October

Read the news you want to read

Authors:	Tom Dooner, Mika Little, Brian Stack
Project:	Project October
Date:	April 20, 2013

Contents

List of Figures

1 Abstract

Longtime users of modern news aggregation services (such as Reddit, Slashdot, Digg, and Hacker News) report noticing a marked decrease in quality of discourse as the services gain mainstream attention. This gradual, irreversible decline was noted as early as the newsgroup era, when new college students would log in for the first time in September, causing an influx of new users and subsequently diluting discussion upon the sites which they joined. The situation worsened after AOL opened newsgroups to the masses, causing the community standards to continue to devolve, according to newsgroup veterans[1]. This gradual decline of content quality due to large numbers of new users came to be known as "Eternal September", being "Eternal" as the influx of new users was no longer restricted to September, but persisted through all months of the year. This phenomenon indicates that balancing a large community with high quality content is a challenge.

October is a solution to this problem. By providing users with an automatically-customized experience, we believe we can provide a large community thoughtful discourse and interesting articles while avoiding the effects brought on by Eternal September. October will employ a technique to recommend news articles which, to our knowledge, is not currently applied on any other major news aggregators. With custom recommendations for each user, we believe that the October community can attain large size without a sacrifice in quality.

2 Project October: Recommender System

Due to the ever-growing amount of information available online, the need for a highly developed personalization and filtering system is growing significantly. Recommender systems constitute a specific type of information filtering that attempt to present items according the interests expressed by a user. Most web recommenders are employed for e-commerce applications or customer adapted websites, which assist users in decision making by providing personalized information, but the same techniques that suggest related items on e-commerce websites can recommend news articles to users as well. We believe Project October is the first attempt to apply recommendation techniques to social news aggregation.

2.1 Background

It is our hypothesis that the recommendation of news sources will provide a scalable community experience that can be tailored to each person's interests. Providing an automatic, customized, recommendation of articles will prevent the community from being diluted by new users and thus prevent the Eternal September phenomenon. Each user will have their own viewing environment curated for them, with different mindsets and preferences forming their own communities in which like-minded individuals can partake in discussion, eliminating the cross-contamination of user bases.

2.2 Intended Audience

Project October will be open to the public. At launch, we will be inviting our friends from the CWRU community to try it out first, so we expect the user base to be somewhat technical in nature.

2.3 Progress Since Progress Report 2

Since Progress Report 2, we have implemented all our design requirements, released a beta version to the CWRU EECS department, and collected user metrics. More details about the v1.0 minimum viable product can be found in Section 5.1.

For the sake of releasing a minimum viable product within the semester, we have removed some complexity from our requirements in Progress Report 2. Specifically, we have made the following changes to the report, or the scope of our project:

- We have specified the plan for the backend recommendation engine in tangible terms.
- We have removed the ability to upvote, downvote, and tag comments. Those abilities will be easy to add later but are not core to the application, especially until October gains critical mass and more people comment.
- We have added an RSS Subscription feature, specified in Section 3.7 and described in Section 4.1.1.
- We have made numerous copy tweaks to the report, such as
  - Reordering the report to list the requirements before the details of the project's implementation
  - Adding Sections 4.1.1-4.1.6 about the frontend and Section 7 about the overall final status.
  - Moving the User Interface section into the requirements.
  - Describing our testing process in Section 6

2.4 This Report

3 Software Design Requirements

3.1 Homepage

The are two cases for the homepage: logged in users and logged out users.

Logged out users arrive at the homepage and are presented with a splash page containing a description of October's features. There is a registration link so that interested users can register for their own account.

Logged in users are, upon landing on the homepage, presented with a display of news articles. Articles are displayed with pictures (when available) along with headlines. Each article is accompanied with links to positively or negatively influence the article's recommendation rank, and also to comment on the article. The source of the article is also provided, as is the time that it was posted.

The homepage is an especially crucial interface to get right. October should mimic a newspaper feel with articles staggered in a rough column grid. This will allow readers to be presented with many recommendations of different weights simultaneously in a manner in which they are accustomed to. Articles will have two links (upvote and downvote) into the article area, along with the link to that article's comments.

The search bar, along with a 'Post An Article' button, a link to the user's profile page, and a Logout link will be located at the top of the site.

3.2 User Creation, Login, and Display

New users are able to create October accounts with but three pieces of information – their desired username, their email, and a password. After creating an account, an email is sent to the user welcoming them to October.

When a user first logs into October, the backend recommender won't know anything about their interests. Thus, users should be shown various sample topics to browse through. They are also able to (via their account page) manually type in relevant keywords to aid the recommender in providing them with articles.

Every mention of a user's name (i.e. the credit for who posted an article or comment) is a link to a user profile page. More detail about the user profile page is given in Section 3.3.

3.3 User Profile Page

Each user has a profile page containing information related to that user. The profile page contains a list of comments that the user has made on posts and a list of articles the user has posted. All information on this page is public, as it is merely a summary of content available elsewhere on the public website.

3.4 User Profile Edit Page

In contrast to the profile page, which is a summary of a user's public activity, the profile edit page allows a logged in user to edit their own information. Settings such as their email address and password can be changed here.

The other significant part of the user profile edit page is a section which shows the user their own recommendation settings. This shows up in the format of a list of keywords that the backend has ascribed to that user. The user can remove keywords that they do not wish to be associated with.

Furthermore, there should be a field where a user can submit additional keywords they wish to see more news from. The user will be associated with the keyword when they submit the form.

