uDorm: The College Roommate Matching Service

Leela Sedaghat
Department of Computer and Information Sciences
Towson University
8000 York Road
Towson, MD 21204 USA
lsedag1@students.towson.edu

Trevan Jackson
Department of Computer and Information Sciences
Towson University
8000 York Road
Towson, MD 21204 USA
tjacks3@students.towson.edu

ABSTRACT

In this paper, we describe the design and implementation of a new database-driven Web application called uDorm using the Ruby on Rails framework. uDorm helps current undergraduate Towson University (TU) students, including incoming freshman, to find compatible roommates. Users must register an account with a valid TU e-mail address in order to use the application. After registration, users must complete a profile before being able to access the application's features and functionality. A matching algorithm uses information provided in the profile to suggest suitable roommates to the user. While this matching service provides the crux of the application, uDorm offers a wealth of other functionalities: users are able to search for other members based on name or email address, view other member profiles, send and receive messages from other members, post items for sale in a classifieds section of the application as well as ask questions anonymously about residence life in a Q&A section, which is also open to unregistered users in order to invite more content. Ajax, JavaScript and mashup are used throughout the application to provide a richer user experience.

General Terms
Documentation, Design

Keywords
Ruby on Rails, Ajax, JavaScript, Mashup

1. INTRODUCTION

Over the past five years, the number of applications on the World Wide Web has skyrocketed. One factor that has driven the growth of such websites is the rising popularity of open-source frameworks, such as Ruby on Rails, which are geared toward high-productivity and, as such, allow rapid web application development. By following a "convention over configuration" design paradigm, allowing agile development and integrating an Object-Relational Mapping layer, Rails makes the once difficult and time-consuming process of Web development both quick and enjoyable.

We chose Rails as the framework to use for the development of an application that would allow undergraduates at TU to find suitable roommates. While students at any university could potentially use this application, we chose to limit the user population to the TU community for its initial release. As such, registration is restricted to students with a valid TU email address.

In addition to matching students with suitable roommates, our aim was also to provide users with the basic functionalities that can be found on other social networking Web applications, such as creating and maintaining a personal profile, viewing other member profiles, searching for other users by name or email, and sending and receiving personal messages within the application. Certain additional functionalities that are relevant to student residential life were added to further enrich the application. These functionalities include a classifieds section in which users can post items for sale as well as a Q&A forum where all individuals can (anonymously) ask or answer questions about residential life.

2. DESIGN

The design of uDorm was an iterative process. Before commencing any programming, we did some initial design planning, including constructing user stories for the application, building an Entity-Relationship Diagram (ERD) for the database as well as site map diagrams.

2.1 User Stories

User stories are generally brief, nondescript ways of expressing the different actions a representative user would like to perform in the application. Each action that the user wants to perform corresponds to a single user story. Generating a list of user stories during the design phase of development was important to be able to identify the features that needed to be implemented in our application as well as the different types of users that would be using our application.

We found listing the user stories in table format (see Table 1) allowed us the following conveniences: to enumerate each user story with a unique identification number to which we could refer in discussion during development, to associate a "Priority" field to each user story so that we could clearly visualize which entries should take precedence in development, and to associate an "Assigned to" field to each user story so that we could keep track of which team member was responsible for completing each entry.

Table 1. The first nine user stories in the design of the uDorm application are shown in table format

As a user…				
User Story ID	**I want to**	**So that**	**Priority**	**Assigned to**
1	Be able to create an account	I can use the application	Mandatory	
2	Be able to log in and log out of my account	I can use the application and access my own personal account	Mandatory	
3	Be able to create a profile, listing my name, a photos, a short bio	Other users can learn more about me	Mandatory	
4	Be able to complete a questionnaire	I can tell the application about my preferences/traits	Mandatory	
5	Be able to match with other users based on my preferences from the questionnaire	The application can find potential roommate matches	Mandatory	
6	Be able to view my potential roommate matches	I can decide on who to request as my roommate	Mandatory	
7	Search users by name and email address	I can find a specific user to send a message to, view their profile or to request as a roommate	Mandatory	
8	Be able to send messages to other users	I can communicate privately with potential roommate matches	Mandatory	
9	Be able to make formal roommate requests	The university can see whom I would like to be roommates with	Mandatory	

2.2 Entity-Relationship Diagram

After user stories were generated, an ERD was constructed to design the database behind our application. Since the construction of an ERD is generally an iterative process, we were aware that the ERD designed during the design phase would not fully reflect the final database schema. However, by identifying necessary entities and attributes, this initial ERD (see Figure 1) provided a good starting point from which to identify essential classes and objects to begin development of the application.

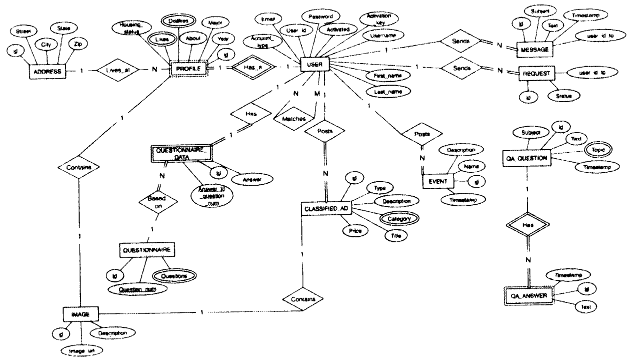


Figure 1. ERD constructed during design phase and used to commence development

2.3 Sitemaps

Sitemap diagrams were constructed to visualize the flow of user actions and to determine site navigation. One sitemap showed the hierarchy of information and functionality provided in the entire Web application (see Figure 2) and the other was more specific to the mapping the process of user registration (see Figure 3).

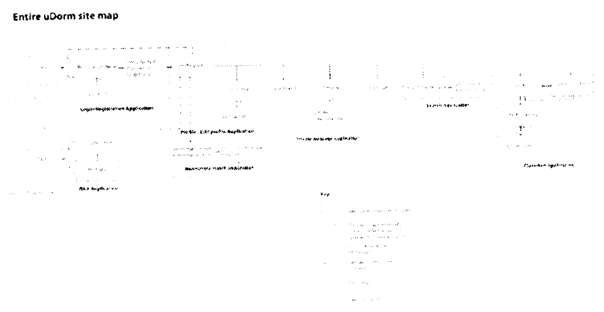


Figure 2. uDorm sitemap

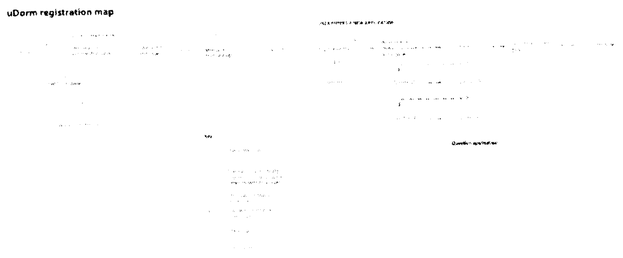


Figure 3. uDorm user registration sitemap

3. GUI DESIGN AND IMPLEMENTATION

The design and development of the uDorm interface was unique and challenging to implement. The initial design had to solve many problems that plague design patterns and accessibility issues in common social network applications. The actual design solved many of those issues with careful and strategic planning. Many of the major components within the interface are carefully considered and placed based on the best usability and accessibility issues within the application. This section will delve into the process of GUI design and implementation, the steps that were used to tie the entire front-end user interface with the backend logic.

3.1 GUI Design

After studying the application design and user flow, the actual GUI design was templated and tested in many browsers for cross-browser compatibility issues. Before any code was actually written the interface was carefully designed using design software i.e. Photoshop. The application colors were carefully chosen to emulate a strong social setting. A mixture of dark blues and light grays fill the navigation, background, and other areas of the application. All of the major pages and page functionalities were designed using the design software as well. After the design process has been completed the next step was to start writing and implementing code. With the comprehensive understanding of the available browsers that users may use to view the application, every templated section was tested using a development tool (i.e Firefox, Firebug) for compatibility issues.

3.2 Global Stylesheet and CSS Reset

The entire application was designed using a global stylesheet. The global stylesheet controls the styles for every element in the application. Using standard compliant CSS and XHTML mark-up, each element in the application has its own declaration. The stylesheet is broken down into sections of hierarchy and organized based on how they are displayed on certain pages in the application. Some variations of intricate implantation were involved when determining various layout schemes. The application was mostly based on a two-column layout. The only page that required three columns was the profile page. The application's page dimensions were carefully determined and compared against the standard dimensions for web applications. The decision to make the uDorm application dimensions 850px wide was determined based on the amount of information the application displayed at any given time.

3.3 JavaScript – JQuery User Experience

The implementation of JavaScript and its well-known counterpart framework jQuery is very useful for the enrichment of the user experience. UDorm used these two technologies interchangeably to enrich our users' overall experience throughout the application. JavaScript and jQuery can be seen on the application's index page and registration page. The recent user scroll uses JavaScript and jQuery to automatically scroll a set amount of recent users. When users click on the "Terms of Use" link before registering with the application, a modal window will appear and present the user with the terms of use documentation. This was considered a usability bonus, because instead of presenting the user with an entirely new page, the modal window appears before the user's eyes and allows the user to stay on the current page to view this very important information. The modal window not only called various jQuery functions, but it also relied on being able to dynamically change CSS elements as well. For instance, when a user clicks the "Terms of Use" link, it calls several functions one being a CSS function that invokes the background property and applies an opacity property to that element.

4. ENVIRONMENT

uDorm was developed on a Mac OS/X platform using Rails 2.2.2, Ruby 1.8.6 and SQLite (an embedded SQL database). The Mongrel Web server was used for deployment during development.

5. IMPLEMENTATION

5.1 Functionality

5.1.1 User Registration

Users must register, create and activate their accounts in order to be able to use the uDorm application. The user registration form is located on the home page of the application (see Figure 4). Users must be a student at TU in order to register. In order to enforce this restriction, we implemented a validation using regular expressions in the User model to check that the email address entered in the user registration form fits the format of a valid TU student email address. Any email address not matching the TU student email address format will fail validation, and the user is notified on the page that a valid TU email address is required to proceed with user registration.

