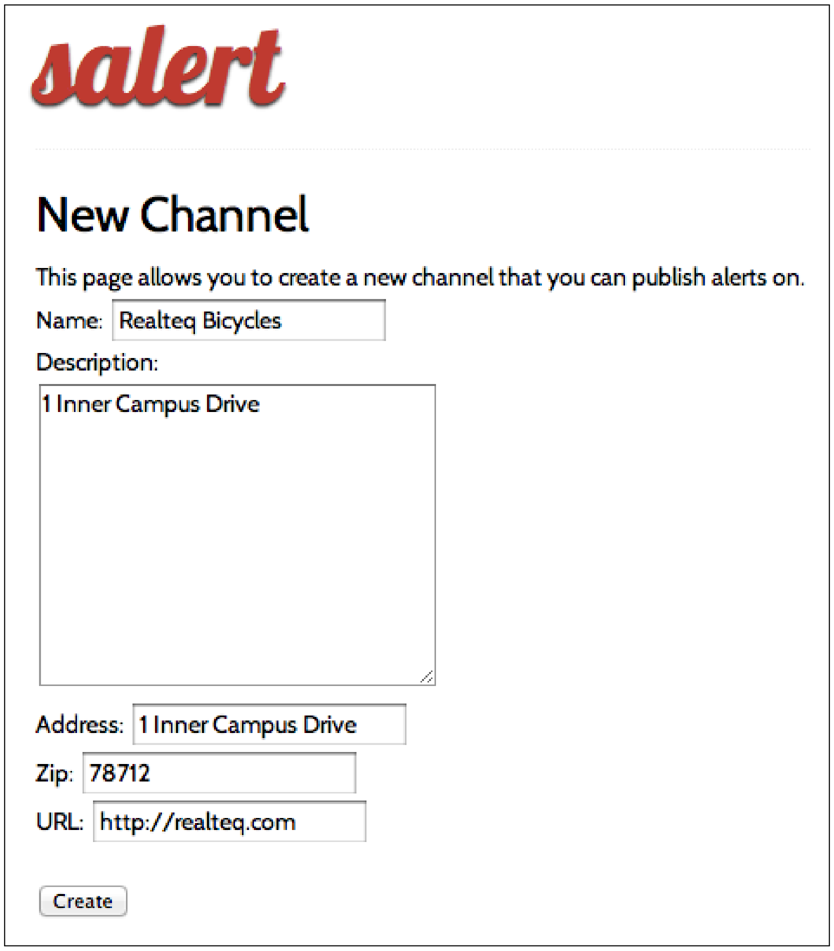
Figure 3.9: My Subscriptions

channel. This a key glue page that I did not mock up.

Figure 3.11 is a screenshot of the *Edit Channel* page of the administration interface within the site. The administration interface is optionally automatically created by the admin module within Django by introspection into any models defined by the Django application itself. This figure corresponds to the *Channel* data model within Figure 3.2. This interface allows specially tagged super-users to perform a multitude of administration related tasks without having to modify the database directly. This functionality is very configurable, although I used mostly defaults.

3.5 User Information Crawler

The User Information Crawler, whose basic operation is visualized in Figure 3.12, is a headless component within the sALERT architecture whose sole purpose is to retrieve and keep updated Facebook Social Graph about registered users. This is important to insure that user interest-based alert



New Channel

This page allows you to create a new channel that you can publish alerts on.

Name: Realteq Bicycles

Description:

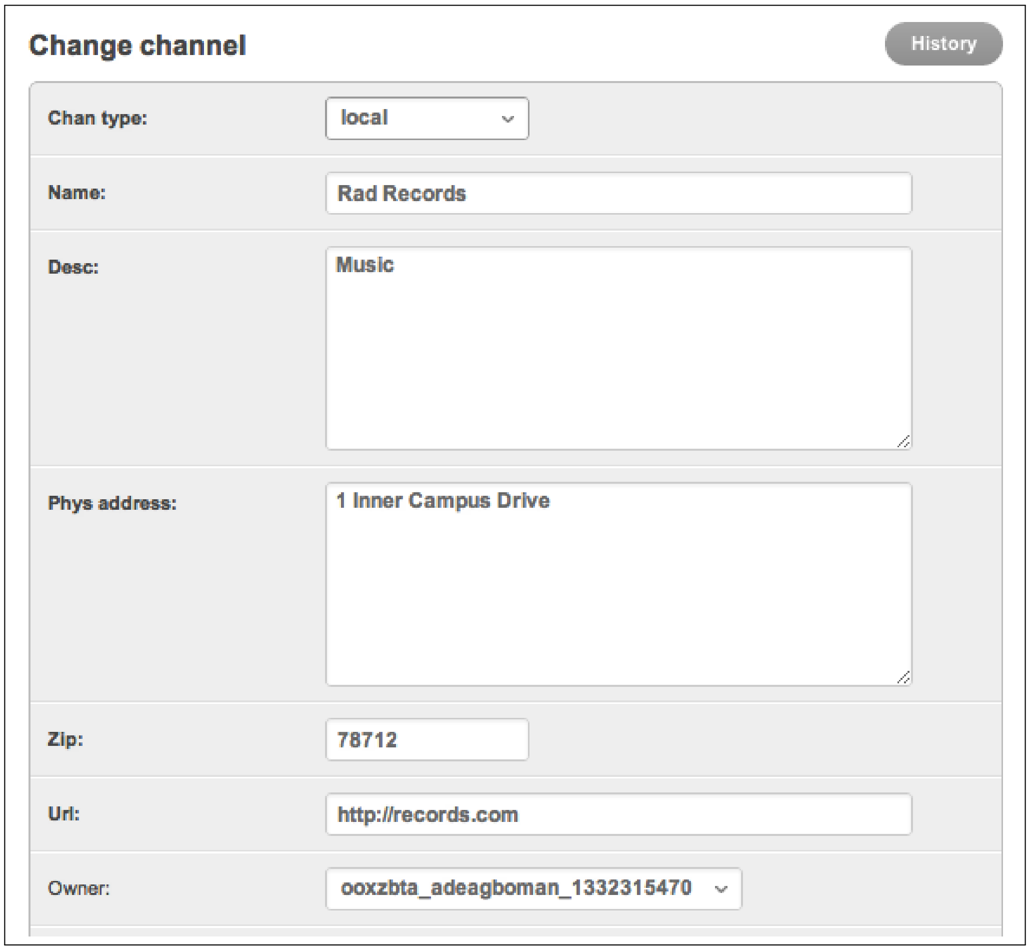
1 Inner Campus Drive

Address: 1 Inner Campus Drive

Zip: 78712

URL: http://realteq.com

Figure 3.10: New Channel Page



Change channel

Chan type:	local
Name:	Rad Records
Desc:	Music
Phys address:	1 Inner Campus Drive
Zip:	78712
Url:	http://records.com
Owner:	ooxzbta_adeagboman_1332315470

Figure 3.11: Administration Page

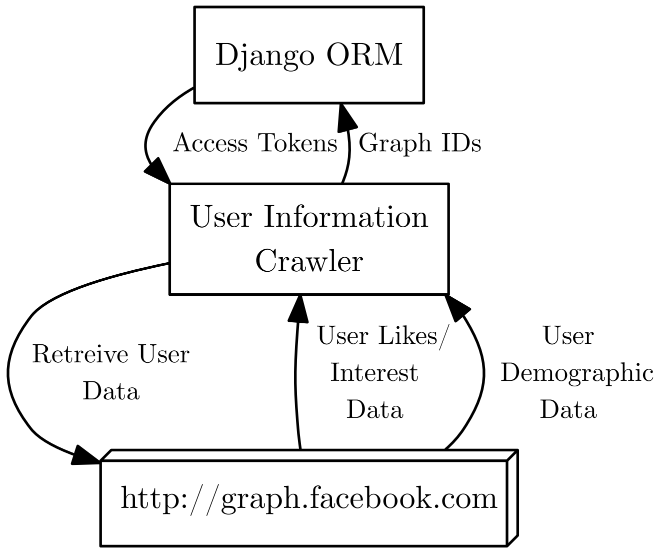


Figure 3.12: User Information Crawler

recommendations have the correct input data to trigger off of.

The crawler operates as a daemon process and periodically accesses access tokens that were generated when users use the *Login with Facebook* button in the web service and subsequently were stored within a UserProfile Django model. It then uses the access token to initiate authenticated requests on the user's behalf to obtain several pieces of information, such as user likes and activities and retrieval of demographic information from the user's Facebook profile, such as hometown, location, and the languages the user speaks.

Each of the discrete pieces of information retrieved from Facebook exists within its Social Graph, whether it is an activity such as singing, a location such as New York City, or a language such as Spanish. Some of these objects are created by Facebook itself by importing data from other sources into the graph, while others are created by its users, and yet others are automatically created through applications that interact with the Facebook platform. Each of these objects is associated with an unique numerical identifier. When the crawler asks for user interest information from within this graph, Facebook is essentially returning the objects that have edges emanating from the object that represents the user themselves. The crawler stores the object identifiers locally on a per-user basis for subsequent recommendation operations by the web service. The crawler uses a simplistic sequential update model that is effective when the pool of associated users is low, or when the periodicity of updates is small. Scalability is discussed in Section 5.3.4.

3.6 Internet Channel Crawler

The Internet Channel Crawler component of the sALERT architecture exists to power Internet-based channels. A summary of the workings of this component is presented in Figure 3.13. This component, like the User Information Crawler, is a headless daemon process and attempts to fulfill subscriptions to Internet-based feeds. It periodically iterates through the Subscription objects created in the Django ORM, filtering on subscriptions to Internet-based channels. It then invokes one of several different types of workers, depending on the source of the channels.

Currently, two types channel sources are supported, each of which are illustrative of two different input methodologies and serve as a template for others to be created in the future. The first of these is a Craigslist input source. This allows the user to input a Craigslist search URL to monitor. The crawler uses the RSS feed source that this website exposes. RSS is a XML-based schema that summarizes informational items from websites [33]. RSS feeds are often tagged with Globally Unique Identifier (GUID) elements that allow for discerning unique items being created within a website. I used the Python-based library Feedparser to handle translation of the raw RSS data to a higher level description. For each unique item within the feed, a MongoDB database is queried on a per-subscription basis for the GUID to see if the item had been encountered before. If not, the GUID is stored in the MongoDB database and the Alert Dispatcher described in Section 3.7 is asynchronously invoked. I expect this mechanism generalizes well to any feed in the RSS or the newer Atom feed formats, which is important as these feeds have become a widely adopted means of information syndication for websites. This would allow channels to be created trivially for a large set of websites that have these feeds.