3.5 Article Posting

Logged in users are able to click a link on the homepage to submit a story. The submit story page will ask for the link to the desired story, and automatically show the results of scraping that story's contents (see Section 4.2 for more about scraping). If the post looks good to the user, he or she can click a "Post!" button, which will persist the news story in all appropriate databases.

3.6 Article Display Page

The article display page is a page dedicated to a posted news article. Every news article has an associated article page. The intent of the article page is to facilitate sharing of opinions around the posted news story through comments.

A user arriving on an article's page arrives at a page which contains the title of the news article along with a threaded comment list. Users may leave comments as children of other comments, and thus may engage in a lively discussion.

3.7 RSS Subscription

Users should be able to subscribe to RSS feeds by entering the URL to a feed in their User Profile Edit page. This will allow users to customize the news sources present in October without painstakingly submitting things until the community grows larger.

Subscribing to an RSS feed will automatically post the articles from the feed to October in a timely manner. Users should be able to click the feed name on an article displayed and be taken to a list of posts from that RSS feed (much in the same way that users can see a list of another user's posts on the user profile page).

3.8 Frontend $\leftrightarrow$ Backend API

Since the frontend and backend are divided with a Service Oriented Architecture (with Apache Thrift bridging the gap[1]), there must be an API between them.

The API should be as simple as possible to maintain the functionality of October. This is not a public API, so it needs to be robust to volume but we do not need to implement API endpoints for the functionality mentioned elsewhere in the requirements.

The API is defined in a Thrift definitions file in the project-october-api repository on GitHub[9].

3.9 Backend

THIS NEEDS TO BE MADE

4 Application Architecture

4.1 Frontend

The frontend is similar in function to a standard social news aggregator. Upon going to the homepage, new and logged-out users are presented with a splash page that describes the nature of Project October and are given a registration link. Users can register for the application by selecting a username and password. A registration confirmation is emailed to newly registered users.

[1]See Section 4.4 for more about Thrift

When logged in, users are presented with their personalized content. The design is modelled after the front page of a newspaper – articles that are evaluated to be more interesting to a user are placed in more prominent positions while articles that are deemed less relevant to the user's interest will fill the side columns, be located further down the page, and occupy less space. Users are able to click on the headline (or associated image, if existent) to be taken to the original news source.

Each news article features a link to view the associated comments as well as icons which allow the user to plus and minus the article. These are analogous to upvoting in Reddit, except that plusing or minusing an article on October will inform the recommendation engine of your individual preference, rather than directly impacting the weighting of an article by a predefined formula. Therefore, voting serves two purposes: to inform other users of the quality of the article and to inform the recommender system of the user's interests.

Users will be able to click a link on the homepage which takes them to an article submission form. The submission form requests a URL and a headline for that news story. Upon entering the URL, the news story is scraped in the background using a web scraping tool which attempts to extract properties about the page automatically – relying upon the ¡meta¿ tags and heuristics to determine the article's body text. The user is also presented with a list of tags that are garnered from the article, and they can delete or add their own tags to better represent the content of the article. A list of scraped images is also presented, and the user may select which one (if any) they want to head the article with.

4.1.1 RSS Subscription

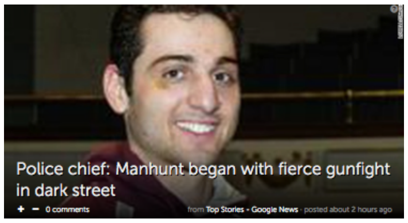
In order to implement the RSS Subscription feature (described in Section 3.7), we created a background task that scrapes each user-submitted RSS feed and posts each RSS entry as a new story. Since this means there are two types of entities that can post articles, and we wish to share logic between both, we use ActiveRecord's Single Table Inheritance[10] to make Users and Feeds share attributes in a single table **posters** (see the database schema in Figure 5). The most important reason for this is because Users and Feeds must share the keyspace for their primary key (their `id` columns) so the backend can treat them the same when dealing with following relationships (a user should be able to follow both a user and a feed, under the same logic).

This will also allow other types of posting entities in the future if we decide to add additional integrations.

4.1.2 Homepage Article Display Sizes

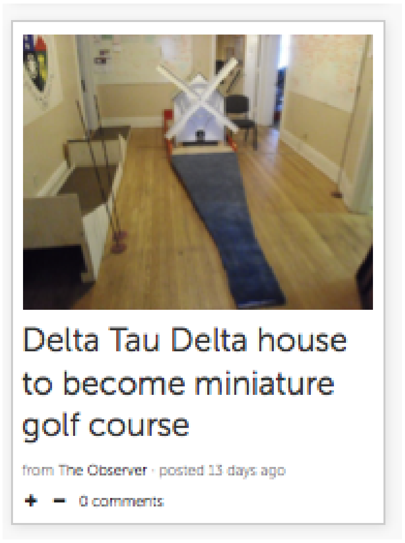
According to the specification (in Section 3.1), the homepage should display articles in a visually appealing fashion. Towards this end, we split the homepage into three equal-width columns and display each article with one of the following types:

- `featured` type articles take up two columns and contain a large image from the post. The headline is large and stands out. We intend these featured articles to be the eye-catching cornerstone(s) of the homepage.
- `primary` type articles are displayed in a white box to emphasize them from the background. The headline is large and flows with the picture to create the newspaper feel.
- `secondary` type articles are displayed with a grey headline and grey link text, to blend in with the background and offer the reader additional options if they desire to look harder to find them.