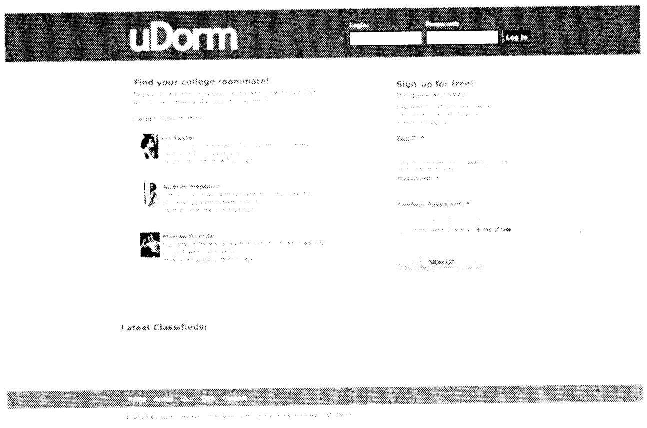


Figure 4. uDorm home page prominently displays the user registration form on right-hand side. The home page also displays a scrolling list of recent members who have joined as well as a listing of the most recent classified ads that have been posted in the application. The top-right corner displays a login form for registered users to sign in.

In addition to having a valid TU student email address, validation for user registration includes checking that no other user has already created an account with the entered email address, that none of the fields have been left blank, that the password and password confirmation fields match (and are longer than 6 characters), and that the user has agreed to the Terms of Use by clicking on the checkbox.

Once all of the fields have successfully passed validation, the page will display a message thanking the user for signing up and notifying him/her to expect an email containing an activation code (see Figure 5).

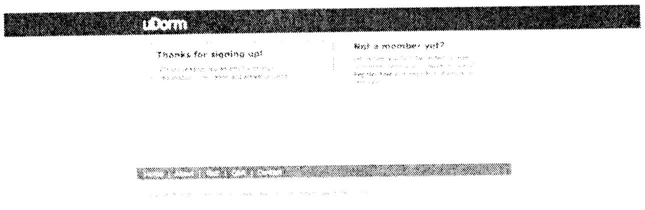


Figure 5. uDorm page displayed after user account created

To further ensure that users are members of the TU student community, all newly registered accounts must be activated through a unique URL sent to the registered email address. Once the user visits this unique URL, the account is marked as activated in the User model and the user is redirected to the dedicated login page to enter his/her login and password (see Figure 6).

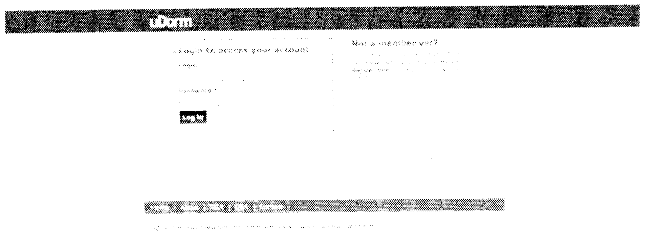


Figure 6. uDorm login page displayed after a user has successfully activated his/her account

In order to ensure consistency with other web applications that are used in the Towson University community, such as Webmail, Blackboard and PeopleSoft, each login is the user's own Towson University ID (TUID) instead of the user's TU email address. Though the user did not enter their TUID in the user registration form, a method was implemented within the User model to parse the email to generate the TUID and store it in the login attribute of the User model.

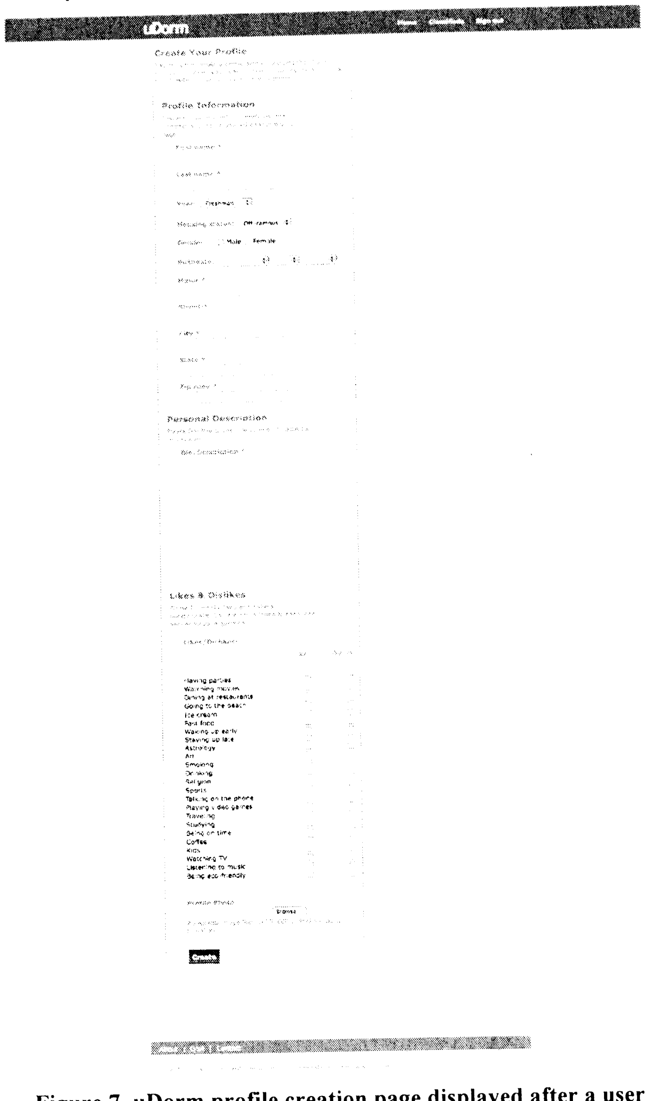
Again, all user input for login undergoes validation to ensure that fields are not left blank and that the login and password match an activated account. The user may choose to login from the dedicated login page or he/she may return to the application home page and login using the form in the top-right hand corner of the screen.

5.1.2 Profile Creation

After a user has logged in for the very first time into the uDorm application, he/she will be prompted to create a profile. Since the user profile is central to the application, the system was designed such that users must create a profile in order to begin using the application. If the user decides not to create a profile and to sign out of the application, the next time the user signs in, he/she will again be prompted to create a profile.

The process of profile creation is quick and simple. Users are prompted with a form to enter some personal information, such as name, gender, birthday, and a short bio or description of themselves, and some academic information, such as their major and year in school. Users must also specify their current housing status by selecting whether they are living on-campus or off-campus. Users also have the option to upload a small image that will be displayed on their profile page as well as in other places within the application (e.g. search results, recent members on the application home page) for other users to see.

Lastly, users are presented with a list of personality traits, hobbies and general concepts from which they can select to designate their own personal preferences. This section is called "Likes/Dislikes" in the application since users select, quite simply, whether they like the concept listed or whether they dislike it. Users may elect not to designate a preference on any of the options. The "Likes/Dislikes" section, along with the user gender and specified housing status, provides the basis of matching users to potential roommates. Users may also modify their input to any of the fields in the profile creation page later on using the edit profile page.



uDorm

Create Your Profile

Profile Information

First name *

Last name *

Year: Freshman

Housing status: Off campus

Gender: Male Female

Birthdate:

Major *

Address *

City *

State *

Zip code *

Personal Description

Bio Description *

Likes & Dislikes

Likes/Dislikes

Having parties
Watching movies
Dining at restaurants
Going to the beach
Ice cream
Fast food
Waking up early
Staying up late
Astrology
Art
Smoking
Drinking
Religion
Sports
Talking on the phone
Playing video games
Traveling
Studying
Being on time
Coffee
Kids
Watching TV
Listening to music
Being eco-friendly

Profile Photo

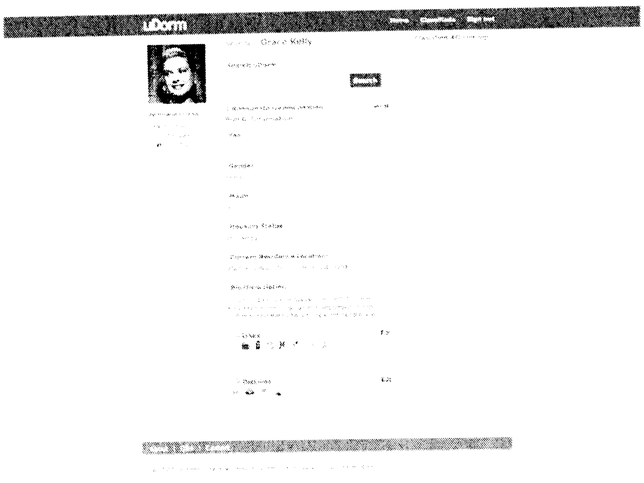
Browse

Figure 7. uDorm profile creation page displayed after a user logs into account for first time

In order to successfully pass validation and to create the user profile, all fields must be supplied values with the exception of the Likes/Dislikes section and adding a Profile Photo. If a profile photo is not uploaded, a default graphic will be used.

