Rush Tracker: Design Document

By Random Engineers (random-engineers@mit.edu)

Ben Shaibu
Dennis Smiley
Matt Kerr
Louis Descioli

Overview

Purpose and Goals:

We're creating a web application called Rush Tracker that is an information storage, management, and presentation solution for keeping track of rushees during fraternity rush.

The current method for managing rushee information during rush is slow and inefficient because it involves using two decoupled views of the same data. The first view is the data management/storage view, which is generally in the form of a Spreadsheet within Google Drive, and is used to keep track of vital information for potential rushees. The second view is the rush-meeting presentation view, which is generally in the form of a Slideshow that is manually created and curated from key rushee information contained in the Google Drive Spreadsheet, and is used during meetings to drive recruitment discussions about each rushee.

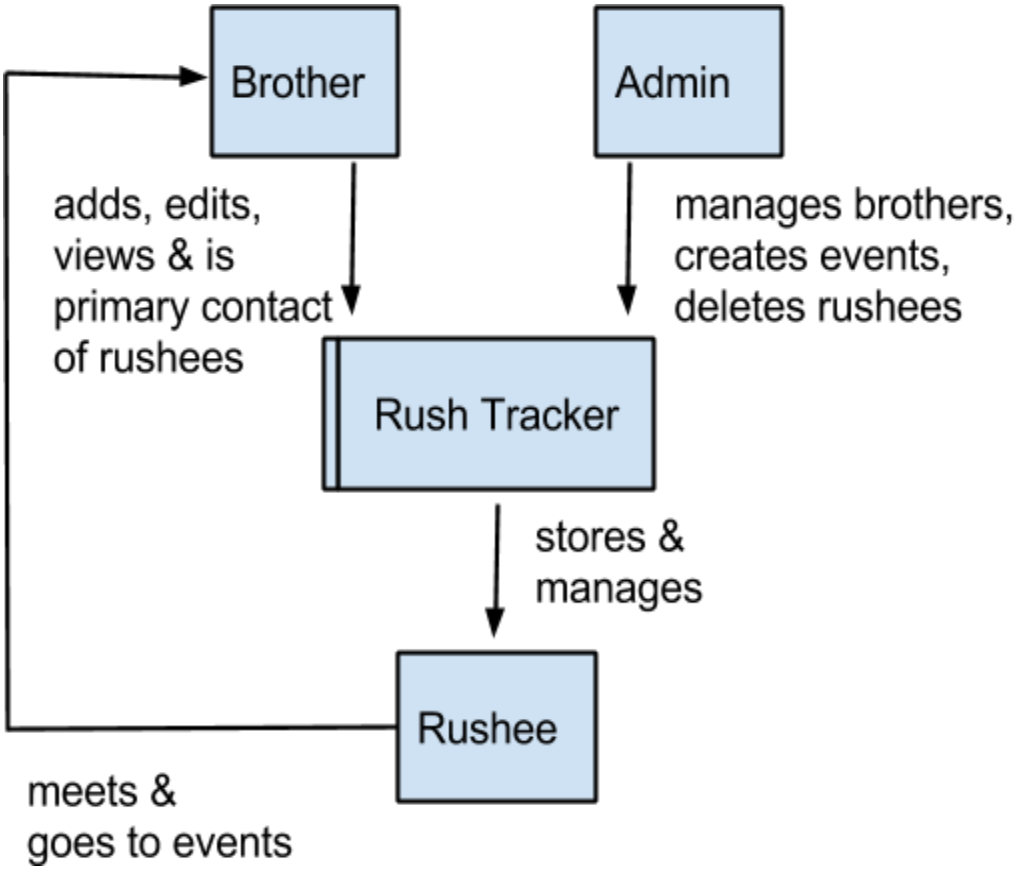
Rush Tracker is designed to be a centralized rushee information storage solution with which 1) the rush-meeting presentation view is generated automatically from the data management view, and 2) the brothers of a fraternity can easily access and edit rushee data from the data management view. Our goal is to create an application that allows brothers to store information on rushees, and that helps with rush-meetings by rendering that information in a rush-meeting presentation view.

Currently there are a variety of Rush Management applications on the market, but none of them has attained widespread appeal due to missing features and a lack of visual appeal. They have a very corporate feel that is unattractive to College Students. RushTracker offers a friendlier interface along with a unique Presentation Mode that automatically showcases Rushee information in a format perfect for Rush Meetings. Currently, users of the alternatives (such as RushTap and ChapterPlan) still have to manually create such presentations using other software, such as PowerPoint.