Police chief: Manhunt began with fierce gunfight in dark street

+ − 0 comments from Top Stories - Google News · posted about 2 hours ago

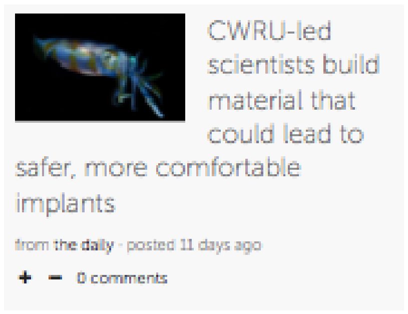
`featured` article (scaled down to 50%)


Delta Tau Delta house to become miniature golf course

from The Observer · posted 13 days ago

+ − 0 comments

`primary` article


CWRU-led scientists build material that could lead to safer, more comfortable implants

from the daily · posted 11 days ago

+ − 0 comments

`secondary` article

`primary` and `secondary` articles may take up any amount of vertical space, but are constrained to one column in width. If the homepage is not wide enough to fit three columns (on clients of less than 768px), the homepage goes into single-column mode where all articles are displayed linearly.

4.1.3 Homepage Article Layout Algorithm

The article layout algorithm takes place in Ruby (the backend of the frontend) and determines the size and approximate position of the articles to be displayed to the user. The algorithm is complex enough to make intelligent decisions, yet simple enough that it can be performed quickly and repeatedly. The algorithm is, given a list of n article IDs and weights for each article from the recommender, take the following steps:

1. Sort the article list by weight, so the highest-weighted article is first.
2. Take the top two highest-weighted articles with pictures. Assign them the type `featured`.
3. Take the remaining $round(\frac{n}{2}) - 2$ items and assign them the type `primary`.
4. Assign the remaining items the type `secondary`.
5. Introduce some randomness by naïvely grouping all posts into four categories by ID.
6. Insert the `featured` articles into the first and the $\frac{n}{4}$th indexes of the list.
7. Return the list of $(item, type)$ tuple to the frontend.

The frontend will output HTML for each item and type in the order returned from the algorithm. The CSS for the homepage contains basic styling instructions, but not enough to create the desired homepage effect. For that, the user's browser will invoke the JavaScript masonry.js library[11]. Masonry.js takes a set of arbitrarily-sized DOM nodes and displays them in a compact way, thus giving us the desired newspaper front-page effect.

Using custom responsive CSS and JavaScript event handlers on window resize, we ensure that the newspaper feel is consistent and usable on any size screen and when the screen is resized after the page originally loads.

4.1.4 Backend Proxy

Since the connection to the backend is via a web service, the frontend cannot always assume a working connection to the backend. Towards this end, we have written a Ruby proxy around Thrift which keeps the frontend function even in case of problems with the backend. It will try again to establish the connection in case of momentary connection break, and will switch to a mock backend if the real backend still does not reply. The mock backend is API-compatible with the real backend, but is not capable of making informed decisions so we display a message to October users that October is operating at diminished capacity ("amnesia mode").

4.1.5 Sending Email to Users

October uses Ruby on Rails's packaged ActionMailer to send emails. We use the premailer Ruby gem to speed up development – it allows for the use of real stylesheets which are then compiled inline to the appropriate HTML elements of the emails before they are sent.

Currently, the only email that is delived is a welcome email for the user offering basic onboarding instructions. However, in the future, we intend to engage users via email with customized recommendation emails (i.e. "We just found this article that we know you will like!")

4.1.6 User Tracking + Analysis

In order to understand how effective our users find October, we have instrumented the site with a heavy dosage of analytics. To track basic page views and real time events, we have installed Google Analytics. But, Google Analytics does not provide the granularity that we would need to adequately understand user behavior. To this end, we have instrumented the site with Mixpanel[8], which specializes in tracking user-triggered *events* rather than simple pageviews.

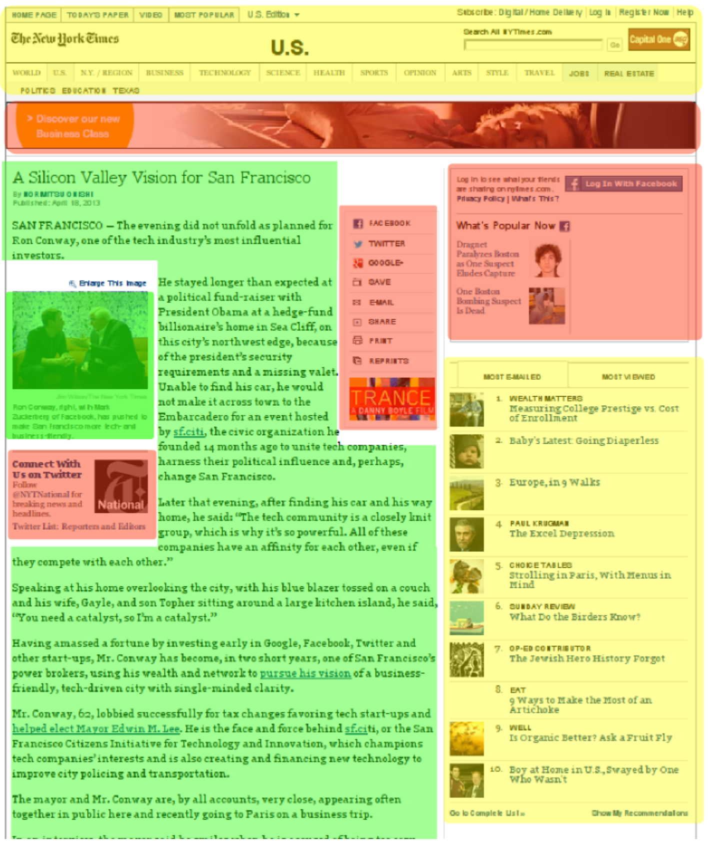
Mixpanel helps us track user behavior by tracking an event every time a user performs one of the following actions: log in, sign up, receive recommendations, vote, click article headline, or post a new article.