5.1.3 Profile Page

After a user creates their profile for the first time, the user will be forwarded to his/her profile page (see Figure 8). This profile page will serve both as the landing page upon signing into the application and the user's home page when he/she is signed into the application (i.e. clicking on the "Home" link at the top navigation bar will always return the user to his/her profile page). The profile page displays all of the information entered upon profile creation in an organized manner. The information entered for the user's Likes and Dislikes are presented in the form of colorful icons, which makes the page more visually pleasing. If the user is unsure what the icon represents, he/she can hover the mouse over the icon and the description of the icon (retrieved from the Likedislike model in the database) is displayed.



uDorm

Grace Kelly

Search uDorm

Gender

Major

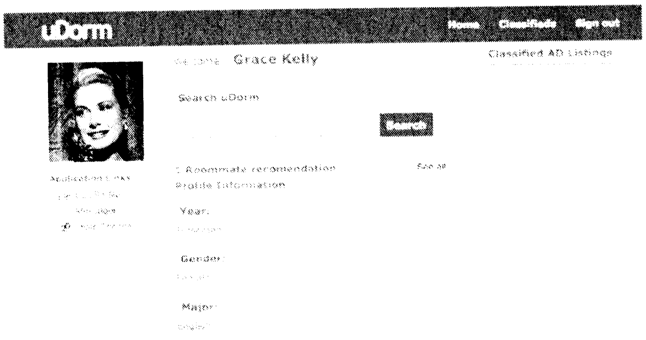
Housing Status

Likes

Dislikes

Figure 8. uDorm profile page

In addition to displaying the user's information, the profile page includes a listing of recent classified ads in the sidebar as well as links to access the user's (sent and received) messages and to edit current profile information. The number of any potential roommate matches are also displayed at the top of the page with the option to "See all" of the matches. Clicking on the "See all" link will expand the div to show a list of all current matches the user has in the system without reloading the page (see Figures 9 and 10). Links are provided under each match to allow the user to view the match's profile or to send the match a message.



uDorm

Home Classifieds Sign out

Grace Kelly

Classified AD Listings

Search uDorm

Search

1 Roommate recomendation

See all

Profile Information

Application Links

Messages

Year:

Gender:

Major:

Figure 9. uDorm profile page before clicking on the "See all" roommate recommendations link

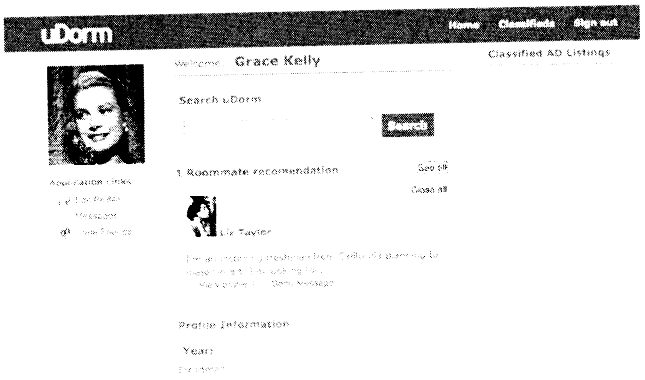


Figure 10. uDorm profile page after clicking on the "See all" roommate recommendations link

Furthermore, a search form is present within the profile that allows users to search for other users by name or email address.

Ajax is implemented on the profile page to allow in-place modification of the user's likes and dislikes. By clicking on the corresponding "Edit" link for the Likes or Dislikes section, a hidden div will appear on screen without the need to reload the page (see Figures 11 and 12).

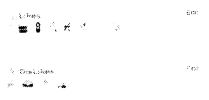
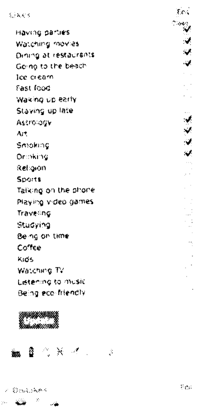


Figure 11. The Likes and Dislikes section on a sample uDorm profile page before clicking on the "Edit" Likes link



Having parties	✓
Watching movies	✓
Dining at restaurants	✓
Going to the beach	✓
Ice cream	
Fast food	
Waking up early	
Staying up late	
Astrology	✓
Art	✓
Smoking	✓
Drinking	✓
Religion	
Sports	
Talking on the phone	
Playing video games	
Traveling	
Studying	
Being on time	
Coffee	
Kids	
Watching TV	
Listening to music	
Being eco-friendly	

Figure 12. The Likes and Dislikes section on a sample uDorm profile page after clicking on the "Edit" Likes link

The div presents the user with the list of Likes/Dislikes and the corresponding checkboxes will be checked to represent the user's current designated preferences. The user may uncheck boxes already checked or check new boxes to designate new preferences. After clicking on the "Update" button, the div will close and the icons appearing on the profile page in the respective section will be updated without reloading the page to represent the new preferences. Alternatively, the user may close the expanded div without making any changes by clicking on the "Close" link.

The profile page, however, must be reloaded in order to generate new roommate matches based on the new preferences.

5.1.4 Editing Profile

At any time, the user may decide to modify his/her profile information, which can be done by clicking on the "Edit Profile" link under the profile photo in the Application Links area. A similar form will be displayed from when the user first created the profile, except that all fields will be pre-populated with information that is currently in the system (see Figure 13). This form is subject to similar validations as when the user first created the profile. Once the user clicks on the "Update" button, he/she will be redirected back to his/her profile page. If any Likes/Dislikes were changed, new potential roommate matches will be generated and displayed upon return to the profile page.

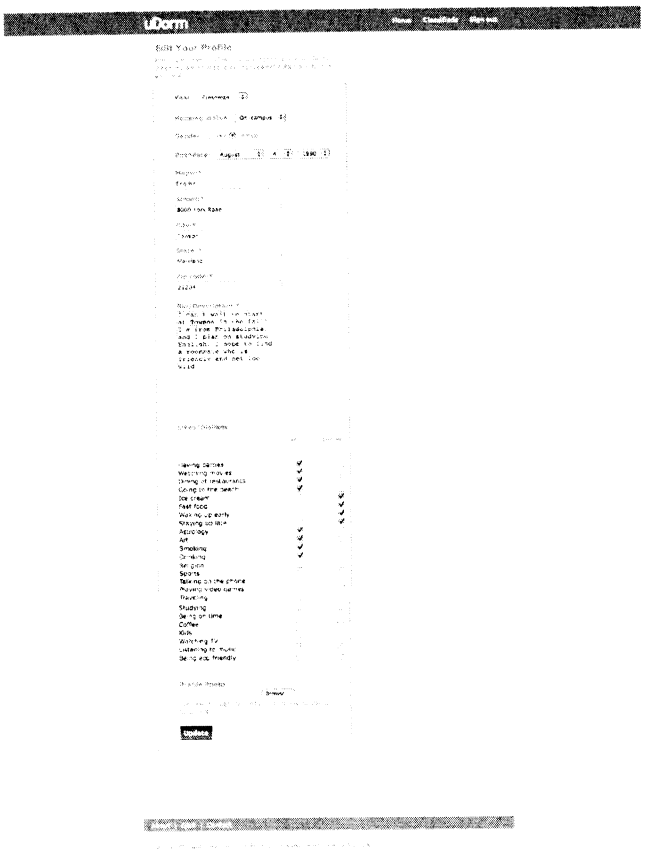


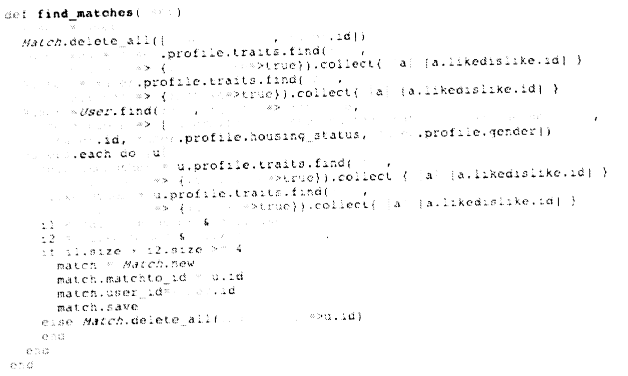
Figure 13. uDorm page to edit profile

5.1.5 Viewing Profiles

All user profiles can be viewed by any registered user. A user may navigate to another user's profile by clicking on the "View Profile" link associated with match listings, search results and incoming messages. The view used to display another user's profile page is different from the view used to display the current user's profile page, because certain pieces of information (e.g. roommate matches) as well as certain links (e.g. Edit Profile, Messages, etc) are not appropriate in this context. Instead, other links are displayed when viewing another user's profile (e.g. "Request" or "Send a Note").

5.1.6 *Matching*

Matching users as potential roommates is central to the purpose of this application. The profile information forms the basis to make this match. A method called find_matches was written in the Profile controller to match users (see Figure 14). The method find_matches is called every time the user makes a change to his/her likes and/or dislikes, whether it is when the user initially creates his/her profile, when the user edits his/her profile or when the user uses the in-place edit feature on the profile page to modify his/her likes or dislikes.



```
def find_matches( [illegible] )
  [illegible]
  Match.delete_all([ [illegible] , [illegible] .id])
  [illegible] .profile.traits.find( [illegible] ,
          => { [illegible] =>true}).collect{ |a| |a.likedislike.id| }
  [illegible] .profile.traits.find( [illegible] ,
          => { [illegible] =>true}).collect{ |a| |a.likedislike.id| }
  [illegible] User.find( [illegible] , [illegible] => [illegible] ,
          => [ [illegible] ,
     [illegible] .id, [illegible] .profile.housing_status, [illegible] .profile.gender])
  [illegible] .each do |u|
     [illegible] u.profile.traits.find( [illegible] ,
          => { [illegible] =>true}).collect { |a| |a.likedislike.id| }
     [illegible] u.profile.traits.find( [illegible] ,
          => { [illegible] =>true}).collect{ |a| |a.likedislike.id| }
    i1 [illegible] & [illegible]
    i2 [illegible] & [illegible]
    if i1.size + i2.size >= 4
      match = Match.new
      match.matchto_id = u.id
      match.user_id= [illegible] .id
      match.save
    else Match.delete_all( [illegible] =>u.id)
    end
  end
end
```

Figure 14. Code for find_matches method in Profile controller

First, the method deletes all current matches in the system. This was necessary if a user changes housing status to ensure that all current matches (with the old housing status) are removed. Next, the method collects all the entries in the Traits table that belong to the current user for which the Boolean attribute isDislike contains the value "true" and stores the unique id of the likedislike to which each entry refers in the instance variable @dislikes. For example, if the user has selected that they dislike "Drinking," the id for drinking (i.e. 12) from the likedislikes table will be stored in the instance variable @dislikes. Essentially, the id of everything the user does not like is stored in the @dislikes variable.

