LINK 2.0 Final Report

Thunder Cowan
Kylee Franci
Brianna Lijewski
Joshua Schoep

CSCI 370
Client: Kyle Castro, Admissions Officer
Colorado School of Mines

06/10/2020

Introduction

Acceptance of Admission = Attendance to University

This simple math problem is more complicated than originally perceived. Universities across the world have been experiencing a lower attendance rate compared to those who accepted their offer of admissions. Colorado School of Mines is no stranger to this phenomenon. In fact, the reputation held by Mines further creates a barrier of intimidation for students prior to beginning the fall semester.

For many years, Kyle Castro, a staff member of the admissions department at Colorado School of Mines, has been cultivating a vision for a better way to bridge the gap between acceptance and attendance and better integrate students into the Mines community. His vision first came to life in the form of LINK 1.0. A team in the 2019 fall field session created this nexus of information as an avenue of connectivity striving to ease the commencement of students' time at Mines. Moving forward with LINK requires an improved user interface, additional content, backend reworking and continued communication with Kyle Castro.

With a goal of bridging the gap between acceptance and attendance, LINK provides students with useful information preparing them for the opening months of their college experience and attempting to further excite them about their coming years at Colorado School of Mines. This website holds a plethora of information ranging from financial aid advice to counseling details; from highlighted sports and clubs to residence life facts; and even specific checklists for incoming freshmen. Additionally, students can learn more about the wonderful city of Golden by perusing through an index of popular eateries, entertainment and things to do around the local area. We hope that by making this information accessible and compiling a perspective centered around the ideal student experience, any incoming student will continue to be enthusiastic about their admission leading up to move-in. Furthermore, we hope this excitement will equate to engagement, and hopefully, an increased attendance rate at the beginning of August.

Requirements

As stated above, our goal is to build upon LINK 1.0 by improving user configurability with cleaner, content heavy pages, adding in more modules to highlight what Mines has to offer, and link more resources to make this a homepage for new members to our community. This goal shifted our focus to usability, ease of access, and a simple flow to make incoming freshmen feel less “out-of-the-loop” in regards to coming to Mines.

Functional Requirements

We added new content to the existing system, and thus needed to focus on getting our user stories and modules out the door, rather than improving on already existing functionality. This included:

- New categories of favorites in the user page
- Spotlights on various organizations on the user page and home page
    - Embedded social media and general information on student-led organizations
- Filtering of campus policies and redirects to Mines resources

    - Students are battered with a lot of information the summer before they come to Mines; we needed to find a way to reduce this to what they need to know
- Calendar module listing campus events, filtered option to reduce clutter
- Information section on tracks and minors
    - Educate and inform new students unfamiliar with this idea to help them in the long term
- Update developer backend to be more modular and expandable
    - Use of the SQL database to update hard-coded sections
- Integration with Mines Multipass system, which uses the Shibboleth authentication protocol

Non-Functional Requirements

While most of this project did not contain non-functional requirements, we had one set of these that we needed to implement.

Our customer wanted us to look into integration within the Mines Multipass system. On top of actual implementation of the Mines single sign-on (SSO) protocol, this requirement introduced some new non-functional requirements:

- Running the production server through Apache with SSL certification
- Grant client tokens to the Link server for use with Mines SSO
- Security integrations
    - HTTP forwarding to the secure site
    - Allowing external callbacks from our production website
- Admin Interfacing
    - Ability to update pages with limited technical skill and time
    - Access to backend user information including favorited clubs, activities and majors

System Architecture

As our project is a website, a lot of our design had to be done during the course of development and had to be modifiable. Architecture and planning could be done in advance, however, for the backend code and database structure. Ruby also has a strict structure that forces the frontend to be MVC and the backend to be REST. This alleviated much of our design of the frontend.

Database Schema

We made a number of improvements to the database system specifically. The system previously relied on hardcoded seeding elements and pulled from multiple seeding CSV files which contained largely the same information to populate two very information-heavy parts of the website. Through our development, we were able to refactor the database to follow the correct principles of database management and eliminate the mess of CSV seeding files that were in place originally. Figure 1 and Figure 2 (shown below) demonstrate the updates on the database schemas further illustrating the site-wide changes.