We currently use Mixpanel is a couple ways beyond simply tracking user events, however. Primarily, we track how many milliseconds the user waits to receive recommendations from the backend. We also track the direction of users' votes, so we can tell how satisfied users are, at a high level, with their article recommendations.

4.2 Web Scraping

We use a library called Pismo [4] to scrape webpages. While this may seem like a trivial operation that we should implement ourselves, there is much that must be done in order to determine the actual content of a webpage. We have selected to use the "cluster" approach Pismo provides to determine which parts of the webpage are actually content. This approach ranks each <div> element on a page according to the following heuristics:

1. element must contain text
2. earlier elements are more likely to be in the body of text
3. each element must be over some minimum length (80 characters by default)
4. more text in an element and more "." delimited sentences raise chance of being content



A Silicon Valley Vision for San Francisco

By NORIMITSU ONISHI
Published: April 18, 2013

SAN FRANCISCO — The evening did not unfold as planned for Ron Conway, one of the tech industry's most influential investors.

Jim Wilson/The New York Times
Ron Conway, right, with Mark Zuckerberg of Facebook, has pushed to make San Francisco more tech- and business-friendly.

He stayed longer than expected at a political fund-raiser with President Obama at a hedge-fund billionaire's home in Sea Cliff, on this city's northwest edge, because of the president's security requirements and a missing valet. Unable to find his car, he would not make it across town to the Embarcadero for an event hosted by sf.citi, the civic organization he founded 14 months ago to unite tech companies, harness their political influence and, perhaps, change San Francisco.

Connect With Us on Twitter
Follow @NYTNational for breaking news and headlines.
Twitter List: Reporters and Editors

Later that evening, after finding his car and his way home, he said: "The tech community is a closely knit group, which is why it's so powerful. All of these companies have an affinity for each other, even if they compete with each other."

Speaking at his home overlooking the city, with his blue blazer tossed on a couch and his wife, Gayle, and son Topher sitting around a large kitchen island, he said, "You need a catalyst, so I'm a catalyst."

Having amassed a fortune by investing early in Google, Facebook, Twitter and other start-ups, Mr. Conway has become, in two short years, one of San Francisco's power brokers, using his wealth and network to pursue his vision of a business-friendly, tech-driven city with single-minded clarity.

Mr. Conway, 62, lobbied successfully for tax changes favoring tech start-ups and helped elect Mayor Edwin M. Lee. He is the face and force behind sf.citi, or the San Francisco Citizens Initiative for Technology and Innovation, which champions tech companies' interests and is also creating and financing new technology to improve city policing and transportation.

The mayor and Mr. Conway are, by all accounts, very close, appearing often together in public here and recently going to Paris on a business trip.

Log In to see what your friends are sharing on nytimes.com. Privacy Policy | What's This?

Dragnet Paralyzes Boston as One Suspect Eludes Capture

One Boston Bombing Suspect Is Dead

1. WEALTH MATTERS
Measuring College Prestige vs. Cost of Enrollment
2. Baby's Latest: Going Diaperless
3. Europe, in 9 Walks
4. PAUL KRUGMAN
The Excel Depression
5. CHOICE TABLES
Strolling in Paris, With Menus in Mind
6. SUNDAY REVIEW
What Do the Birders Know?
7. OP-ED CONTRIBUTOR
The Jewish Hero History Forgot
8. EAT
9 Ways to Make the Most of an Artichoke
9. WELL
Is Organic Better? Ask a Fruit Fly
10. Boy at Home in U.S., Swayed by One Who Wasn't

Figure 1: Sections of webpage colored by desirability.

Green Sections are best for determining content of article. Also includes potential images to use as a headline image for the article.

Yellow Sections may contain useful information but are not part of the body of the article.

Red Sections do not contain useful information and will harm results if included.

This works well for our purposes. It is possible that as time progresses, we can make use of HTML5 semantic tags such as <article> in order to more accurately extract text.

Once this text is extracted, Pismo relies upon the Phrasie [7] library to determine keyword tokens. The initial version of Pismo we used was based upon a naive list of (token, freqency) pairs, however Phrasie uses parts-of-speech to determine keywords and gives them a weight based on estimated importance rather than a straight frequency. The strength of this approach is that it results in a much higher signal-to-noise ratio than the naive approach. A weakness is that we've now reduced our possible languages down to just english. As time goes on we can modify this approach to either support more languages or just work in a general sense on any language.

We also make use of Pismo's ability to extract images on the page that have a high likelihood of being content rather than UI elements.

Firgure 1 gives a visual representation of the content on a page we want to scrape.

4.3 Backend

The backend recommender service is a Scala [6] application that uses MongoDB [5] as its persistent data store. The frontend communicates with it using the API that we defined using thrift. When a post is submitted – either through a user submission or via a feed – the results are sent over. When a user loads the homepage or searches for a string, the backend is invoked to service the request.