Next, the method collects all entries in the Trait model that belong to the current user for which the Boolean attribute isLike contains the value "true" and stores the unique id of the likedislike to which each entry refers in the instance variable @likes. For example, if the user has selected that they like "Art" and "Having parties," the ids for art and having parties (i.e. 10 and 1, respectively) from the likedislikes table will be stored in the instance variable @likes. Again, essentially, the id of everything the user likes is stored in the @likes variable.

Then, the method collects all users in the Users table (after a join with the Profiles table) whose id does not equal the current user's id (this is to ensure the user is not matched to himself/herself) and whose housing status and gender are the same as the current user. The collected users are stored in the instance variable @users. The @users variable now contains all the users with which the method will compare the current user. It was imperative that only users who have similar housing status (e.g. On-campus) be matched to each other. Similarly, it was important that only users of the same gender be matched since TU requires roommates to be of the same gender. In the future, this method can be modified to allow users who have a housing status of Off-campus to be matched to users of either gender.

Next, the method iterates over each user stored in the @users variable. In each iteration, the method collects all the entries in the Traits table that belong to the user for which the Boolean attribute isDislike contains the value "true" and stores the id of the likedislike to which each entry refers in the instance variable @dislikesOther. Similarly, the method collects all the entries in the Traits table that belong to the user for which the Boolean attribute isLike contains the value "true" and stores the id of the likedislike to which each entry refers in the instance variable @likesOther. This is similar to the process that was taken at the beginning of the method for the current user.

At this point, the method contains two arrays for each user (the current user and the user to which he/she is being compared): an array containing the likes of each user (@likes and @likesOther) and an array containing the dislikes of each user (@dislikes and @dislikesOther). The intersection of the set of likedislikes ids stored in @likes and @likesOther is found and stored in the variable i1. This intersection is representative of the things that both users have in common in terms of their "Likes" (i.e. what they like). Next, the intersection of the set of likedislikes ids stored in @dislikes and @dislikesOther is found and stored in the variable i2. This intersection is representative of the things that both users have in common in terms of their "Dislikes" (i.e. what they do not like).

Finally, the size of each array (i1 and i2) are added to each other to find the total number of similarities the two users have in their preferences. If the two users have more than 4 preferences in common, a new entry is created and saved into the Matches table storing both users' ids. If the two users have less than 4 preferences in common, a new entry in the Matches table is not made and any existing Matches between the two users are deleted from the Matches table (note that this line to remove existing matches may be removed during re-factoring since all matches were already deleted at the start of the method).

The design of this algorithm and the Traits model ensures that if the user has not selected any Likes or Dislikes, they will not be matched with any other user (e.g. another user who has also not selected any Likes or Dislikes).

In the future, this algorithm can and should be made more selective and complex. Currently, only 4 similarities are needed to make a match. However, 4 is a relatively low number and will likely generate a lot of matches that may not be suitable. Other possible future changes to the algorithm include immediately disqualifying users who like or dislike opposite traits. For example, if a user likes "Staying up late" and another user likes "Waking up early," these users should not be matched no matter how many preferences they have in common. Similarly, if one user likes "Smoking" and another user dislikes "Smoking," these users should immediately be eliminated as matches, since smoking is often a deal-breaker when it comes to cohabitation.

5.1.7 *Search*

Users have easy access to search functionality from their profile page. Users may search for other users in the system by email address or by first and/or last name (see Figure 15).

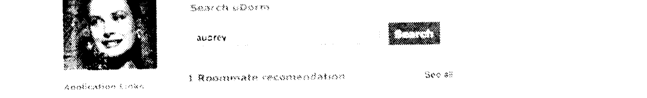


Figure 14. Cropped uDorm profile page showing the search term "audrey" entered in search form.

When a search term is entered, the user is redirected to a search page that will display all relevant results (see Figure 15).

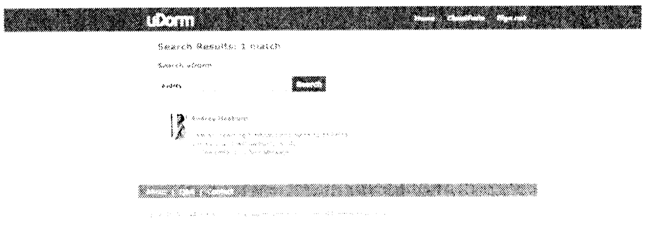


Figure 15. Search results page for the query "audrey"

Results are listed individually with links to view the user's profile and to send a message to the user. A partial was used to display this search field. Having used a partial made it easy to allow the user to conduct another search from the same page without having to return back to the profile page.

5.1.8 *Sending and Receiving Messages*

Communicating with other users is integral to any Web application that aims to connect individuals. As such, a private messaging system has been implemented from within the application. Users can send and receive messages to any other user who has registered, activated their account and created a profile.

In order to view any messages that have been sent or received, the user must click on the "Messages" link in the Application Links area located below their photo on the profile page (see Figure 16).

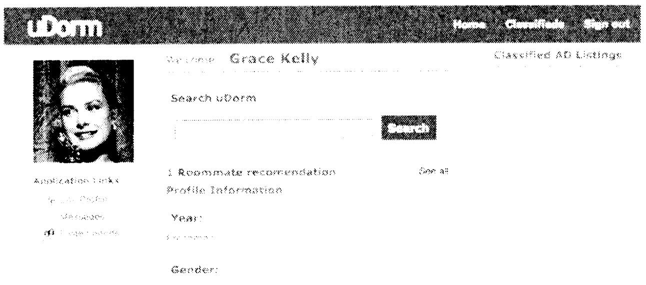


Figure 16. Cropped uDorm profile page showing the "Messages" link in the Application Links area

After clicking on the "Messages" link, the user is taken to the Mailbox area, which by default will display any messages that the user receives (see Figure 17). If the user has not received any messages, the page will display a message that the user has no new messages. If the user has messages (as in Figure 17), the name and photo of the sender is displayed as well as a truncation of the message body. There will also be links to view the sender's profile and to read the message.

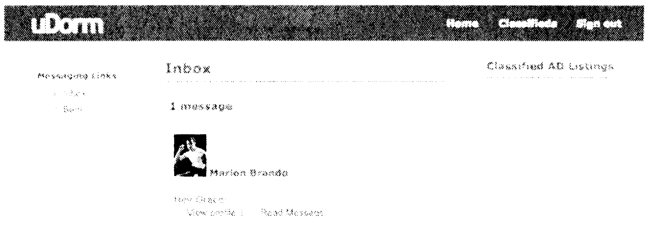
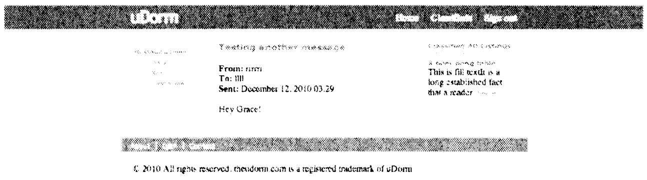


Figure 17. uDorm Inbox of messages to the current user

If the user clicks on the "Read Message" link, the contents of the message will be displayed on a new page (see Figure 18).


Testing another message

From: mm
To: llll
Sent: December 12, 2010 03:29

Hey Grace!

This is fill textlt is a long established fact that a reader

© 2010 All rights reserved. theudorm.com is a registered trademark of uDorm

Figure 18. Page displaying an individual message in the inbox

The message's subject is displayed at the top of the page, as well as "From" and "To" fields (displaying each user's respective TUID), a "Sent" field (displaying the time the message was sent) and the entire content of the message. The new page will also contain a link in the Messaging Links area labeled "Send a Reply." Clicking on "Send a Reply" will redirect the user to a page containing a form to compose a message to the selected user (see Figure 19).

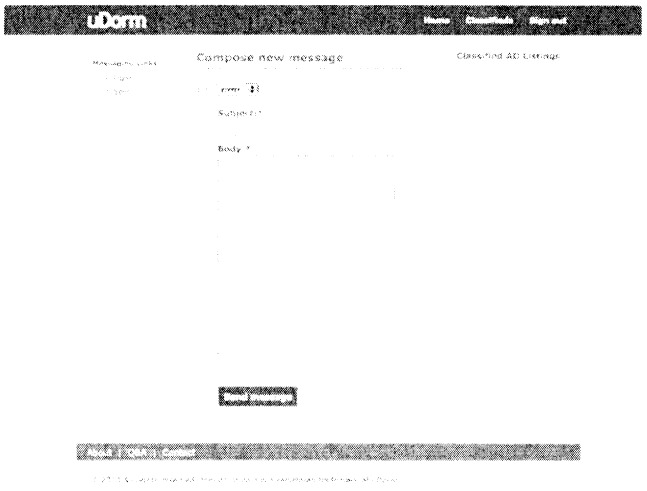


Figure 19. Page displaying a form to compose a message to another user

If the user would like to see the messages they have sent, they must click on the "Sent" link in the Messaging Links area of the Mailbox, which will redirect them to a page listing all messages that they have sent.

5.1.9 *Q&A Forum*

The original premise behind the Q&A forum was to develop an open platform where non-registered users can explore categories, questions and even post answers related to roommate and residential life topics. The actual application allows any user to create categories, questions and even post an answer to an already existing question. The original design took into consideration that all users would like the ability to be involved with everyday discussions about roommate etiquettes and any other related topic. The design behind the Q&A forum was simple. It involved developing three different controllers: a category controller, a questions controller and an answers controller. Designing the relationships between models involved an understanding of entity relationships and foreign keys. Each category has many questions, and one question has many answers. The implementation of this type of stand-alone application was considered vital to the entire uDorm application because it allows users to gain knowledge and post important relevant answers to topics.

5.1.10 Classifieds

uDorm classifieds allows registered users to post ads in relevant categories. The initial design took into consideration that users would like to take advantage of an open ad platform that allowed users to post ads that were relevant to the user's location. Classifieds that are posted to the application are seen throughout the uDorm application. Recent ads are posted on the applications homepage and in the sidebar of various pages in the application. The initial design of the classifieds application only used two different controllers to run the application: a category controller and a classifieds controller. Developing the relationships between models was as simple as determining that a category has many classifieds and a classified belongs to a category. The only way for a user to post an ad is if the user is logged in. Once the users session is recognized, a user is allowed to create a new ad. The ad is assigned to a user's id so each ad that is posted is relevant to a given user. As mentioned earlier, ads are posted throughout the uDorm application. However, a user is only allowed to view the details of an ad on any relevant page if the user is logged in.