Disclaimers of Scope and Goals:

The first iteration of the application is designed to be used by a single fraternity. We have designed it where it will be easily possible to extend the application to support multiple fraternities. This is shown in the data model diagram. Further, our application tracks events, but only for the purpose of seeing which rushees attended them; providing event management and planning tools is **not** part of our goals.

Context Diagram:

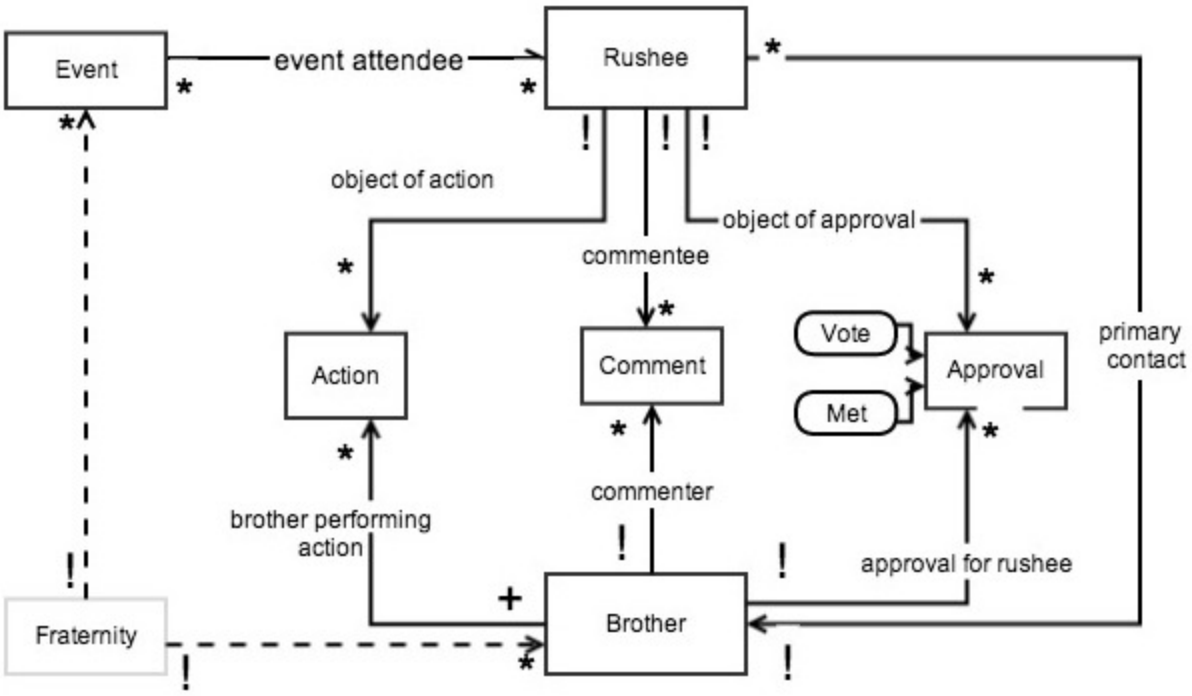


Concepts

Key Concepts:

- Rushee
    - A Rushee is an individual who is a potential Brother of a Fraternity. As such, besides their basic information (Name, Age, Potential Major, Hometown, Contact Information, etc.), it is also important to keep track of which Events a Rushee has attended and their current rushee status.
    - Rushees have an Action Status, based on the Fraternity's plan of action for recruiting the Rushee. An Action Status is either 'Stay the Course', 'Push Harder', or 'Repudiate'.
    - Rushee's also have a Bid Status. A Bid Status is either 'None', 'Offered', 'Accepted', or 'Rejected'.
- Brother
    - A user of the system and a member of the Fraternity. Brothers can also serve as a primary contact for a Rushee. Most of the contact with a Rushee is usually through their primary contact.
- Event
    - An event is a representation of an activity the Fraternity holds in order to recruit Rushees. If a Rushee attends an activity, they are added to the attendees of the respective Event.
- Action
    - An Action is a task or event that happens between one or more Brothers and a Rushee. It is something such as 'Invite to Go Kart Racing', or 'Take out to Lunch'. These Actions are influenced by a Rushee's status and comments regarding the Rushee from Brothers. Actions are usually created during Rush meetings in order to recruit Rushees more effectively.