4.3.1 Vectorization of Posts

The scraper from section 4.2 presents a list of (token, weight) pairs to the backend through the API. This is stored in the database as a document where the key is the document id from the frontend. In addition, each token in the document is incremented by the weight of the document so that we can calculate the document frequency when it is needed. The id of this document is also added to a list on the token so that we can find this post later when we need it.

4.3.2 Vectorization of Users

Users are also a vector in the key-value database. The user vector is updated in the same way as a document vector except that the document frequency of each term is not incremented. A user's vector is updated whenever the user performs some sort of action on a document. These actions are defined in the API, but include actions such as voting up or down, reading, or commenting on a post. Now that each user is a vector in the database and each post is a document in the database, we can operate on each one.

4.3.3 Inverted Index

Now that we have all of the terms that a user will be potentially interested in, we must find the documents in the database that are interesting to the user. We can do this easily by merging the lists of posts that each token contains for documents that contain it. This is known as an inverted index.

4.3.4 Tf-Idf

We use a term-frequency/inverse-document-frequency approach to scaling the vectors in preparation to return our ranked results. We apply two scaling factors to each of our terms in our term vectors. The first is term-frequency(TF). This scales the raw term up by the number of times it is present in a document. So if an article mentions CWRU 15 times, it is more likely to be about CWRU than a document that mentions it 0 or even 1 times. Next we scale it by inverse-document-frequency(IDF). This means that terms that are common in all documents are weighted less. So for example the term "it" is common to just about every document we record and therefore is not a very valid signal for us to recommend on.

4.3.5 Dot Product

Now that we've scaled both the user and document vectors, we can perform a dot-product on both vectors such that each term in the user vector that is present in the document vector is multiplied and summed with each of the other similar terms. As the reader will most likely know, vectors that are pointing in similar directions will have higher dot-products than vectors that are divergent. We use this fact to rank the posts in order of most likely to be interesting to the user to least and return this ranking to the frontend after passing it through one last filter.

4.3.6 Time Scaling

We also want to only return to the user newer results so that there is a constant "churn" of articles on the frontpage. We first limit the results returned from the database to the number of results that the frontend requested. Next we apply a scaling function to all results so that newer results are favored over old ones. The function is as follows

$$weight(x, dt) = \begin{cases} x & : dt < T \\ x/\ln(dt + T) & : dt \geq T \end{cases} \tag{1}$$

Where dt is the time from the time from the time the post was submitted until the recommendation was requested, x is the initial weight supplied by the Tf-Idf and dot-product, and T is a time constant we decide upon. In our current implementation, T is set at 10 hours so that posts can stay on the frontpage unhindered for that amount of time. Once this scaling is applied to each of the posts we're returning, we can send them back to the frontend using the API.

4.3.7 Searching

A side effect of storing the documents in that manner is that we can easily implement full-text-search by the frontend submitting a vector of a search term and the backend treating it just like a user is treated in the above explanation.

4.4 Frontend/Backend API

October will employ an API to promote separation between the frontend and the backend recommender system. This will provide a clear interface and facilitate easy simultaneous development of both parts of October.

To implement the API, we will use Apache Thrift, an Interface Description Language[2]. The essence of the API is simple, featuring primarily two types of calls:

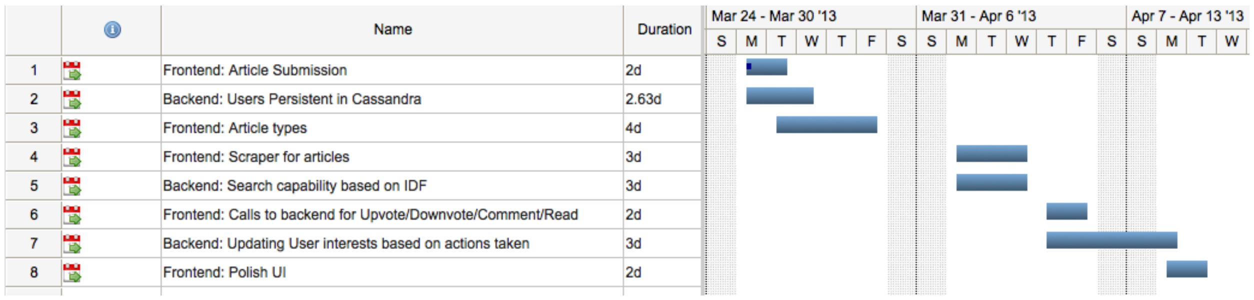
Give n **recommendations for user** u This is the main output from the backend, returning recommended news stories or comments for a given user. Ancillary parameters will be added to this to facilitate the frontend placement of articles, e.g. the recommendation confidence and individual article weightings.

User u **took action** n This is the main input to the backend, allowing it to adjust recommendations according to user action. The parameters to this API call can be of many types. For example, "User commented on article $\#n$", "User plused or minused comment $\#n$", and "User visited link $\#n$" are all valid parameters for this API call.

These two calls manifest themselves in Thrift with a few simple object definitions and an interface. The abbreviated version `0.1.0` is attached as Appendix A.