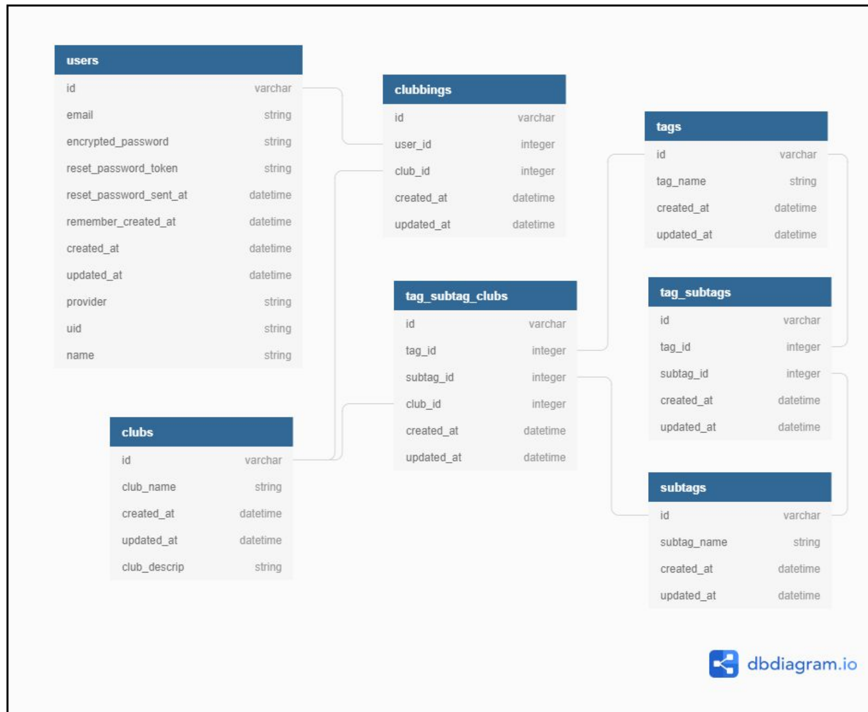

Figure 1: Original Database ERD

Figure 2: Current Working State ERD

We also changed the method used to seed the database itself, as the CSV files were not friendly to work with, and hardcoded elements that were not open to expansion in the future.

```
# Tag.create([
    # {tag_name: 'Academic'},
    # {tag_name: 'Advocacy'},
    # {tag_name: 'Athletics'},
    # {tag_name: 'Business'},
    # {tag_name: 'Creative'},
    # {tag_name: 'Cultural'},
    # {tag_name: 'Greek Chapter'},
    # {tag_name: 'Health'},
    # {tag_name: 'Leadership'},
```

Figure 3: Example of hardcoded seeding section

```
master_tag = CSV.read(Rails.root.join('lib', 'seeds', 'Master_Tags.csv'),

master_tag.each do |row|
    # Club and desc
    if !Club.exists?(club_name: row[0])
        Club.create(club_name: row[0], club_descrip: row[1])
    end
    # Tags
    if !Tag.exists?(tag_name: row[2])
        Tag.create(tag_name: row[2])
    end
    # Check for uncategorized (nil)
    if row[3] == nil
```

Figure 4: Example of new method of seeding

This change in methodology will allow for the next team that works on this project to implement a way for administrators to easily alter or directly add new content to the website. At a base level, the navigation of the seeding files and information has been made easily expandable.

Production Structure

In order for our intended audience to use this website, our production level server needs to be outside the protection of the Mines firewall, so we needed to highlight security considerations to protect our users. We hold our Rails production server behind an Apache reverse proxy, with Rails being served via Passenger. Passenger is an Apache library that integrates with full-stack web frameworks like Rails or Django. It knows how to bridge the gap between these apps and Apache, and can restart Ruby as needed to minimize downtime.

After we got our SSL certificates and keys, we turned the current HTTP Apache server to run SSL and made a request to open it from the Mines firewall. SSL encrypts user requests and increases the security of our site, allowing us to handle authentication safely and open our site to the world.

Figure 5 and 6 illustrate systems models for our development and production level servers, respectively.