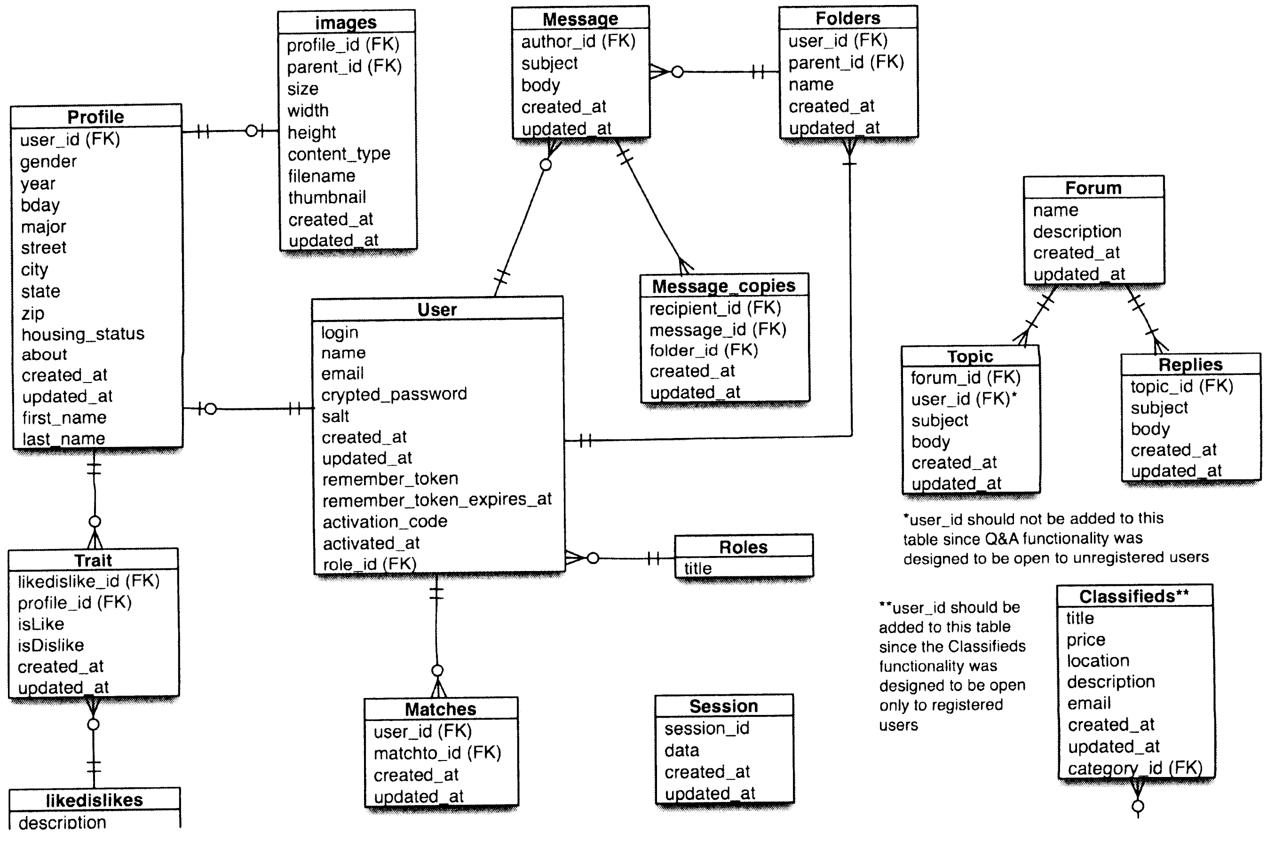


Figure 20. Final uDorm database schema

5.1.11 Mashup

uDorm application offers mashup capabilities. Users have the ability to view off-campus housing listings from an external source: the YAHOO Towson available apartment listings. uDorm integrates a repopulated list of housing data inside the classifieds application. This functionality is extended and made possible by including the feedtools gem.

The feedtools gem allows uDorm to connect to existing RSS feeds with a few simple steps. Including an instance variable inside the classifieds controller that connects and opens an existing RSS feed allows uDorm to gather data in an XML format. The XML data is then parsed and displayed in its own view within the uDorm application. When a user first visits the uDorm classified ads application, the sidebar displays a link that allows user to view the relevant housing listing.

5.2 Database Schema

After development of uDorm was complete, the initial ERD constructed during the design phase had been changed, as we expected. A diagram reflecting the final database schema was constructed and can be found in Figure 20.

6. PLUG-INS

The following plug-ins were used in the development of uDorm.

6.1.1 restful_authentication

Restful_Authentication was used to ensure secure user authentication (including secure password handling and account activation by email) and to facilitate user login and logout.

6.1.2 attachment_fu

Attachment_fu was used to facilitate the uploading and storage of user images to the application, such as for the Profile section.

6.1.3 acts_as_ferret

Acts_as_ferret was used to implement search functionality in the application

7. CONCLUSION

The goal of creating a functioning database-driven Web application that matches users with potential roommates was achieved. However, given time constraints, certain user stories were not implemented, such as allowing users to submit, accept or decline roommate requests to one another. After such functionality was implemented, the aim was to allow University admin to access the application to view current roommate requests in the system and assign roommates accordingly.

Also, certain functionalities should be added to the application to improve user experience, such as paginating search results or (sent and received) message listings. A large user base would necessitate such improvements.

Lastly, much of the code written for this application is in need of refactoring.

8. ACKNOWLEDGMENTS

We thank Dr. Siddharth Kaza for all of his guidance, feedback and encouragement and for being such a wonderful and patient teacher.

9. REFERENCES

[1] Ruby, S., Thomas, D., and Heinemeier-Hansson, D. 2009. Agile Web Development with Rails. Pragmatic Programmers (*Third Edition*).

Happenings: Social Networking Website for Music Events

Sharon Kirk
Computer and Information Sciences Department
Towson University
7800 York Road
Towson, MD 21204 USA
+1 443 604 1634
skirk2@students.towson.edu

Daniel Lenhardt
Computer and Information Sciences Department
Towson University
7800 York Road
Towson, MD 21204 USA
+1 443 604 1634
dlenhal@students.towson.edu

Daniel Strassler
Computer and Information Sciences Department
Towson University
7800 York Road
Towson, MD 21204 USA
+1 443 604 1634
dstrasl@students.towson.edu

ABSTRACT

In this paper, we describe the design, development, and implementation of Happenings, a social networking website built for people to share a common appreciation of music events and performers. Happenings is a Ruby on Rails web application that enables users to search for music events, add events to personal calendars, and interact with friends about their events and musical interests. The search function parses XML data returned from searching Last.fm, another music-driven website. Facebook inspired the friend interaction design. The end result is a mash-up of social networking functionalities.

Keywords

Social networking, Ruby on Rails, music, events, REST, Ajax, XML, Plug-ins

INTRODUCTION

Happenings is a social networking mash-up website for finding concerts, festivals, and any other music-related events that are usually ticketed. Its key features resemble Last.fm's Events section: browsing upcoming events, adding performers to follow their tour plans, and using calendars to track events that you or your friends will be attending. The ability to friend other users, create photo albums and write comments are popular Facebook features that are also integrated with Happenings.

Happenings users can coordinate meet-ups at events by commenting on an Event page or on friends' profiles. They can keep track of favorite artists and their tour dates. Users may find people with the same music tastes by reading their comments on Event/Performer pages or mutual friend's profiles.

We will discuss the design, implementation, testing, and technical details of how Happenings came to be.

DESIGN

Developing the initial design for Happenings required the creation and evaluation of use cases that were then used to design the database schema.

Use Cases

There are two primary kinds of users: User and Administrator. Users could be performers, venue owners, or just members of the general public. Administrators are the website managers. They have authority to edit or remove inappropriate content.

User Use Cases

For the sake of brevity, we will compile the individual use cases into a list. Users can do all of the following:

- Create an account
- Login and logout securely
- View own profile and calendar
- View friend, performer, and event profiles and calendars
- View lists of friends, performers, and events
- Add friends, performers, events, and photos
- Delete friends, performers, events, and photos
- Accept and reject friend requests
- Receive friend request notifications via email
- Search for friends, performers, and events
- Comment on profiles and photos

Administrator Use Cases

Administrators can edit and delete users, events, performers, photos, and comments.

Database

Happenings' schema consists of eleven models (entities) and thirty-two tables. The Entity-Relationship diagram is provided in the appendix.

The eleven models are:

area	friend_mailer	ticket_seller
calendar	genre	user
comment	performer	venue
event	photo	

There are tables for each model and all of the relationships. Other tables include sessions, sqlite_sequence, and schema_migrations.

IMPLEMENTATION

Happenings was developed with Ruby on Rails [9], which implements the Model-View-Controller (MVC) software architecture as a framework for developing Web-based applications. The model maintains the configuration of data stored within the application, defines associations, and validates data transactions. The view's sole purpose is to display the user interface. Controllers contain the logic to coordinate the interactions between user inputs, models, and views [8].

Happenings' models represent the objects involved with the system. Users, events, performers, and calendar entries are examples of models. There are many views, such as the profile, lists of friends, performers, and events, calendar, and administrative views. The User controller contains methods for all of the user activities, the Login controller authenticates users and administrators, and the Admin module contains controllers for administrators to manage users, events, etc.

Specific details about versions, plug-ins, and other technicalities are presented later in the Technical Details section.

Views and Functions

This section describes the views and functions for user interaction with Happenings.