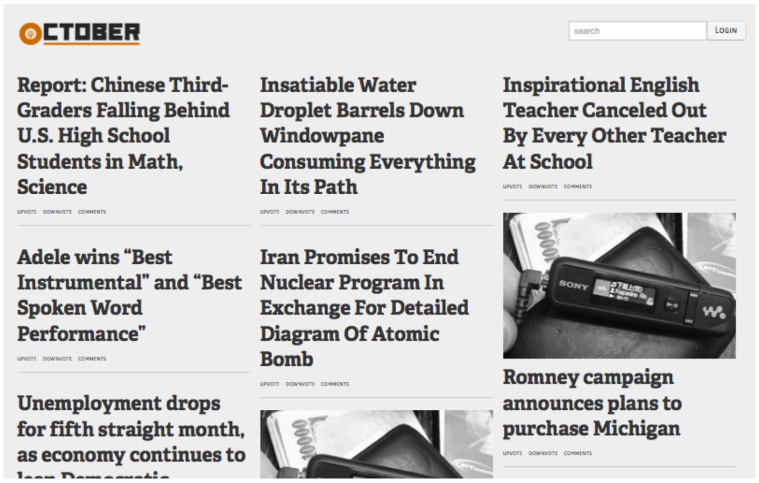
5 Project Management–Administrative Details

We have split the team into two parts, frontend and backend. With respect to the frontend, Tom and Mika have worked on the development and design respectively. The biggest difficulty here is to determine how to represent the news in such a way that the user will be inclined to continue using the service. As for the backend, Brian has been tasked with creating both the service architecture and the recommender engine. This includes the implementation of our backend's TF-IDF algorithm and the Scala connector to the Thrift API and MongoDB [5] that computes recommendations.



		Name	Duration
1		Frontend: Article Submission	2d
2		Backend: Users Persistent in Cassandra	2.63d
3		Frontend: Article types	4d
4		Frontend: Scraper for articles	3d
5		Backend: Search capability based on IDF	3d
6		Frontend: Calls to backend for Upvote/Downvote/Comment/Read	2d
7		Backend: Updating User interests based on actions taken	3d
8		Frontend: Polish UI	2d

Figure 2: Project October Gantt Chart



Report: Chinese Third-Graders Falling Behind U.S. High School Students in Math, Science

Insatiable Water Droplet Barrels Down Windowpane Consuming Everything In Its Path

Inspirational English Teacher Canceled Out By Every Other Teacher At School

Adele wins "Best Instrumental" and "Best Spoken Word Performance"

Iran Promises To End Nuclear Program In Exchange For Detailed Diagram Of Atomic Bomb

Romney campaign announces plans to purchase Michigan

Unemployment drops for fifth straight month, as economy continues to

Figure 3: Project October Homepage, pre-Alpha (Headlines courtesy The Onion)

5.1 Release v1.0

In Project Report 2, we promised a fully-functional beta by "early to mid-April". On April 15th, we released the beta to the CWRU community, and it has been met with a warm reception.

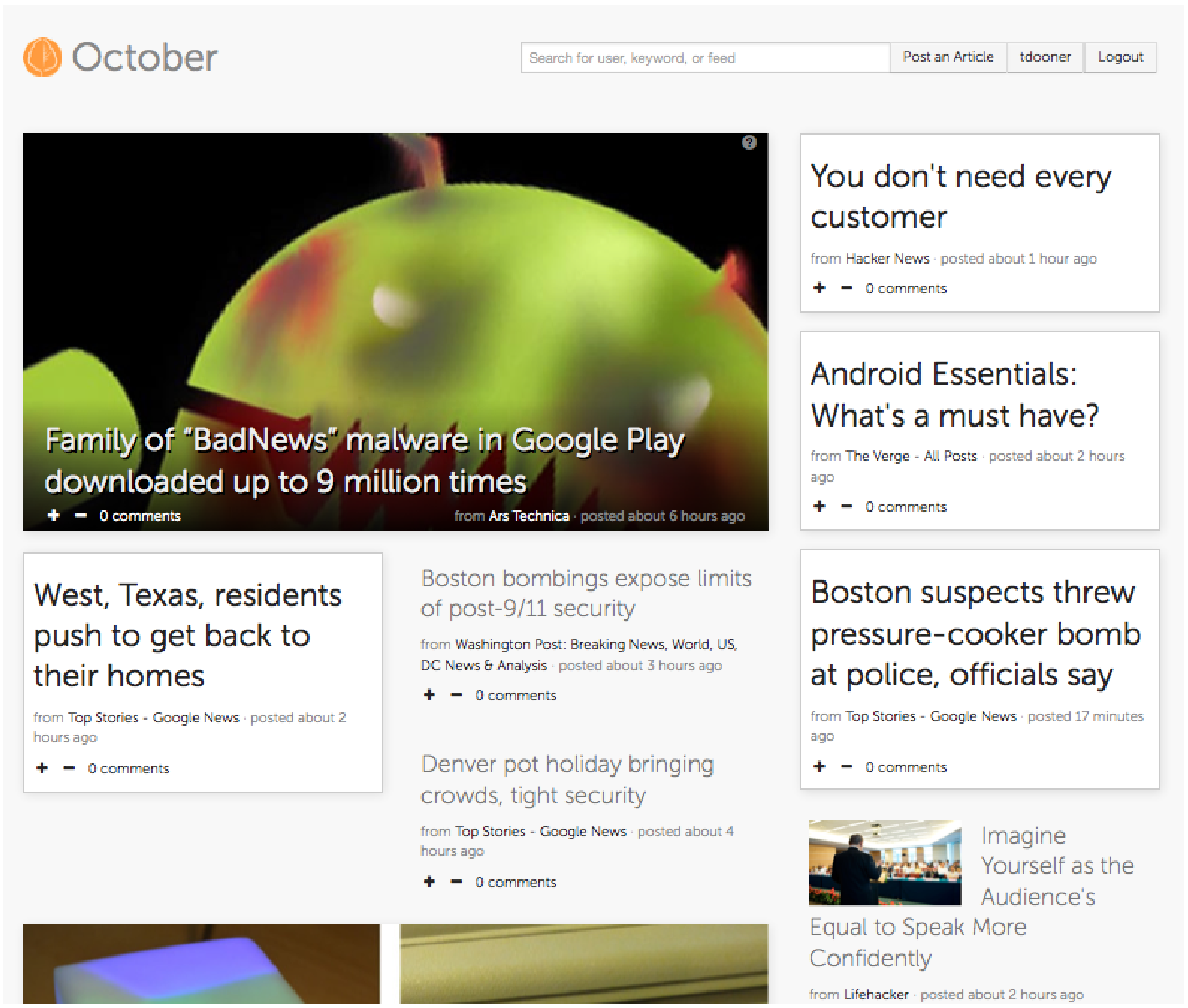
The user interface for other pages will be in the same general content area, however there are no further screenshots to share at this time.