The second type of Internet channel uses Twitter to allow users to receive alerts upon specific Twitter search phrases. Twitter is a microblogging-based social networking service which is commonly used by people to seek or share information, or share their daily activities [24], in the form of SMS-sized messages called "tweets." Like many newer web services, Twitter does not expose a RSS feed but rather exposes a rich API that third party software can use as a platform to build applications upon. Like Facebook, this API uses an OAuth2-based authentication to control access to a Representational

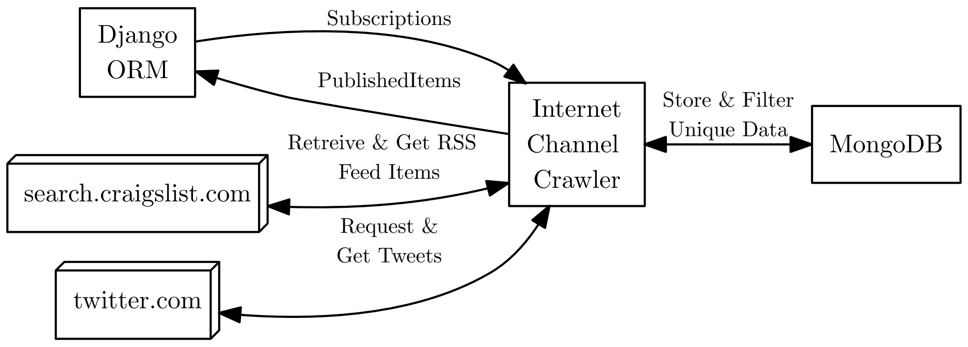


Figure 3.13: Internet Channel Crawler Summary

State Transfer-based (RESTful) API, except that in this case, a pre-authorized long-lived access token is stored that can be created on Twitter's developer website. I used the python-twitter library to provide high level access to this API. Twitter's search expressions are highly expressive, so users of this mechanism can search for phrases, specific topics (called hashtags), people, etc. Each tweet retrieved from the service is tagged with a unique number, which is used to demarcate old tweets from newly discovered ones.

3.7 Alert Dispatcher

The Alert Dispatcher is the last component of the sALERT implementation. This is asynchronously invoked by both the Web Service (i.e, for local channels, when the channel owner creates a new item) as well as by the Internet Channel Crawler (i.e, for remote Internet channels). A summary of the pipeline for this mechanism is shown in Figure 3.14.

Celery, a Python-based asynchronous task queueing library, is used to specify tasks to asynchronously invoke. Celery itself is a thin wrapper that provides a high level API and attempts to abstract out an actual transport and message passing mechanism, of which several backends exist. Celery defaults to RabbitMQ, a distributed message passing system. Two types of output tasks are defined within the component: one that can send emails, and one that can send text messages via SMS to a phone number. These tasks, which are actually implemented as simple Python functions, are registered with Celery and ultimately invoked by it via a RabbitMQ broker process.

The email task simply uses Python's built in SMTP module to send a email defined by the end user when a subscription is set up. The body of the email contains a link to the alert that is created by whatever component initiated the creation of an alert, either the Web Service or Internet Channel Crawler. The text message task works similarly, but instead of interacting with an SMTP server, it interacts with a web service called Twilio [21], which offers a number of telephony-related web APIs, including the ability to send text messages at a trivial cost per message. I used Twilio's official Python-based SDK to interact with Twilio's lower level RESTful APIs.

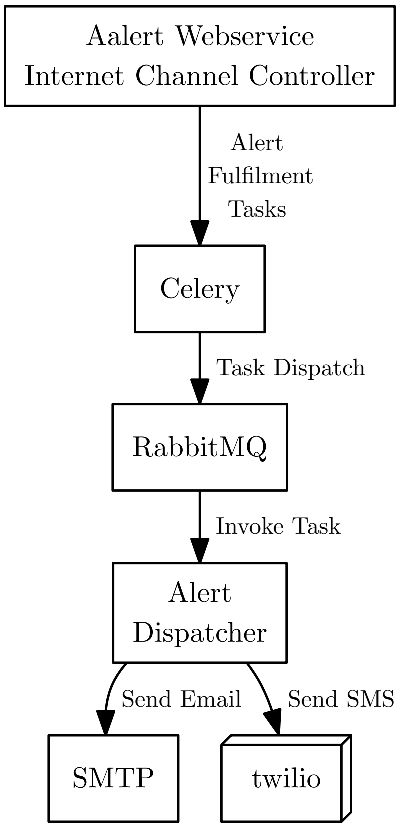


Figure 3.14: Alert Dispatcher

Chapter 4

Results

4.1 Benchmarks

I developed a suite of tools to use the Facebook API's Test User functionality to simulate real world usage of the implemented system. This is a mechanism that Facebook provides as part of its platform to allow application developers to test their applications by being able to programmatically create and modify up to 500 virtual Facebook accounts that are invisible to the rest of the network, but can see each other [16]. My tools included dynamic test user creation, deletion, and listing all test users associated with an application. The latter action is needed to periodically refresh test user login URLs that can be used to login to test user accounts using the Facebook website. Unfortunately, the test user API functionality does not currently allow for the creation of new interests to be associated with an account, so this had to be done by hand by logging in to the test user.

In order to assess how the current implementation would perform for a small amount of users using the system simultaneously, I performed a basic benchmark test. I created five test users using these tools and associated them with a variety of interests on the Facebook website. I then created a test bootstrap script that used direct access to Django's ORM to create ten test local channels and subscriptions (to SMS) for all of these users to those channels. Lastly, I created another script that also functioned using direct access to Django's ORM to publish 50 items over a 60 second test window into these test channels.

To bootstrap the test, I logged into the sALERT website using each of the test users using different browsers, and then used the test bootstrap script to create test channels and subscriptions. I then measured various pre-test memory metrics, which are summarized in Table 4.1. I then ran the test script to publish alerts, while simultaneously viewing the dashboard for each of the browsers as to generate recommended items. After the total number of expected SMS messages had been sent, post-test memory usage metrics were measured.

I measured memory usage per-process by examining the output of the `ps` command, while examining the virtual memory usage field. In some cases, such as `apache` and `celeryd`, there were multiple forked copies of processes, and thus the average and range are presented in these situations. I used the `free` command to examine total system memory usage. Determining the total SQL queries performed in this period was made possible by Django's ORM-level debugging functionality. The amount of text messages and timing is made possible by logging into Twilio's online account management tool [21], which allows tracking of API usage.