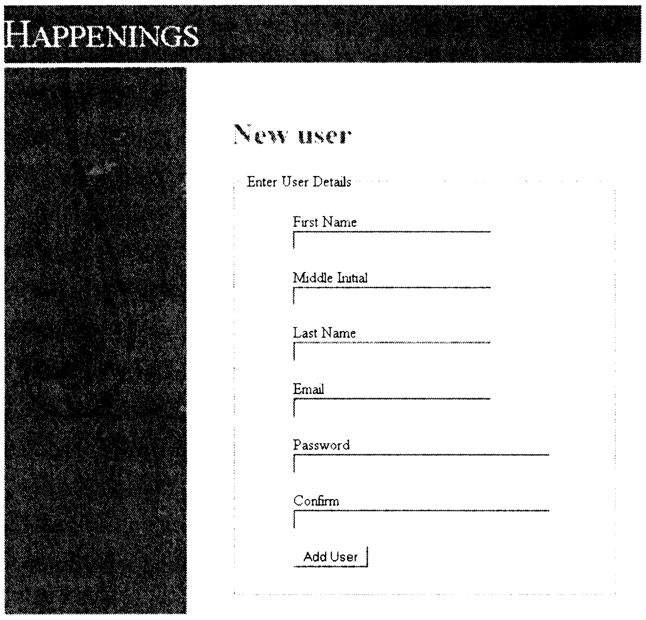
Layouts

Each view contains a reference to a standard user layout. It includes the Happenings banner, a horizontal menu of links and a sidebar containing links and the search bar. The horizontal menu displays the user's name and links to Profile, Calendar, Friends, Events, Performers, and Photos. The sidebar displays "Welcome to Happenings!" above the user's name and contains links to Home (Profile), Friend Requests, and Logout. If `user.role==1`, an additional Admin link is shown because the value 1 for `user.role` signifies that user is an Administrator. Otherwise, it contains the value 0 to signify a regular User.

The other pages, Login and Admin, utilize the parent layout. It is similar to the user layout except there are no menus or search bar.

Login

The Login page displays a form for user to enter email and password and a link to create an account. To create an account, `add_user` renders the view shown in Figure 1.


HAPPENINGS

New user

Enter User Details

First Name

Middle Initial

Last Name

Email

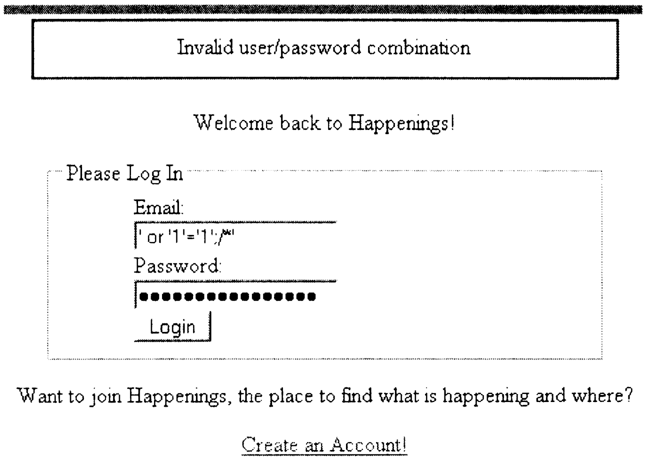
Password

Confirm

Add User

Figure 1: Form to create an account

`create_user` is called when the form is submitted. It creates the user in the database and then redirects back to profile. For login, the User model contains a method `def self.authenticate(email, password)`. When the user selects "Login," the Login controller invokes `user.authenticate` which searches the User table for the given email address. It encrypts the password that was entered to obtain the expected password and then checks for a match. Password encryption is described later in the Technical Details section. If the credentials match, Login controller redirects to the user profile page. Otherwise, the login page displays a flash notice indicating "Invalid user/password combination." This is shown in Figure 2.


Invalid user/password combination

Welcome back to Happenings!

Please Log In

Email: ' or '1'='1';/*

Password: ●●●●●●●●●●●●●●●●●

Login

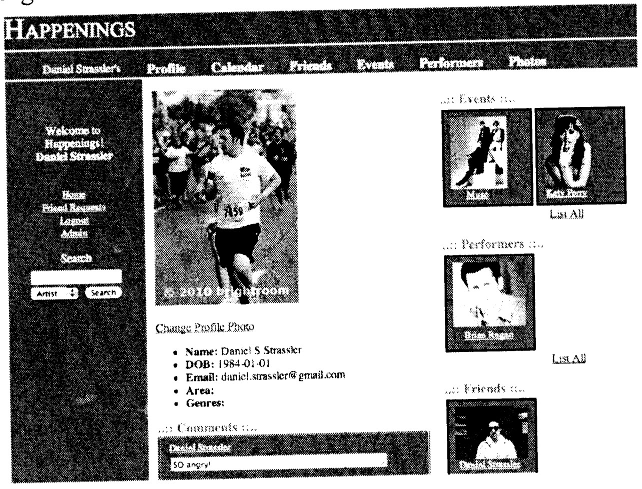
Want to join Happenings, the place to find what is happening and where?

Create an Account!

Figure 2: Failed login

Profile

The Profile dashboard is the most visually complex Happenings view. It displays the user photo, basic demographic information, and comments in the first column. The second column contains a small sampling of thumbnail photos for the events, performers, and friends that are associated with the user. A profile is shown in Figure 3.



- **Name:** Daniel S Strassler
- **DOB:** 1984-01-01
- **Email:** daniel.strassler@gmail.com
- **Area:**
- **Genres:**

SO angry!

Figure 3: Profile view

On the profile page, the sampling of Events, Performers, and Friends are rendered with partials.

For many pages, the available actions depend on the user's relationship to the profile owner. An initial check is performed in the controller method for the view. The method compares the `session[user_id]` of the viewing user with the `id` for the user to whom the page belongs. That determines the privileges the viewing user is granted. The privilege types are `user_self`, `user_friend`, `user_not_friend`, `not_logged_in`, `unknown`, and `user_admin`. Viewers who are not logged in or unknown are redirected to the Login page.

Someone viewing his or her profile can perform the following actions:

Change profile photo links to a form where the user can upload a photo. When the user clicks "Create," the User controller calls one of two methods: `update_profile_photo` or `create_profile_photo` if this is the user's first profile photo. That is because the method must create a new instance of Photo in the latter case.

Comment on profile uses a partial template `_comment_form` to receive the comment. When the user clicks "Comment," the comment list is updated using Ajax and another partial template `_comment` that renders each of the individual comments.

Read comments which are displayed using the two partial templates described above. The comments are found with a `user.comments` query that loops through the profile comments found in the `comment_user` table. They are ordered based on `updated_at` time, with most recent comments displayed first.

List all events, performers, or friends are links to views that are self-explanatory. Their respective methods loop through the objects and display them. If the user has none, a message is displayed to encourage the user to add events, performers, and friends.

Use the main menu described in User Layout to navigate to the user's calendar and lists of friends, events, etc.

Someone viewing a friend's profile can perform most of the above actions plus one more. This is shown in Figure 4.

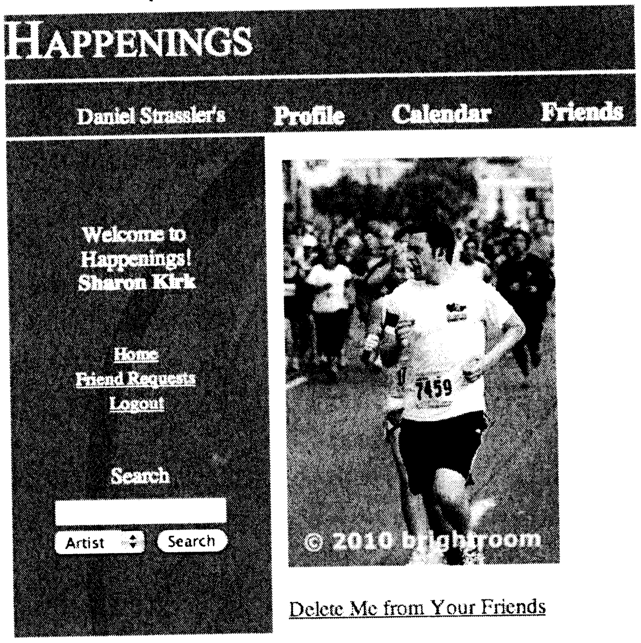


Figure 4: Friend's profile

Figure 4 demonstrates that the logged in user is Sharon Kirk viewing the user profile for Daniel Strassler.

The available actions are:

Delete me from your friends invokes `delete_friend` which sets variables `friend` and `user_record` equal to the respective user's IDs. `user_record.friends.delete(friend)` removes the friend and then the user is redirected to his or her own profile.

Comment on friend's profile is implemented the same way for friends as described for someone viewing his or her own profile.

Read friend's comments as above.

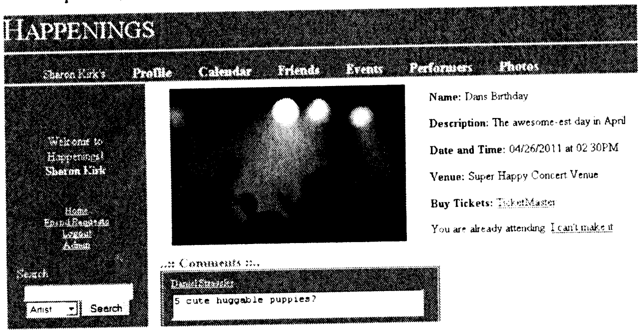
List all of friend's events, performers, or friends is also handled the same way for friends as it is handled for someone viewing his or her own profile.

Use the main menu to navigate to friend's calendar and lists of friends, events, etc.

Finally, someone who is not a friend is limited to one action:

Add me to your friends calls a method that sends an email to the user notifying him or her that a friend request is pending. This was achieved using ActionMailer, which is described in the Technical Details section.

Event
The Event page contains an event picture, commenting area, event information, links for indicating intent to attend or miss the event, and a link to the ticket vendor (if applicable). Event information includes the event name, description, date and time, and venue as shown in Figure 5.



Name: Dans Birthday

Description: The awesome-est day in April

Date and Time: 04/26/2011 at 02:30PM

Venue: Super Happy Concert Venue

Buy Tickets: TicketMaster

You are already attending I can't make it

5 cute huggable puppies?