6 Testing & Evaluation

Testing and evaluation will be performed as we progress through the project. Currently, only very minimal testing is implemented.

7 Project Progress

The project is progressing satisfactorily. We are attempting to follow Agile Design Patterns, and we have been effective at meeting for short periods frequently. On the frontend, we have completed the user creation + login processes, mocked up the homepage, and connected the Thrift library. On the backend, we have implemented the API so it responds to requests from the frontend. The recommendation engine is still in-progress. We remain committed to maintaining team communication via scrum and continue working.



October

Search for user, keyword, or feed

Post an Article | tdooner | Logout

Family of "BadNews" malware in Google Play downloaded up to 9 million times

+ − 0 comments from Ars Technica · posted about 6 hours ago

You don't need every customer

from Hacker News · posted about 1 hour ago

+ − 0 comments

Android Essentials: What's a must have?

from The Verge - All Posts · posted about 2 hours ago

+ − 0 comments

West, Texas, residents push to get back to their homes

from Top Stories - Google News · posted about 2 hours ago

+ − 0 comments

Boston bombings expose limits of post-9/11 security

from Washington Post: Breaking News, World, US, DC News & Analysis · posted about 3 hours ago

+ − 0 comments

Boston suspects threw pressure-cooker bomb at police, officials say

from Top Stories - Google News · posted 17 minutes ago

+ − 0 comments

Denver pot holiday bringing crowds, tight security

from Top Stories - Google News · posted about 4 hours ago

+ − 0 comments

Imagine Yourself as the Audience's Equal to Speak More Confidently

from Lifehacker · posted about 2 hours ago

Figure 4: Project October Homepage, version 1.0

7.1 Discussions & Conclusions

Overall, the project is still on schedule. We still intend to complete a fully-featured product on-time.

8 Lessons Learned

Throughout the development of October, we gained insight as how to properly design and create a product that will be used by many. Foremost among lessons learned, is that agile development ended up not being suitable for our work agenda. It required a certain rigidness in our schedules that we could not maintain. Instead, periodic scrum meetings sufficed for communicating progress and sharing ideas.

Alongside the ineffectiveness of agile development in a college setting, an important lesson was learned when October was first released to the public in early April. Based on user feedback, we learned that October was, in fact, a viable web service, and we have had moderate user retention. We have enjoyed working on this, and we hope to continue refining it after the class is over. We will be keeping in contact with users to build upon October, enhancing the service and hopefully increasing user retention.

8.1 Advantages & Disadvantages of a Service Oriented Architecture

9 Appendix

References

[1] http://en.wikipedia.org/wiki/Eternal_September

[2] http://en.wikipedia.org/wiki/Apache_Thrift

[3] http://en.wikipedia.org/wiki/Tf-idf

[4] https://github.com/peterc/pismo

[5] http://www.mongodb.org/

[6] http://www.scala-lang.org/

[7] https://github.com/ashleyw/phrasie

[8] http://mixpanel.com

[9] https://github.com/bis12/project-october-api

[10] http://api.rubyonrails.org/classes/ActiveRecord/Base.html#label-Single+table+inheritance

[11] http://masonry.desandro.com/

A Thrift API Definition

```
const string VERSION = "1.0.0"
struct Post {
    1: required i64 post_id,
    2: optional double weight,
}
struct PostList {
    1: optional double confidence,
    2: required list<Post> posts,
}
struct User {
    1: required i64 user_id,
}
struct Token {
    1: required string t,
    2: required i32 f,
}
enum Action {
    READ,
    VOTE_UP,
    VOTE_DOWN,
    VOTE_UP_NEGATE,
    VOTE_DOWN_NEGATE,
    POST,
    COMMENT,
    REPORT,
    TAG,
    FOLLOW,
}
service Recommender {
    string ping() throws (1: TimeoutException te),
    PostList recPosts(1: required i64 user_id, 2: required i32 limit, 3: required i32 skip)
      throws (1: NotFoundException nfe, 2: EngineException ee, 3: TimeoutException te),
    bool addUser(1: required i64 user_id)
      throws (1: EngineException ee, 2: TimeoutException te),
    bool addPost(1: required i64 user_id, 2: required i64 post_id, 3: required list<Token> raw_fr
      throws (1: EngineException ee, 2: TimeoutException te, 3: NotFoundException nfe),
    bool userToPost(1: required i64 user_id, 2: required Action verb, 3: required i64 post_id)
      throws (1: NotFoundException nfe),
    bool userToComment(1: required i64 user_id, 2: required Action verb, 3: required i64 comment_
      throws (1: NotFoundException nfe),
    bool userToUser(1: required i64 actioner_id, 2: required Action verb, 3: required i64 actione
      throws (1: NotFoundException nfe),
    map<string, i64> userTopTerms(1: required i64 user_id, 2: required i32 limit)
      throws (1: NotFoundException nfe),
    map<i64, double> textSearch(1: required list<string> tokens, 2: required i32 limit, 3: requir
      throws (1: EngineException ee),
    bool addUserTerms(1: required i64 user_id, 2: required list<string> terms)
      throws (1: NotFoundException nfe),
    bool removeUserTerms(1: required i64 user_id, 2: required list<string> terms)
      throws (1: NotFoundException nfe),
}
```