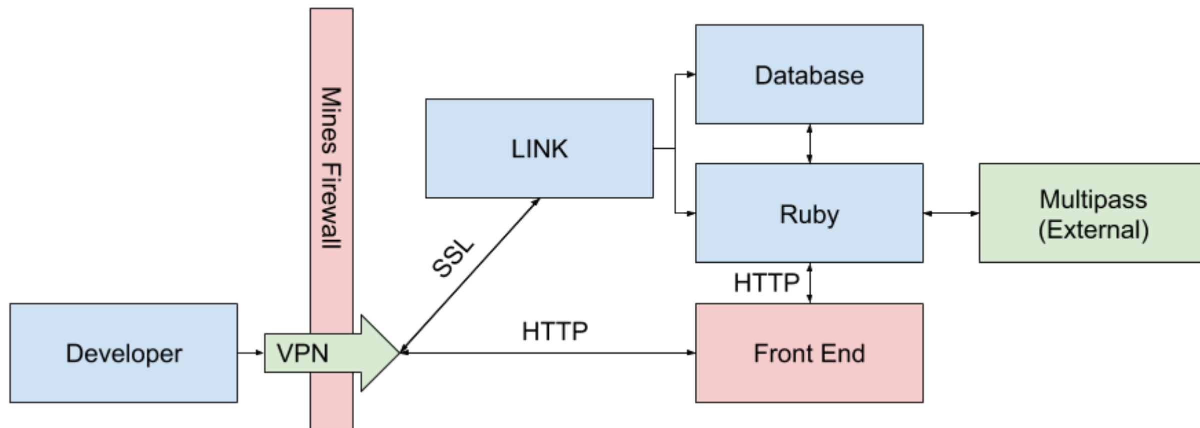

Figure 5: Development server, for testing on the Link server

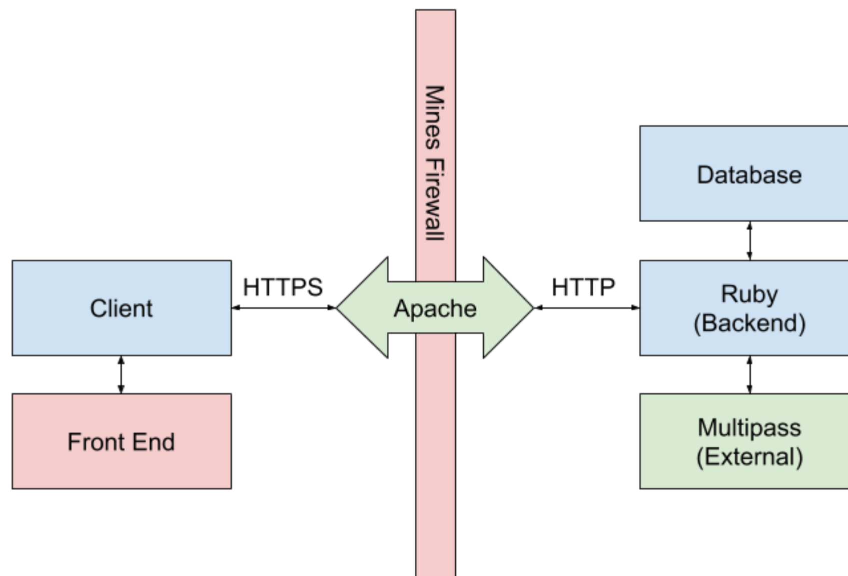

Figure 6: Production level server (Apache serving as reverse proxy)

Technical Design

Unchanged:

The original LINK team did an excellent job of maintaining the website. Here, we will give a short summary of their technical components before getting into more detail about what *we* did. In summary, the original specifications of the LINK website are:

- Ruby (2.6) on Rails (v6) web application served from a red hat Linux box within the Mines network
- Apache reverse proxy provides security and can boot up the Rails server via the Passenger library

- SQLite database started and managed by the Rails framework, with seed provided via CSV
- Multiple front-end javascript libraries
    - Coffeescript
    - Bootstrap CSS and SASS libraries
    - JQuery
    - Minimal Typescript
- SASS framework for stylesheets -- including animations
- Ruby Spring for improved caching and startup time
- Google OAuth2 with OpenID Connect (we removed this for Multipass integration)

Multipass:

Multipass is handled by the Shibboleth single-sign-on protocol, which is a stateful authentication protocol managed within the Mines network. For our component of the integration, we had to fill in a few requirements of the Shibboleth protocol.