The EC2 instance in this case was a *micro* level, which is allocated a

Name	**Type**	**Value**
Apache	Memory Usage Pretest	260980k $\pm$ 271k
Apache	Memory Usage Posttest	261852k $\pm$ 226k
mongod	Memory Usage Pretest	132204k
mongod	Memory Usage Posttest	132204k
celeryd	Memory Usage Pretest	33340k $\pm$ 12845k
celeryd	Memory Usage Posttest	57936k $\pm$ 1091k
mysqld	Memory Usage Pretest	137612k
mysqld	Memory Usage Posttest	139109k
RabbitMQ	Memory Usage Pretest	21112k
RabbitMQ	Memory Usage Posttest	22493k
Internet Feed Crawler	Memory Usage Pretest	24640k
Internet Feed Crawler	Memory Usage Posttest	24633k
User Information Crawler	Memory Usage Pretest	27752k
User Information Crawler	Memory Usage Posttest	28101k
System	Total Memory Usage Pretest	309140k
System	Total Memory Usage Posttest	314752k
mysqld	Total SQL Queries	751
Twilio	Total Messages Spent	250
Twilio	Total Time Spent	63 $secs$

Table 4.1: Summary of Benchmarks

total of 613 MB of memory, of which roughly half was used. Most of the memory was used while the system was in steady-state, due to the large number of different services being used in this project, and mostly default settings within the servers being used. In this particular test, there was only a modest memory usage increase post-test. The sum of the per-process memory usages does not add up to the total system memory used because the former is tracking virtual memory, and most of the processes have shared libraries virtually mapped to the process that is only used once in physical memory. In the future, it may be advisable to sample memory usage at regular intervals during the benchmark process, as to measure peak memory usage before memory is freed or garbage collected in different languages. However, it appears the EC2 *micro* level seems to usable for the current needs.

4.2 Lines of Code Analysis

In order to analyze the lines of code of the system with fidelity, each of the source files in the sALERT tree, sans modified 3rd party libraries, were placed in different categories corresponding to which of the system's logical components they best matched to as presented in Figure 3.1. An additional category was created for files whose code was used by multiple components or was test code. Lines of code information was extracted using the common UNIX `wc`, and subdivided depending on language. This is presented in Table 4.2.

Component	**Files**	**Python**	**HTML/JavaScript**	**CSS**
Shared Code	4	414	-	-
Web Service	27	1059	1516	91
Internet Channel Crawler	3	226	-	-
User Information Crawler	6	216	-	-
Alert Dispatcher	7	129	-	-

Table 4.2: Lines of Code

4.3 Software Change Management

The git version control system was used to track changes in this project. Git is a Distributed Version Control System (DVCS) system that was originally created by Linus Torvalds to be the official version control software for the Linux Kernel, but has since spread rapidly to other projects. A DVCS such as Git is especially useful for teams of developers, but even for a single developer, it offers advantages over conventional version control systems. For example, Git branches are very lightweight, and it is easy to change between them [4]. The typical developer workflow evolves to creating many branches when working simultaneously on different topics within a single source tree. Additionally, Git has a staging area for commits, and offers intelligent merging algorithms.

Since the Fall of 2011, a total of 33 commits were made to the master repositories tracking the implementation's codebase until March 24th, 2012. In addition, 9 local branches were created where most commits initially occurred before merging back into the master.

4.4 Runtime Cost Estimate

There are two driving operating costs for the current implementation of the system: cloud infrastructure and web services. It is convenient to estimate these costs in some sort of per user basis. In this section, a model is formed for estimating the cost of running the service for 100 users using the current infrastructure in place.

To keep this model simple, the following general assumptions are made:

1. An user will have 5 subscriptions they are subscribed to at any given time.
2. An user receives all notifications over SMS.
3. The source of each subscription generates two new items per day.
4. A month has 31 days.

To estimate the cost of cloud infrastructure, Amazon's extensive cost estimator tools can be used to generate "what if" scenarios for a variety of AWS services. Most of cost of EC2 is tied to the compute power (which, on EC2 is also tied to maximum memory available) rather than bandwidth or disk space, as can be common with other hosting solutions. For example, the cheapest instance, the *micro* instance that is always on, has a 1 GB persistent volume attached to it and a 5 GB of both input and output network transfer and is billed at a $15.89/month rate. On the other hand, the compute power is billed at \$14.64, the storage is billed at \$1.25, and the data transfer is billed at \$1.32. Only when the monthly network transfer is raised to 230 GB/mo (115 GB in, 115 GB out), does the cost of the monthly network transfer eclipse the cost of the compute power. Meanwhile, going from a *micro* instance to the next level up, a *small* instance (which provides 1.7 GB of memory, 160 GB of built-in non-persistant storage, and 1 guaranteed virtual CPU core) raises the compute cost to \$58.56/month. I expect that the *micro* instance would suffice for some time, especially if configuration parameters were tweaked from default values for a lower memory situation.

The only commercial web service being used at the moment is that of Twilio. Twilio's cost structure is relatively simple, with a fixed cost of a one cent per message (with a free \$30.00 credit upon opening an account) and increasing volume discounts, culminating at 0.2 cents per message over 100 million messages sent/month. To estimate the number of messages sent using the above assumptions, the number of users can be multiplied by the average number of subscriptions and the average number of new items per day for each subscription. In this case, this would amount to 1000 messages per day, or \$310.00 per month at a cent per message rate. This volume is not high enough for volume pricing, so no other adjustments are necessary in this estimate. In order to avoid this cost, other notification schemes may need to be explored such as direct mobile pushes to phones. Some of these schemes are explored in Section 6.2.