Data Model:



The Fraternity entity is included to show how the system would be extended to support multiple fraternities using the system. It will not be included in the MVP.

Behaviour

Features:

- Rushee Profiles
    - All rushees are given profiles that contain their basic information and any additional information a Brother feels is important.
    - The profiles are integrated with a Rushee's social media accounts to display photos
    - Track which brothers have met which Rushees
    - Keep track of Rushee Action Status and Bid Status (described more in the Rushee key concept)
    - Brothers can write comments on a Rushee's profile.
- Administrator Accounts
    - Create and remove Admin accounts
    - Create and delete brother accounts
    - Change status of Rushees
    - Delete Rushees
- Brother Accounts

    - Assign Brother as the Primarxy Contact for several Rushees
    - Receive Action Reminders for Rushees the Brother is the primary contact for, via email and a view on the Brother's account page
- Presentation Mode
    - A summary of all the Rushees and their profiles
    - Highlight Overall Status of Recruitment
- Create Events
    - Date, Name, and Event Details
    - Monitor Which Rushees are Attending Which Events
- Reminders
    - Daily messages are sent to Brothers (via email) reminding them to invite Rushees to events, meals, etc.
    - The messages contain information about the status of the Rushee and what are the next actions the Brothers need to take
    - Brothers will be able to view reminders of their pending actions on their Homepage
- Profile Search
    - Filter Rushee Profiles by Names, Activities, Potential Major, Hometown, etc.
- Voting System
    - Brothers can upvote Rushees
    - A brother has one vote per Rushee. It is either present, or not. Only upvotes, à la Hacker News, or Facebook likes.
- Mobile Version of the Site
    - Allows quick information updating
    - Easy Access to Rushee Contact Information
    - Simplified, focused layouts
- Social Media Integration
    - Our application will pull Rushee images from their Facebook profile for use as a profile picture
    - Can display a Rushee's Twitter feed on their profile page

Security Concerns:

Security Policies

- Only authenticated Brothers (and Admins) can access the system.
- Only Admin accounts can authenticate new users.
- Only Admins can remove Rushees, Brothers, and Events.
- Users passwords are encrypted and stored as a hashed digest.

Threat Model

- An unauthenticated user can try to construct requests to the application.
- A user with basic credentials can construct requests or make requests via forms.
- A third party website can attempt to load data from application.

Specific Concerns

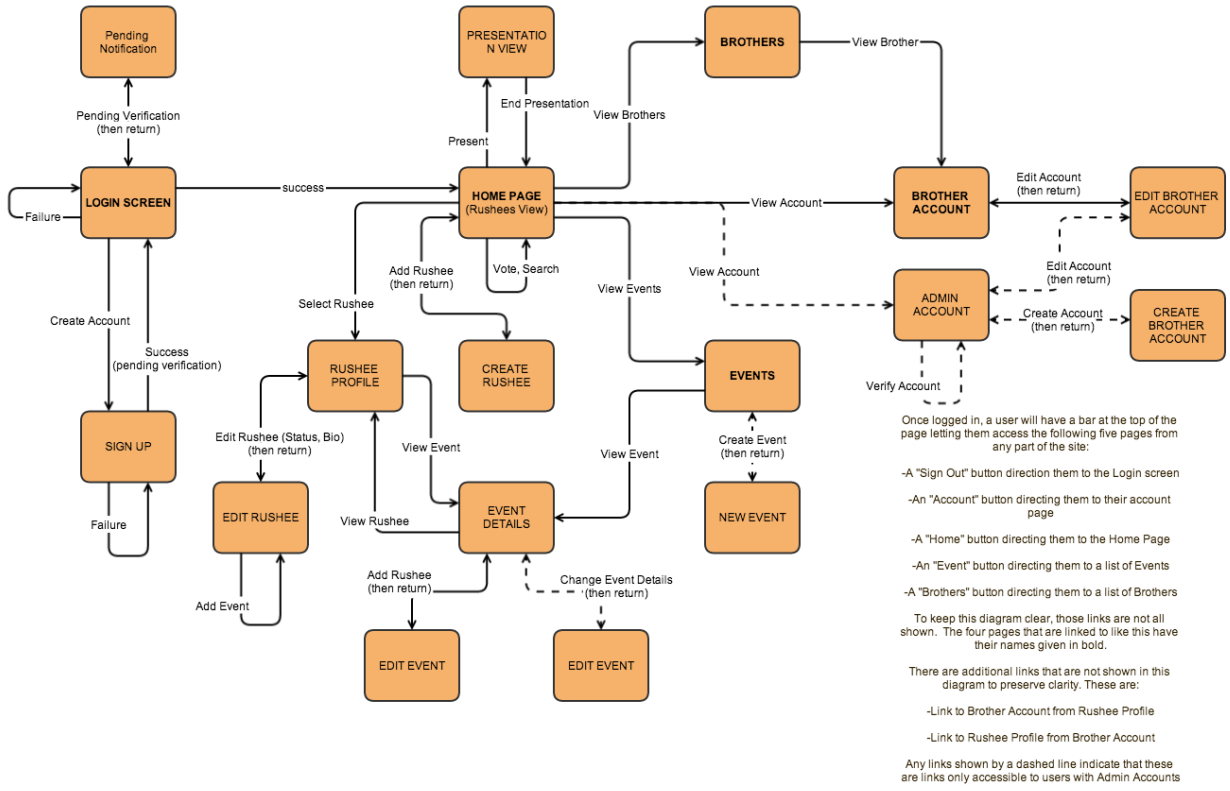
- An anticipated concern is that of malicious users impersonating Brothers in order to gain access to the system. In order to prevent this scenario from occurring, all new accounts need to be verified by Admins before they are granted access.
- Another concern is that of a malicious Brother, or a malicious user who has gained control of a Brother's account, corrupting the data stored in the application. In order to prevent data from being maliciously removed, only Admins are allowed to delete information from the application.
- There is also the potential that an Admin's account can be compromised. In this scenario, short of rolling back the database, it is very difficult to mitigate the fallout.