First, Shibboleth requires SSO tokens to be handled and sent over a secure connection. We fulfill this by requiring all connections to LINK to go through Apache via SSL. We were granted an SSL certificate via Let's Encrypt by the Mines ITS team, which needs to be refreshed every three months. With user data encrypted, we can be sure that the connection is kept secure and user data is not compromised over our website in transit.

Second, we require implementation of the Shibboleth client protocol. From inside the Mines firewall, we needed to create routes and callbacks over which the Mines shibboleth server can connect users and give us confirmation that the user we are communicating with is a valid user.

Once signed in, a user is granted an access token with a timeout stored in their browser's local storage which will be sent with all requests to LINK. This allows them to access protected resources at /users/*, and allows the server to get the correct resources for the correct user ID.

Clubs and Organizations:

One of our most significant improvements was the user interaction with clubs and organizations. Rails provides a complete routing configuration for database resources which we used to handle club data.

HTTP GET requests were implemented at /clubs_orgs/{id} for each club to have its own page within our site, which displays all the info we have on clubs.

We added POST and PUT routes to /clubs_orgs/{id} to allow updates and additions to the clubs and organizations. On top of this, we created forms behind password-protected pages to allow the site's manager, Kyle, to edit and insert clubs without needing any technical background. This allows for additions after the initial website's seeding is completed, without having to manually edit the seed CSV files.

A special GET request to /clubs_orgs/spotlight gives a spotlight of the 5 most recently updated clubs and organizations for use on our homepage and profile pages. This route can also be embedded in other GET requests to put anywhere we would like to.

Database Handling:
The database schema and information stored is the skeleton of our project. Since we are showcasing such a large amount of information, and pulling from so many different URLs both on the Mines network and outside of it, keeping a well-managed database was key to producing an organized website.

We began this project by upgrading the backend portion of the website so it was more readable and expandable. The LINK 1.0 team did a great job defining a project vision and producing a working proof of idea, but there were a few things, like with any project, that were done in haste or done with the purpose of providing a direct deliverable. This led to a large amount of hard coded elements, and conflicting dependencies that resulted in an extravagant amount of data. For this reason, we placed an emphasis on improving this structure and making it more modular so that the next team working on LINK would be able to develop faster and implement their own ideas. We accomplished this goal through a rework of the database relations, and the elimination of hard-coded information in the website framework. Below is the structure that our database currently uses when handling the club and organization access.