Chapter 5

Conclusions

5.1 Summary

The implemented system satisfies the criteria laid out by the vision and most of the criteria within the specification. The following major objectives outlined in the vision were achieved:

- Users can be alerted of information and events from various sources that correspond to their interests.
- Users may receive mobile notifications based on these alerts.
- Information publishers may send out information items and target them to specific users.
- Social network-derived data is used to match these two user groups.

5.2 Discussion

While a famous software engineering axiom as stated by Knuth is that "premature optimization is the root of all evil" [26], many modern libraries traditionally thought of as within the domain of scalability or optimization are well designed enough that it is often better to use them early than later.

This was the case in this project with the usage of the Celery/RabbitMQ stack to perform asynchronous task handling. I used these somewhat late in the development process as there was a hesitancy to use it for fears of unnecessary complexity. However, the stack turned out to have a high benefit to learning curve and complexity ratio. Because of time constraints, I only used this stack for scheduling the asynchronous handling of alert notification dispatching, but using it in the crawler components to handle the scheduling of various periodic tasks would have saved having to write custom controller code. I would have saved net development time savings with an additional benefits of less brittle code, more flexibility and a great deal more scalability.

I found Django to be a highly productive framework. My main problems with it were not found in the core libraries itself, but in the form of the `django_facebook` library, which was not always updated to whatever API changes Facebook had made recently. In hindsight, I should have explored other Django or Python-based libraries that wrap the Facebook APIs to see if there were any that were more actively maintained.

5.3 Future work

In this section, possibilities for future work on the sALERT system are discussed.

5.3.1 Quality

The current recommendation system is simplistic in nature, and although it works for various scenarios, more complex algorithms would serve well to provide higher quality and more comprehensive recommendations. Some of the possible avenues are discussed in Section 6.1 and involve either matching users to similar users, or interests to other similar interests. From the perspective of the system, the data needed for these algorithms could be gained by more comprehensive retrieval of information from Facebook and other social network sources as well as subsequent data mining-related post processing steps. The information retrieved could include links between users themselves, such as friendship and group relationships. Additionally, recommendations can be made more comprehensive by including items from Internet-based channels. This would need richer parsing of metadata (e.g., geographical location, content topics/interests, etc) from those sources or interpretation of metadata from the contents of items when metadata is not available.

Deeper integration into social networks themselves may present opportunities for quality improvement. Facebook offers different levels of integration with the service [40]. At the deepest level, applications can run in top of Facebook's Canvas API, which allows applications to leverage Facebook's look and feel and run within a HTML iframe within the Facebook site itself. Using this in the web service, even in a limited fashion, may encourage user participation. For a more data-oriented integration, Facebook's Graph API allows applications to objects into the Facebook Social Graph complete with pieces of metadata, which allows those objects to be viewed and be searched for on the Facebook website as well as be usable by other third party applications. This may be useful for various constructs such as published items.

5.3.2 Usability and Functionality

In addition to usability-related feedback from end users, there are several general areas that require usability-related improvement. The current web interface is not optimized for a large number of channels, items, and alerts, and can become unusable with many of these elements. This may happen, for example, if there are many channels in a single locality being shown in a map, or if an user has too many active alerts. As the service grows, this will be increasingly important problem to address to avoid information overload. This could be solved with simple techniques such as pagination, text-based search capabilities, and limiting item display to a maximum amount of items at any one time.

I proposed System Channels as a piece of functionality during initial design stages but I did not ultimately implement them. These channels could improve the usefulness of the system to users while leveraging much of the existing infrastructure. Examples of these types of channels include allowing the system to create synthetic alerts at specific dates that are sent as notifications to users as well as mobile location based alerts. The scope and relevancy of these pieces of functionality will have to be decided.

The system currently uses United States zip codes extensively. Ultimately, I may want to internationalize this service. In this case, various Geographic Information System (GIS) systems may be beneficial for both more efficient geographical coordinate conversions, but more easily internationalized mechanisms. Django includes a set of modules that may simplify this task by interfacing with a number of GIS libraries and services.

5.3.3 Commercialization

Approaches to marketing and subsequent monetization of this system will be discussed in this section.

Marketing the product system is an important means to an end to generate a critical mass of users needed for the service. As an important part of the service is geospecific, focusing on a specific initial locality for a pilot program may be important. The area around The University of Texas at Austin campus itself may be a good candidate for this program owing to the density of people and businesses, including a particularly social network savvy student base. Initially, low cost mechanisms such as fliers will be used as a means of marketing to prospective users around campus, combined with reach-out partnership programs to local business owners to encourage publisher participation.

There are a number of ways that are possible to monetize the service itself. An initial solution may be to use online advertisement services such as Google Adsense [20] in the web service, which delivers targeted ads to users and ultimately revenue to site owners based on advertisement clicks. Further work on mobile-development may also open doors to mobile advertisement platforms such as Google Admob [19], which are tailored to mobile applications and services.

Paid services may also be a monetization opportunity. These services may be in the form of parts or the entire site being available on a subscriber model to specific user bases, such as commercial entities wishing to act as subscribers. This may be tiered, for example, by offering extra functionality and features such as analytics information, preferred recommendations, and a larger maximum amount of published items. Furthermore, other services, such as API-level access to the system, may be possible to monetize off of.