B Database Design

B.1 Frontend Database

Please see Figure 5.

B.2 Backend Database

MongoDB [5] is a schemaless database and as such there is no schema to show here. However, the backend uses "models" to access the data and as such there is a pseudo-schema which we can present here. The notation is simply cleaned up Scala without the actual logic attached.

```
class User(
    id: Long,
    tokens: Map[String, Long],
    friends: Seq[Long])

class Token(
    id: String,
    df: Long,
    posts: Seq[Long])

class Post(
    id: Long,
    posted: DateTime,
    tokens: Map[String, Long])
```

C User Manual

D Programmers Manual

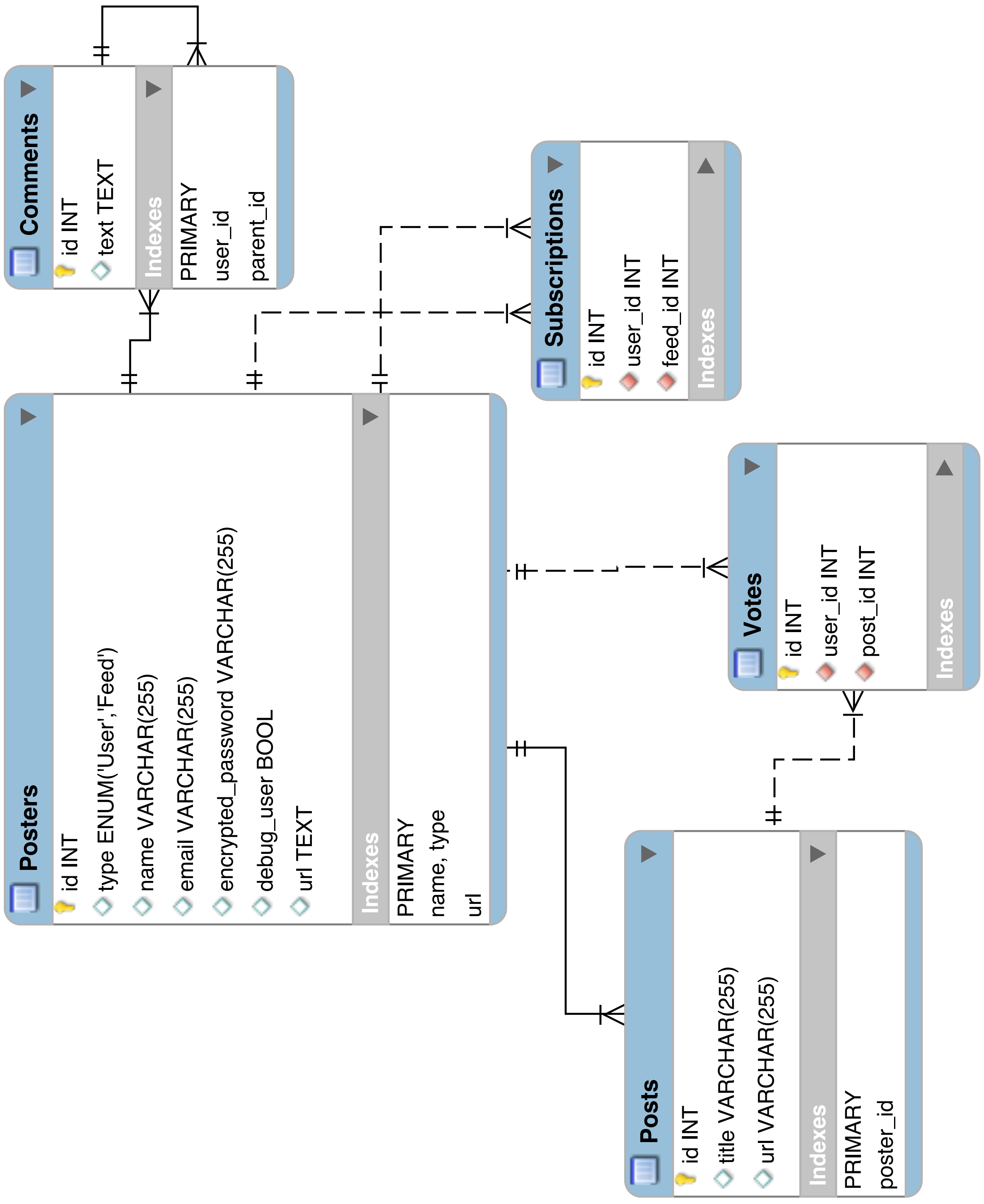


Figure 5: Frontend Database Relational Diagram