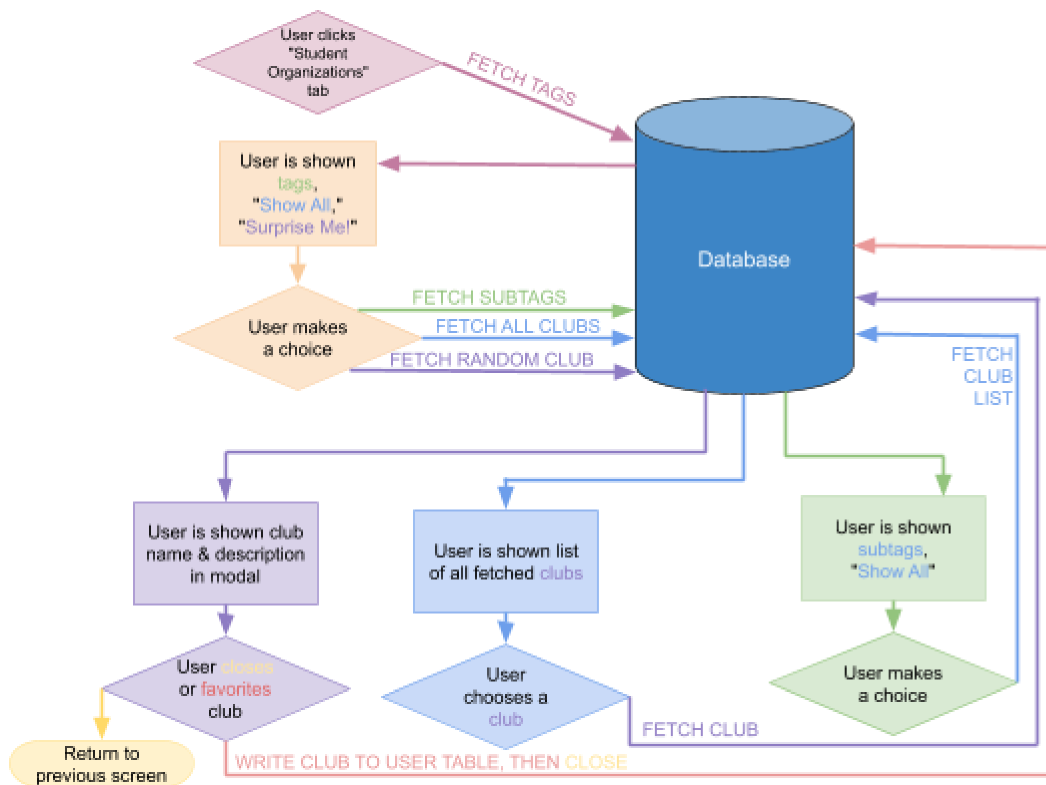

Figure 7: Database Structure from LINK 1.0

We also completely overhauled the method used to seed our database, as the original design relied on hardcoded elements and seven separate CSV files which had to be updated manually when a change was necessary. We were able to reduce the amount of CSV files which needed to be scanned down to one per joined relation in the database. An example of how this was done is provided below.

```
master_tag.each do |row|
    # Club and desc
    if !Club.exists?(club_name: row[0])
        Club.create(club_name: row[0], club_descrip: row[1])
    end
    # Tags
    if !Tag.exists?(tag_name: row[2])
        Tag.create(tag_name: row[2])
    end
    # Check for uncategorized (nil)
    if row[3] == nil
        TagSubtagClub.create(tag_id: Tag.where(
            tag_name: row[2]).pluck(:id)[0],
            subtag_id: nil,
            club_id: Club.where(club_name: row[0]).pluck(:id)[0]
        )
        # We can proceed to next row from here
        next
    end

    # Start at index 3 and loop over rest, makes totally expandable subtag structure
    row.drop(3).each do |sub_name|
        # Check if entry is nil
        if sub_name != nil
            # Need to make subtag
            if !Subtag.exists?(subtag_name: sub_name)
                Subtag.create(subtag_name: sub_name)
            end
            # We need to make tag subtag and tsc for non nil subtags
            if !TagSubtag.exists?(
                tag_id: Tag.where(tag_name: row[2]).pluck(:id)[0],
                subtag_id: Subtag.where(subtag_name: sub_name).pluck(:id)[0]
            )
                TagSubtag.create(
                    tag_id: Tag.where(tag_name: row[2]).pluck(:id)[0],
                    subtag_id: Subtag.where(subtag_name: sub_name).pluck(:id)[0]
                )
            end
            TagSubtagClub.create(
                tag_id: Tag.where(tag_name: row[2]).pluck(:id)[0],
                subtag_id: Subtag.where(subtag_name: sub_name).pluck(:id)[0],
                club_id: Club.where(club_name: row[0]).pluck(:id)[0]
            )
        end
    end
end
```

Figure 8: Seeding the database from a CSV

Quality Assurance

While software quality is always important, it becomes absolutely integral in a user-facing website such as LINK. User experience and user interface (UX and UI) are complex aspects of software engineering, and thus should have necessary redundancies to ensure their functionality over both the common case and edge case. We hope to use code reviews, acceptance tests via our customers, and strong deployment testing on all our additions to the site to ensure that the LINK website is secure, user-friendly, and functional.

- Unit Testing:
  - While we were left with no unit or functionality tests from the previous group, we built on top of their core code and added our own set of unit tests. In a longer time period, we would want to go back and add coverage to the core LINK *1.0* code, but we wanted to focus on finishing our customer's requests first and foremost. Rails provides a full testing library, which we used to write our tests. It provided coverage for the full stack, and even covered UI element testing.
- Code Reviews:
  - When each team member finished a user story, we had another teammate look over their code. This process helped to boost both product quality and team productivity as it allowed other teammates to be knowledgeable in more sections of the code, as well as making sure the logic is sound and the requirements were met for the user story that the teammate worked on. This additional check is also implemented through GitHub. If a member wanted to merge their branch with the master branch, they had to request the merge. This requires another teammate to look at the code and approve the code prior to allowing the merge onto the master branch.
- User Acceptance Testing:
  - During the last sprint of the project, we planned to have LINK 2.0 finished so that we could have our client look over the final product with us. This testing contributes to the quality of our product as it will allow our client to make sure that everything we discussed was implemented (or at the very least the high-priority items) and that the webpage works as he expected it to. This will also allow another pair of eyes to double-check that LINK 2.0 doesn't have any lingering problems that incoming freshmen might run into when using the website. This will be handled tomorrow during our final integration.
- Deployment Testing:
  - Deployment testing was a huge part of ensuring that our code was working properly. The Link website has numerous components and layers of software, so as each of us worked on different components it was essential to constantly deploy the code and actually test how everything worked together.
  - The deployment testing was also integral when we were implementing Multipass. There were various components of the original code that relied on Google's OIDC protocol, so all of those instances needed to be altered to fit with Shibboleth and the Multipass authentication. Since this could only be tested from within our