Common Concerns

- SQL Injection
    - Mitigation:
        - The application avoids calling an interpreter and only builds on the fly commands with expression objects, not strings.
        - The application relies on Rails' ActiveRecord ORM and uses only parameterized queries.
        - All used methods format and sanitize passed inputs for any queries.
- Cross Site Scripting
    - Mitigation:
        - We primarily rely on the defenses built in to Rails 4 to prevent XSS attacks.
        - We can prevent XSS by implementing Strong Parameters/Mass Assignment. Throughout the application, we use *require* and *permit* to whitelist what arguments can be passed.
        - We are sure to escape all HTML and stings rendered in the view. We do not disable this or use raw.
- Cross Site Request Forgery
    - Mitigation:
        - The application uses Rail's built-in CSRF Security Token to prevent against forgery.
- Packet Sniffing
    - Mitigation:
        - In order to prevent Packet Sniffing (as well as defend against XSS), we implement Transport Layer Security along with HTTP Secure. This encrypts the packets and prevents attackers from reading the transmissions.

User Interface:

Website Flow Diagram:



*A file containing a larger image is saved in the docs folder under ‘Workflow Diagram v1.png’.

Wireframes:

Please refer to the Wireframe Document.

Challenges

Initial Database Schema Design Choices:

- Database Tables
    - **Brother Model** (Devise gem takes care of password / email)
        - STRING firstname
        - STRING lastname
        - INT fraternity_id (not implemented in MVP)
        - BOOLEAN is_admin
        - BOOLEAN is_verified
    - **Rushee Model**
        - STRING firstname
        - STRING lastname
        - STRING email
        - STRING cellphone
        - STRING facebook_url
        - STRING twitter_url
        - STRING profile_picture_url
        - STRING dorm
        - STRING room_number
        - STRING hometown
        - STRING sports
        - STRING frats_rushing
        - INT primary_contact_id
        - STRING action_status
        - STRING bid_status
        - INT fraternity_id (not implemented in MVP)
    - **Approval Join Table**
        - INT brother_id
        - INT rushee_id
        - BOOLEAN vote
        - BOOLEAN met
    - **Comment Join Table**
        - INT brother_id
        - INT fraternity_id (not implemented in MVP)
        - INT rushee_id
        - DATETIME timestamp
        - TEXT comment_text
    - **Event Model**
        - STRING name
        - DATE date
    - **Event Attendance Join Table**

        - INT event_id
        - INT rushee_id
    - **Action Model**
        - INT brother_id
        - INT rushee_id
        - DATETIME expiry
        - TEXT action_text

Design Challenges and Decisions:

➢ **Implementing search/filtering:**
- A feature of the application will allow users to filter Rushee profiles. In order to implement this feature, we considered either using a gem or using Javascript to dynamically hide profiles. Due to the limited number of potential Rushees (this number is usually less than 40), we feel that using Javascript will be sufficient.