5.3.4 Scalability

The Amazon AWS architecture includes a number of ancillary cloud infrastructure technologies usable on EC2 that allow applications to be quickly scaled up in both users and functionality. Currently, the most pressing short term need is for server persistence. This is needed because EC2 instances are not inherently stateful across reboots. This is typically solved using Amazon's Elastic Block Store service [13], which allow virtual volumes to be attached and persisted as underlying EC2 instances get created and destroyed. Amazon's Elastic Load Balancing technology could be used to spread web traffic across multiple EC2 instances according to flexible load and geographical constraints (for example, to handle all requests in the western United States on Amazon's data centers in the west). This may be combined with EC2 Auto Scaling mechanism to automatically create and destroy EC2 instances depending on load [14], effectively generating computational power on demand. The AWS architecture includes a number of other services that may be explored including scalable email sending, database, deployment, and task management technologies.

As the number of users and subscriptions increases, API usage restrictions of various data sources may be hit. This can be managed in several ways. In RSS feeds, subscriptions to popular feeds can be handled in a feed-centric rather than a strictly subscription-centric manner. This could by done by periodically fetching remote feeds into a local caches and having similar subscriptions use this cache as an input source. This would alleviate problems with fetching remote feeds too many times. Social networks such as Facebook and Twitter also have similar usage restrictions with their APIs. They both give developers similar tools to solve this issue by letting high usage applications setup subscriptions for updates to specific data pieces as they change, instead of traditional polling mechanisms [16, 22].

Chapter 6

Related Work

6.1 Personalization

Recommendation engines, or systems which aim to recommend items to users, have rapidly evolved within the last several decades. They have been of great interest to both researchers and to production environments. The importance of recommendation systems is perhaps exemplified by the Netflix Prize, announced in 2006, which promised a \$1 million prize to any group which could improve on Netflix's Cinematch movie recommendation system by more than 10% [23], which was ultimately won in 2009 [27]. Two different broad categories have been have been described for these systems [3]: content-based, where users are recommend items similar to items they have liked in the past, and collaborative, in which users are recommend items from similar users. In the context of social networks, both may be applicable, as the former approach needs preference information from a specific user, and the later needs preference information from multiple users, and possibly the relationships between the users as well. Both of these pieces of data are commonly tracked by social networks.

One of the most important and influential early systems for collaborative recommendations was that of the Ringo system [38], which attempted to make personalized music recommendations based on a holistic approach by considering the overall interest profile of a user and comparing it to other users, in essence trying to capture the intuition behind "word of mouth". The authors found that the system worked increasingly well as more users trained the system.

Recommendation systems have made their way into many commercial environments as well. Besides the aforementioned usage by Netflix for movie recommendations, other notable examples include Amazon, who uses collaborative recommendation systems to extensively personalize its e-commerce site [30], and Google, which incorporates these systems to produce personalization functionality in products such as Google News [6]. Recommendation systems are emerging in a newer sector, "daily deal" services such as GroupOn and LivingSocial, which have gained prominence in recent years. These websites originally focused on showing deals from a person's locality, but now offer increasing amounts of content personalization.

6.2 Notification

With the rise in popularity of mobile devices and the increasing ubiquity of electronic communication, there has been an increased interest in the development and usage of notification systems. These systems have a number of technical challenges such as information delivery and scalability.

Location based notification systems have been an emerging field that have been popularized recently due to geolocation devices and services being found on mobile devices. When users arrive at specific geographical locations, they may be sent notifications through these systems. Munson and Gupta [32] argued that these systems should be highly generalized, and could be useful for a wide variety of applications, including automatic information about landmarks delivered to tourists as they walk around, public safety notifications when drivers are close to inclement weather, and utility companies sending information to all close by customers when maintenance is about to occur.

Mobile devices increasingly support the Multimedia Messaging Service (MMS), an evolution of SMS which allows for much richer content to be delivered to users. It has been argued that typical notification messages sent over SMS are not very informational, and require a user to visit a URL in order to gauge a notification's value [37], which may ultimately decrease follow rates. The usage of richer messages delivered over MMS may eliminate this problem.

In addition to classic mobile messaging services such as SMS and MMS, the concept of push notification has been an important innovation in notification services and have become common in mobile applications. These notifications are typically implemented using mobile-platform specific services, such as the Apple Push Notification Service (APNs) [15] and Android Cloud to Device Messaging (C2DM) [18], on the iOS and Android platforms respectively. Platform-agonistic services have also been proposed [9] that piggyback on top of the platform specific services.

Appendix A

An Overview of Django

A.1 Introduction

Django is a web application framework that was originally developed in-house by the Lawrence Journal-World, a newspaper in Lawrence, Kansas, and was created to power its online presence, which included a number of news and local entertainment websites. After several years of development behind closed doors, it evolved into a more general form and was subsequently spun off as an open source project and released to the public in July 2005 [12].

At its core, Django consists of three major layers: a model layer where data is stored and represented, a view layer in which program logic occurs, and a template layer, where presentation may be expressed. Each of these is explored in the following sections.

A.2 Models

Django includes a complete ORM, with the ability to interface with different database backends such as MySQL, SQLite, and PostgreSQL. In addition, a fork of the framework, called django-norel, is underway to port the Django ORM to non relational databases such as that of Google App Engine, Apache Cassandra, Apache HBase/Hadoop, Apache CouchDB, and MongoDB, with the long term goal to merge these changes back into the official Django release [2]. These databases, collectively referred to as NoSQL-databases, have gained market traction recently [29]. While leveraging the Django ORM, developers write small Python-based classes to describe pieces of data and the relationships between them. These are referred to within the Django lexicon as “models”. The main purpose of Djangos ORM is to abstract away database details so that the developer can work with high-level objects rather than lower level database constructs. Except for exotic situations such as importing a large set of records *en masse*, Django-based applications do not need to be concerned with the details of the actual database they are ultimately storing data into.