Mines server, we had to do on-site deployment testing to ensure the functionality of the system.

Results

Features Implemented:

- Clubs/Sports spotlight
- Upcoming events calendar
- Migrating "Things to do in Golden" into the database
- Upgraded profile page including favorite majors and things to do in Golden
- Added a tools page with important links and resources for incoming freshmen
    - Improvement from a FAQ's tab
- Multipass integration- allowing for a personalized user experience

Features We Were Unable to Implement

- Question board
- Scholarships and scholarship data
- Adding minors into the favorites page

Test Results

- Began writing unit tests within Ruby using the Rails built-in library
- Every new module is edge-tested by the team, and Rails has a built-in caching system and optimizations to improve performance as much as possible

Future Work

- A way to integrate the school's various event calendars (Mines Athletics, Mines Campus Events, etc.) automatically without someone having to add events through the command line
- Allowing the user's favorites to suggest additional clubs and things to do
- Once a messaging board has been implemented, building in an automoderator would make it so that someone doesn't have to constantly check the postings to make sure that no profanity has been used and/or no spam is being posted
- Implementation of user tests to ensure that freshmen are given adequate information with an appropriate user interface
    - Kyle Castro has provided some ideas as to how these user tests could be conducted in the future

Lessons Learned

- Ruby has a bunch of premade gems that are crowdsourced through Git and can do just about anything. One thing that we learned was that installing these gems and using them at a base level is pretty easy; however, once you need more than just the basic functionality that the gem's documentation goes over, the gems can be difficult to make work the way you want them to.
- Practicing good design habits will save time in the long run. Correcting bad habits, and removing existing hard coded elements is very involved. Learning Rails and Ruby was a

major learning curve at the beginning of the project. Ruby is a system that tries to provide help during the coding process which can be difficult for new Ruby users to understand. After an intense learning curve in the beginning of the five weeks, the team was more prepared to deal with the ins and outs of Ruby on Rails.

- Architecture and design is important in a project such as this, especially where there is already development done by others and we have to add in more features.
- Communication between teammates and the customer is invaluable, and design meetings are necessary to get everyone on the same page.
- The behind the scenes work of Multipass is very involved. This took a significant amount of learning and stepping out of our comfort zone to be able to implement everything involved.
- Relying on others can be difficult when it comes to ensuring you keep on schedule with your project. After numerous emails back and forth with ITS, we struggled with simple tasks like running the website over a VPN. The five-week intensive timeline made this roadblock a little more challenging for the team.

Appendix

How to run LINK Locally

Uses Ruby version **2.6.4**

Database creation

just be sure to seed the database:

```
rake db:seed
```

Deployment instructions

ssh into the server:

```
ssh username@link.mines.edu
```

switch to the super user:

```
sudo su
```

switch to the user which maintains the git repository (link user):

```
sudo -u link-user -H bash -l
```

change to the directory with the git repository:

```
cd /var/www/csmlink/code/
```

pull from git (note: you can use branches if you want, but I'd recommend using master for deployment):

```
git pull
```

run bundle install to get any new gems:

```
bundle install
```

set up app for production (precompile assets, perform new migrations, tell rails you want to compile the production app):

```
bundle exec rake assets:precompile db:migrate
RAILS_ENV=production
```

leave link-user and go back to root:

```
exit
```

restart the server

```
sudo apachectl restart
```

The server should now be updated!

Note: If javascript does not seem to be working when on production, ensure that the ***require_tree*** *line is at the bottom of the* ***application.js*** *file*