➢ **Rushees without social media:**
- Another feature of this application is the ability to integrate Rushee profiles with their social media accounts. However, we anticipated that some Rushees may not have social media account or their accounts will be difficult to find. In order to mitigate the problems that can be caused by this, users are able to upload their own images to a profile. This will increase the storage resources required by the application and may introduce other challenges, such as verifying the suitability of an image.

➢ **Image Storage:**
- We considered two potential options in order to handle displaying images - either we would share a link to a Rushee's photo on Facebook or we would store a Rushee's images on an external server dedicated to storing static assets. The first option has the benefit of not requiring us to provide our resources for storage. However it introduces several challenges, the primary one being what to do if the Rushee does not have a Facebook account. Therefore, we have decided to store the images on our own server in order to facilitate uploading of images by users.

➢ **Reminders:**
- During the design of the application, the question was raised whether or not a Reminder should be an object in the Data Model. Initially, it was decided that a Reminder is just a text field under a Rushee and does not have a relationship with any other objects in the Data Model. However, this introduced several challenges. If a Reminder was not an object, it would be more difficult to keep track of which user should be the recipient of a Reminder and which Rushee is the reminder related to. Also by having a Reminder object, it becomes easier to automatically

generate Reminder emails. Therefore in our design, a Reminder is an object that has a one-one relationship with a Rushee/Primary Contact pair. This relationship thus contains the necessary information - who the Reminder is for and who is it referring to.

➢ **Variety of Use Cases**

- An interesting question to be considered for the application is how varied the use cases actually are. While we focused entirely on the Rush use case from the very beginning of the design process, it is possible to imagine this application being used in other scenarios where there is an initial time-period of potential members meeting existing members of a society. Keeping this in mind, a possible future extension to the application is perhaps making the design (layouts/visuals) and names more universal.

➢ **Brother - Rushee Relation:**

- An important design consideration is the relationship between a Brother and a Rushee. We would like to provide brothers the option to view and add comments to all of the Rushee profiles, while being able to keep track of who the primary contact is for a Rushee. In order to provide this flexibility, two possible options were considered. Either the primary contact would just be a field and there would be no explicit relation between a Brother and a Rushee, or there would be a one to many relationship between a Brother and Rushees. By choosing the latter, it provides us with more flexibility and lays the groundwork for the Reminder feature.

➢ **Votes:**

- Votes are a way of expressing the sentiment of the Fraternity towards each Rushee. A design decision regarding this is the definition of what a Vote is and its subtypes. In our design, to keep the voting system simple, there is only one type of Vote - a Upvote, similar to a Facebook Like. Each Brother can vote once per Rushee. To ensure this uniqueness, we have designed Vote to be a separate model object, that belongs to both a Brother and Rushee. Since the number of Rushees is relatively small, it is trivial to add them up every time.

➢ **Security and Admin Accounts:**

- The design of the application introduces several security challenges that needed to be addressed through the use of Admin accounts. In order to prevent ‘fake’ Brothers from signing up, or to preserve the integrity of the application’s information, we only allow Admin accounts to verify new users and only Admin accounts are allowed to delete any data from the application. The introduction of the Admin accounts raises new design challenges as well. First, we considered how we would differentiate between Admin accounts and regular Brothers. An

Admin could either be a new type of data object or a boolean field can be set in the regular user object. As an Admin is merely a subset of users, the boolean field approach was taken. Another design challenge introduced was considering how new Admin accounts would be set, and whether or not there should even be multiple Admin accounts. In order to provide Admin accessibility to multiple types of Brothers (Fraternity President, Rush Chairman, Event Planners), we decided that multiple Admin accounts will be allowed, and that accounts with Admin privileges would be able to designate new accounts to become Admins.

- **How do we create new brother accounts?**
  - **Option 1**: Admin creates account and sends to brothers somehow
    - Pros: All new users are easily verified by the fact the Admin created them.
    - Cons: Admin must manually enter Basic User information and passwords.
  - **Option 2**: Brothers create accounts and cannot do anything until their account is verified by Admin
    - Pros: Allows brothers to enter their own individual information.
    - Cons: Forces users to wait on Admin. Admin may have to check spontaneously/maliciously generated users.
  - **Decision**: Both. This will allow the application to create users with both methods.