The Django ORM classes support the classic Create Replace Update Delete (CRUD) operations on these models by exposing an API. In addition, a variety of operations are supported for querying for existing data. Typically, developer declared models eventually derive from the Django ORMs “model” class somewhere in their inheritance hierarchy. Fields declared within a model are translated into database columns, and relationships between different models are annotated through special types of fields. Code reuse is promoted by allowing models to derive from other models, in which case a child model automatically inherits the fields of a parent model.

A.3 Views

The business logic for a Django program is typically defined in what are called “views”. Views are defined by developers as simple Python functions that operate on a request object, and return a response object. Views are the most analogous construct within Django to CGI scripts and servlets.

Django supports a URL dispatch mechanism to map URLs to specific view functions. A Django application defines a file that contains regular expressions for this mapping. Individual parts of the URL can be captured and sent to the resultant view function using Pythons keyword variable argument feature. This is commonly utilized to support the creation of “pretty” URLs for search engine optimization (SEO) purposes, or for the implementation of RESTful APIs.

Response objects are constructed and returned by view functions based on the request parameter and any other arguments passed to the function. The request object stores information such as HTTP header information. The values represented by the response object are returned back to the web server, and eventually the client. These objects may represent textual information in the form of html, JSON, XML, or other data types, or they may be HTTP-based control statements such as redirects and errors. Typically, views will perform CRUD and querying operations on models in order to build up a correct response.

A.4 Templates

Django includes a templating system. This system is typically used by applications to generate dynamic HTML, although they can also instead generate other textual data types such as JSON, XML, CSV, etc [12]. Commonly, templates are invoked by view functions, and data in the form of Python objects, including model objects, can be passed from the view to the template.

Templates support the concept of variables, which are the pieces of data passed from the view. Additionally, global state is supported through the concept of a context processor. These are typically used to support aspects of web programming such as sessions and cookies. As variables exposed to the template are actual Python objects, the template system also allows accessing fields and invoking methods. Variables are read only within Django templates, however, the system also supports the concept of a filter, which are akin to UNIX shell pipes. A variable can be sent into a filter before the value is read. Django has a multitude of built-in filters for transforming data in various ways, such as justifying text and reformatting absolute dates into relative ones.

The last major construct in Django templates is called tags. These are used to define control flow logic within a template. Typical flow and conditional statements are supported such as if and for statements, and variables may be used as parameters for these conditional and iteration statements.

Tags are also used to implement the concept of blocks. Templates are allowed to extend other templates, and blocks allow templates to define regions that are replaced wholesale in either other templates or in the same one. This is commonly used, for example, to create a base template for an entire website that has a number of blocks corresponding to customization points, and allow individual templates corresponding to different subpages to only contain the blocks those pages need to customize. This technique for implementing inheritance allows for a more consistent look and feel within a web application while reducing code duplication and was used in the implementation.

A.5 Other

When it comes to the breadth of functionality in its standard libraries, the Python language has a "batteries included" philosophy [8]. Django follows this philosophy, and includes a large number of features that support different uses cases of web applications within the `django.contrib` set of modules that Django applications can optionally use. These include HTML form abstractions, generic views, widgets, a built-in admin interface, session support, unit testing support, authentication, etc.

Glossary

AJAX Asynchronous Javascript and XML. 21, 34

AMI Amazon Machine Image. 20

AWS Amazon Web Services. 20, 53, 60, 61

CGI Common Gateway Interface. 20, 21, 67

CRUD Create Replace Update Delete. 66, 67

DRY Don't Repeat Yourself. 21

DVCS Distributed Version Control System. 52

EC2 Elastic Compute Cloud. 18, 20, 53, 60

GIS Geographic Information System. 59

GUID Globally Unique Identifier. 44

IaaS Infastructure as a Service. 18

JSON JavaScript Object Notation. 21, 34, 67

MMS Multimedia Messaging Service. 64

MVC Model View Controller. 20–22

OAuth2 Open Authorization. 24, 25, 44

ORM Object Relational Model. 21, 65, 66

RESTful Representational State Transfer-based. 44, 46, 67

RSS Really Simple Syndication. 10, 43, 44

SMS Short Messaging Service. 36, 44, 46, 49, 53, 64

W3C World Wide Web Consortium. 27

Bibliography

[1] United States Census Bureau Gazetteer. `http://www.census.gov/geo/www/gazetteer/gazette.html`, 2010.

[2] NoSQL Support for Django. `http://www.allbuttonspressed.com/projects/django-nonrel`, 2012.

[3] G. Adomavicius and A. Tuzhilin. Toward the Next Generation of Recommender Systems: A Survey of The State of the Art and Possible Extensions. *Knowledge and Data Engineering, IEEE Transactions on*, 17(6):734–749, 2005.

[4] S. Chacon. *Pro Git*. Springer, 2009.

[5] M. Cohn. *User Stories Applied: For Agile Software Development*. Addison-Wesley Professional, 2004.

[6] Abhinandan S. Das, Mayur Datar, Ashutosh Garg, and Shyam Rajaram. Google News Personalization: Scalable Online Collaborative Filtering. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 271–280, New York, NY, USA, 2007. ACM.

[7] Nick Doty, Deirdre K. Mulligan, and Erik Wilde. Privacy Issues of the W3C Geolocation API. *CoRR*, abs/1003.1775, 2010.

[8] P.F. Dubois. Python: Batteries Included. *Computing in Science & Engineering*, 9(3):7–9, 2007.

[9] Huber Flores, Satish Narayana Srirama, and Carlos Paniagua. A Generic Middleware Framework for Handling Process Intensive Hybrid Cloud Services from Mobiles. In *Proceedings of the 9th International Conference on Advances in Mobile Computing and Multimedia*, MoMM '11, pages 87–94, New York, NY, USA, 2011. ACM.