Figure 5: Event page

The first time an event is viewed, a call is made to Last.fm using REST. This mechanism is described in Technical Details. Details retrieved about the event, venue, and performer are placed in Happenings' database. If Last.fm does not have a picture for an event (which is common), a generic event picture is used. If the event information is in the events table already, the information is retrieved from there. The link to buy tickets does not come from Last.fm because they do not provide a ticketing source reliably.

The event page allows the user to indicate his or her intent to attend the event by clicking "I'll be there!". Doing so invokes the `add_event` method to update the user's calendar by adding the event and user ids to the `calendar_entries_users` table. The event page then displays "You are already attending" along with an option to click "I can't make it." Upon clicking that link, the `delete_event` method removes the event id from the aforementioned table.

Calendar
The Calendar view was developed using the FullCalendar plug-in, which comes with various CSS files to give it either an Outlook or Google Calendar look and feel. Happenings uses the latter. JavaScript renders the calendar and makes an Ajax call back to the `get_calendar_events` method in the User controller. That retrieves the user's events from the `calendar_entries_users` table to be rendered for that particular calendar page (i.e., for the chosen month). The FullCalendar JavaScript libraries take care of rendering. FullCalendar is described further in Technical Details. Figure 6 shows the Calendar page.

Clicking on the event shown on December 19 takes the user to the Event page for that event.

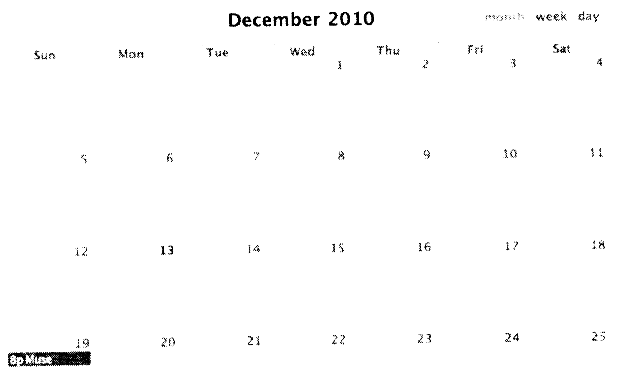


Figure 6: Calendar page

Performer
The Performer page is similar to the Event page. It displays the performer's picture, name, description, a commenting area, and links to view upcoming events or add the performer to the user's list of performers.

As for events, the first time a performer is searched the information is retrieved from Last.fm and stored in the Happenings database. The link to upcoming events takes the user to another page where they can view upcoming events and tour dates, also retrieved from Last.fm.

Selecting "Add performer" signals the `add_performer` method, which operates like `add_friend` except the user and performer ids are added to the `performers_users` table. The option to "Delete performer" appears if the user wishes to remove them from his or her list. Again, this is the same operation as `delete_friends` with the only difference being the type of object.

Photos
Users can add photos to their accounts. Photos can be commented on just like Profiles, Events, and Performers. Figure 7 shows the add photo screen:

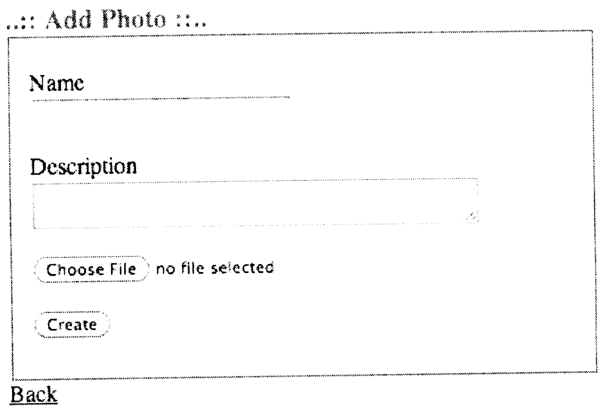


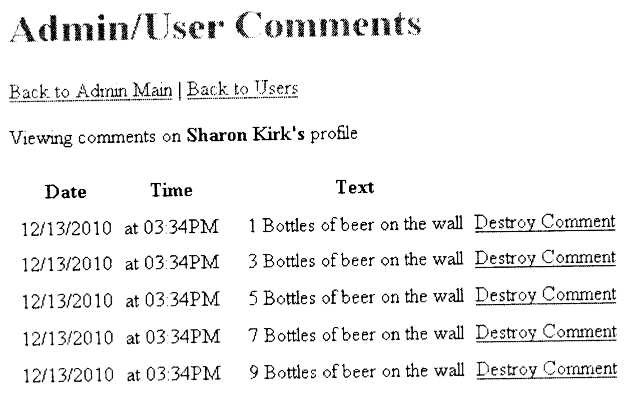
Figure 7: Add a photo

Search
Users can search for people or artists with the search bar, located in the sidebar of the User layout. If the search is for other users, SQL terms are used to return a list of users with names containing the string that was entered for the search.

Artist searches are handled with Last.fm API calls, discussed in Technical Details.

Admin
The Administrator functions were implemented by creating a module containing controllers for all of the resources administrators need to manage: Users, Events, and Performers. Each controller contains methods to `show`, `edit`, or `destroy` the respective resource. The main view contains links to lists of all users, events, and performers stored in the Happenings database.

Comments on Users, Events, and Performers can also be viewed by first selecting a resource. Each resource controller contains a `comments` method for listing the comments, and a self-explanatory `destroy_comment` method. Figure 8 shows an example of listing all comments on a user's profile.


Admin/User Comments

Back to Admin Main | Back to Users

Viewing comments on **Sharon Kirk's** profile

Date	Time	Text	
12/13/2010	at 03:34PM	1 Bottles of beer on the wall	Destroy Comment
12/13/2010	at 03:34PM	3 Bottles of beer on the wall	Destroy Comment
12/13/2010	at 03:34PM	5 Bottles of beer on the wall	Destroy Comment
12/13/2010	at 03:34PM	7 Bottles of beer on the wall	Destroy Comment
12/13/2010	at 03:34PM	9 Bottles of beer on the wall	Destroy Comment

Figure 8: Admin view of all comments on a User

TECHNICAL DETAILS
This section describes our development environment and the various technologies we needed to produce many of Happenings' features. Some functions were implemented with plug-ins, Ajax, Last.fm API, and XML parsing. We will also describe Happenings' user authentication.

Development Environment
Happenings was developed using NetBeans IDE version 6.9.1 [4]. It includes a Ruby on Rails project manager, debugger, and gems installation manager. We used Google Project Hosting for revision control [6].

We used Ruby version 1.8.6, Rails version 2.2.2, Mongrel server version 1.1.5, and SQLite3 version 1.3.1.

Plug-ins
We used various plug-ins including Paperclip and ImageMagick, ActionMailer, and FullCalendar.

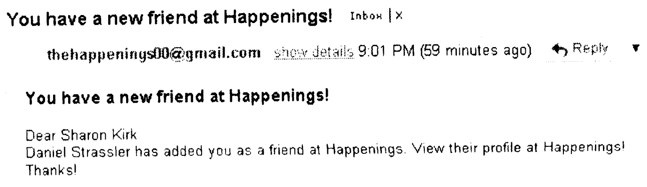
Paperclip and ImageMagick
Paperclip abstracts away from manually handling photos in our code [10]. In order to use it, each photo in the Photos table has a "Paperclip attachment". We added various columns to the `photos` table, which can be seen in the `add_image_column_to_photo.rb` database migration, that are specifically used by Paperclip. Paperclip is then able to "attach" references to three differently sized photos (whole, mid, and thumb) to the Photo object in the database. The photos themselves are stored inside the Public/System/Images folder on the Ruby application server. Paperclip handles making the references to the photos themselves.

ImageMagick is image-processing software with which the Paperclip plugin interacts for the rendering and resizing of photos [7]. It uses its own Ruby gem called 'rmagick.' We do not interact with 'rmagick' directly as it is only used by the 'paperclip' gem.

ActionMailer
Friend request emails are sent using ActionMailer. It is described as "a Ruby framework" for emails [1]. We chose to implement the option to send email based on SMTP, and use the Gmail public SMTP server to do it. This required an additional plugin called 'tlsmail' to handle the TLS authentication between our application and the Gmail server.

The configuration for ActionMailer is documented in `environment.rb`. It instructs ActionMailer to contact the Gmail SMTP server at smtp.gmail.com on port 587 (secure SMTP port) using the Google Mail account name theHappenings00@gmail.com.

We then used the Rails script/generate to generate a mailer called FriendMailer, that exists as a model called `friend_mailer.rb` in our project. It defines two methods, `friend_request` and `friend_added`, each representing a particular email to be sent out. These methods can then be called from the controller to send either of the email types. The friend request email notifies a user that he or she has a friend request from another user. That is shown in Figure 9.


You have a new friend at Happenings! Inbox | x

thehappenings00@gmail.com show details 9:01 PM (59 minutes ago) Reply

You have a new friend at Happenings!

Dear Sharon Kirk
Daniel Strassler has added you as a friend at Happenings. View their profile at Happenings!
Thanks!

Figure 9: Friend request email

The friend added email notifies the friend requestor that the request has been answered. Each email type is linked to its own view in the Views/friend_mailer folder. This is the template that is rendered when the email is actually sent.

FullCalendar
FullCalendar uses JavaScript libraries to handle Ajax calls for producing calendars [2]. The Calendar view contains the JavaScript calls to render the calendar. Every time the calendar is rendered, it makes an Ajax call back to `get_calendar_events` in the User controller. The database feeds in the list of events to be rendered for that particular page on the calendar, and the JavaScript takes care of rendering.

Ajax
Ajax was used in two components of the application. Both use it in slightly different ways. The first was for displaying comments. Displaying comments is handled inside the _comment_form.html.erb partial template in the user controller. To make the Ajax call, the ruby 'form_remote_tag' helper method is utilized. This method takes the controller action to be called, parameters to be passed in, and the 'div' container to be updated when html is returned by the call. In this case, the partial templates for the last 4 comments added on that entity that make up the comment list is returned.