[10] E. Hammer-Lahav, D. Recordon, and D. Hardt. The OAuth 2.0 Authorization Protocol. Draft RFC, March 2012.

[11] A.E. Hassan and R.C. Holt. A Lightweight Approach for Migrating Web Frameworks. *Information and Software Technology*, 47(8):521–532, 2005.

[12] A. Holovaty and J. Kaplan-Moss. *The Definitive Guide to Django: Web Development Done Right*. Springer, 2009.

[13] Amazon.com Inc. Elastic Block Store. `http://aws.amazon.com/ebs`.

[14] Amazon.com Inc. Elastic Load Balancing. `http://aws.amazon.com/elasticloadbalancing`.

[15] Apple Inc. Apple Push Notification. `https://developer.apple.com/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG`.

[16] Facebook Inc. Facebook API Documentation. `http://developers.facebook.com/docs/reference/api`.

[17] Facebook Inc. SEC S-1/Amendment Filing. `http://www.sec.gov/Archives/edgar/data/1326801/000119312512175673/d287954ds1a.htm`.

[18] Google Inc. Android Cloud to Device Messaging Framework. `https://developers.google.com/android/c2dm/`.

[19] Google Inc. Google Admob. `http://www.google.com/ads/admob/`.

[20] Google Inc. Google Adsense. `http://www.google.com/adsense`.

[21] Twilio Inc. Twilio Cloud Communications. `https://www.twilio.com`.

[22] Twitter Inc. Twitter Developer Documentation. `https://dev.twitter.com/docs`.

[23] D. Jannach, M. Zanker, A. Felfernig, and G. Friedrich. *Recommender Systems: An Introduction*. Cambridge Univ Press, 2010.

[24] Akshay Java, Xiaodan Song, Tim Finin, and Belle Tseng. Why We Twitter: Understanding Microblogging Usage and Communities. In *Proceedings of the 9th WebKDD and 1st SNA-KDD 2007 Workshop on Web Mining and Social Network Analysis*, WebKDD/SNA-KDD '07, pages 56–65, New York, NY, USA, 2007. ACM.

[25] M. Jazayeri. Some Trends in Web Application Development. In *Future of Software Engineering, 2007. FOSE'07*, pages 199–213. IEEE, 2007.

[26] D.E. Knuth. Computer Programming as an Art. *Communications of the ACM*, 17(12):667–673, 1974.

[27] Y. Koren. The Bell-Kor Solution to the Netflix Grand Prize. *Netflix Prize Documentation*, 2009.

[28] James A. Landay and Brad A. Myers. Interactive Sketching for The Early Stages of User Interface Design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '95, pages 43–50, New York, NY, USA, 1995. ACM Press/Addison-Wesley Publishing Co.

[29] N. Leavitt. Will NoSQL Databases Live Up to Their Promise? *Computer*, 43(2):12–14, Feb. 2010.

[30] G. Linden, B. Smith, and J. York. Amazon.com Recommendations: Item-to-item Collaborative Filtering. *Internet Computing, IEEE*, 7(1):76–80, 2003.

[31] Balsamiq Studios LLC. Balsamiq Mockups. `http://www.balsamiq.com`.

[32] Jonathan P. Munson and Vineet K. Gupta. Location-Based Notification as a General-Purpose Service. In *Proceedings of the 2nd International Workshop on Mobile Commerce*, WMC '02, pages 40–44, New York, NY, USA, 2002. ACM.

[33] S. Murugesan. Understanding Web 2.0. *IT professional*, 9(4):34–41, 2007.

[34] R.L. Nord and J.E. Tomayko. Software Architecture-Centric Methods and Agile Development. *Software, IEEE*, 23(2):47–53, 2006.

[35] A. Popescu. Geolocation API Specification. *World Wide Web Consortium, Candidate Recommendation CR-geolocation-API-20100907*, 2010.

[36] B.P. Rimal, Eunmi Choi, and I. Lumb. A Taxonomy and Survey of Cloud Computing Systems. In *NCM '09. International Conference on Networked Computing, Advanced Information Management and Digital Content and Multimedia Technologies*, pages 44–51, Aug. 2009.

[37] S. Samanta, J. Woods, and M. Ghanbari. Special Delivery: An Increase in MMS Adoption. *Potentials, IEEE*, 28(1):12–16, January-February 2009.

[38] Upendra Shardanand and Pattie Maes. Social Information Filtering: Algorithms for Automating Word of Mouth. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '95, pages 210–217, New York, NY, USA, 1995. ACM Press/Addison-Wesley Publishing Co.

[39] R. W. Sinnott. Virtues of the Haversine. *Sky and Telescope*, 68:158, 1984.

[40] S. Srivastava and A. Singh. *Facebook Application Development With Graph API Cookbook*. Packt Pub Limited, 2011.

[41] Y. Wang, D. Burgener, M. Flores, A. Kuzmanovic, and C. Huang. Towards Street-Level Client-Independent IP Geolocation. In *USENIX Symposium on Networked Systems Design and Implementation*, 2011.

Vita

Sashmit Bhaduri attended the Georgia Institute of Technology and graduated with a Bachelors of Science in Computer Science in 2006. He has since worked as a Software Engineer for the Applied Research Laboratories at the University of Austin, a Department of Defense-affliated research center, on a variety of DoD and private industry contracts. In 2010, he entered the Graduate School at the University of Texas at Austin and enrolled in the Software Engineering Program. He has long enjoyed building software, learning new programming languages, and keeping abreast of developing technologies.

Permanent address: sashmit@gmail.com
9009 Great Hills Trail Apt. 1013
Austin, Texas 78759

This report was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.