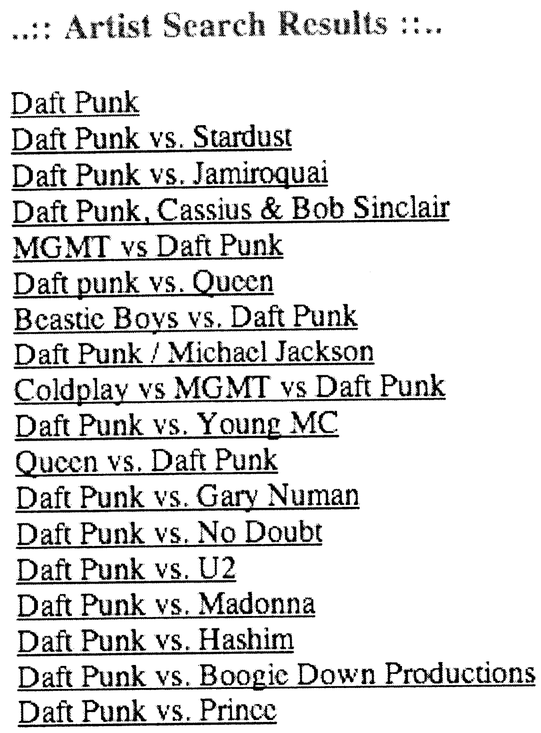
Ajax is also used for displaying events on the calendar. However, in this case the Ajax call was made from inside the javascript code used for displaying the calendar, and is handled by a jQuery call handled by the fullCalendar library. Every time the calendar is changed (i.e. when a user moves from one day, month, or week, to another), the fullCalendar library automatically makes the call to a supplied 'url' which is 'user/get_calendar_events'. Inside the Ruby method, we grab the events to display on the calendar, and assign them to parameters that can be handled by fullCalendar. The events are returned using the 'render' helper in JSON format.

Last.fm API
Last.fm provides a full array of API calls you can make for multiple items such as events, venues, and artists (named Performers in our application). These calls are made using the net/http gem and REST [5] to gather specific information. The API is fully documented and publically available [3].

The methods used in the application are `artist.search` for searching for artists. The results are limited to 25 so they fit on one page. When one of the artists is selected, the `artist.getinfo` method is called. When a person selects "Get upcoming events" on a Performer page, `artist.getevents` is used to get the upcoming events information. If a user selects an event from this listing and it is the first time the event has been selected, `event.getinfo` is used. All of these methods return an XML formatted response. XML parsing is discussed in the next section.

XML Parsing
The XML parsing is done using the 'REXML' gem [5]. The XML feeds returned by the various methods used to access the Last.fm API are well-documented again at the API page [3]. The first step of parsing the XML is converting the XML that is the net/http response object into an XML document.

Once it is a document, you can traverse the elements using various methods. It is extremely important to know the structure of XML documents or else it is very difficult to get the information that is needed from the document. We used the `get_elements` method in a loop to grab multiple reoccurrences of an attribute. This method grabs the attribute tags and text contained within it. We used `Variable.text` to get the text only. Another way is the `get_text` method to get only the text. Figure 10 shows the result of an artist search.


..:: Artist Search Results ::..

Daft Punk
Daft Punk vs. Stardust
Daft Punk vs. Jamiroquai
Daft Punk, Cassius & Bob Sinclair
MGMT vs Daft Punk
Daft punk vs. Queen
Beastie Boys vs. Daft Punk
Daft Punk / Michael Jackson
Coldplay vs MGMT vs Daft Punk
Daft Punk vs. Young MC
Queen vs. Daft Punk
Daft Punk vs. Gary Numan
Daft Punk vs. No Doubt
Daft Punk vs. U2
Daft Punk vs. Madonna
Daft Punk vs. Hashim
Daft Punk vs. Boogie Down Productions
Daft Punk vs. Prince

Figure 10: Search result for artist Daft Punk

Authentication
Sessions were used to hold state information about users who are logged in. Many methods start with defining `user_id = session[:user_id]`. That `user_id` is then compared with database tables to find relationships [8].

It would be insecure to store unencrypted passwords in our database. Passwords are hashed with two words and converted to a forty character string of hex digits [8].

TESTING

This section presents some examples of unit and functional tests that were written for Happenings.

Unit Tests

Most of the models contain a variety of validations. The User model contains the most. Examples are `validates_uniqueness_of :email` and `validate :password_non_blank`.

Many "Invalid with no attributes" tests were written for objects like Comments, Events, Performers.

Functional Tests

The User and Login controllers were tested with functional tests. Examples are tests to see if all pages display when given a session `user_id` and "No profile displayed without login."

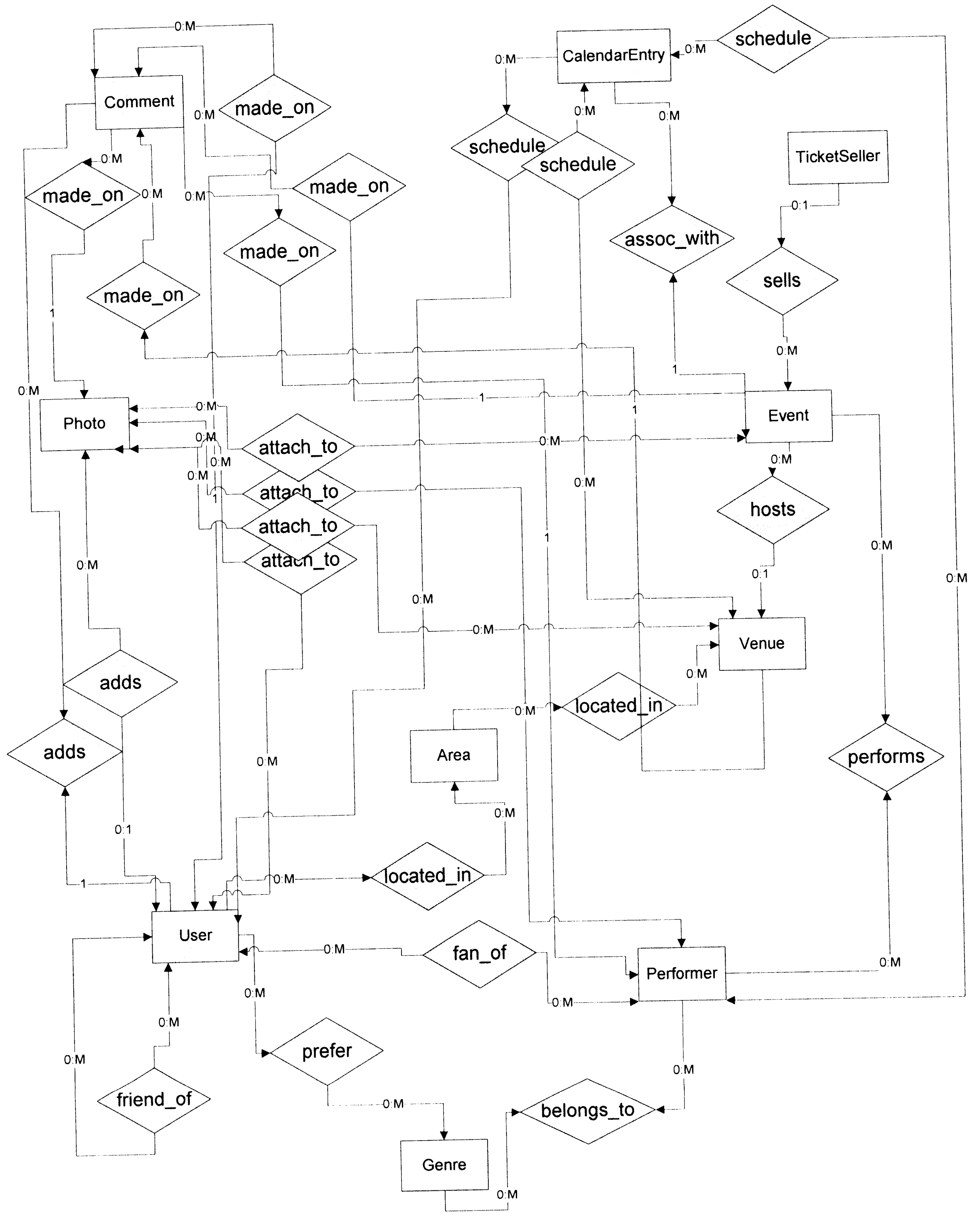
CONCLUSION

Happenings would be a very dull website without all the bells and whistles that are available through the use of plug-ins and open APIs. We demonstrated how we took features from Last.fm's Events section and combined those with elements of Facebook's friend interaction design. The result is a social networking mash-up designed to please music lovers and concert-goers.

REFERENCES

1. ActionMailer. Available at http://am.rubyonrails.org/.
2. FullCalendar. Available at http://arshaw.com/fullcalendar/.
3. Last.fm Web Services. Available at http://www.last.fm/api/.
4. NetBeans IDE 6.9 Features. Available at http://netbeans.org/features/ruby/index.html.
5. Parse XML Using Ruby. Available at http://developer.yahoo.com/ruby/ruby-xml.html/.
6. Project Hosting on Google Code. Available at http://code.google.com/hosting/.
7. RMagick Download Page. Available at http://rmagick.rubyforge.org/.
8. Ruby, S., Thomas, D., and Hansson, D.H. *Agile Web Development with Rails.* 3rd Ed. The Pragmatic Programmers LLC, 2009.
9. Ruby on Rails. Available at http://rubyonrails.org/.
10. Thoughtbot. Paperclip. Available at https://github.com/thoughtbot/paperclip/.

Entity-Relationship Diagram for Happenings Database


Comment
made_on
made_on
made_on
made_on
made_on
CalendarEntry
schedule
schedule
schedule
assoc_with
TicketSeller
sells
Event
hosts
Venue
Photo
attach_to
attach_to
attach_to
attach_to
adds
adds
Area
located_in
located_in
performs
User
fan_of
Performer
prefer
friend_of
belongs_to
Genre
0:M
0:1
1