Matthew Prosser
Caley Shem-Crumrine
Will Wettersten
Gregory Ziegan

Final Report -- Snipit

1. Overview

As it stands, users of web often find themselves unable to express thoughts and feelings in a natural way through social media. For instance, it is hard for a user to come across as sarcastic or witty when they are constrained by only having text comments. Youtube embedding is a welcome help to this issue, where users can simply post a youtube link as a comment or post and have a video embedded in the webpage. The issue with this is that it limits the user by making them post an entire video. If a user wanted to comment on a friend's status with a simple quip of a few seconds from a video, they would have to post the entire video and let their friend just figure it out.

Our project is built to deal with this exact use case. With Snipit, users can take a Youtube video and trim it to a maximum of five seconds. Snipit then hosts the video and can serve it to an embeddable player. This enables them to post just the portion, or snippet, of the video that they want, enabling a much more natural flow or dialogue in social media.

2. Application Specifications

In accordance with recent revelations in design philosophy, Snipit will be comprised of

just four web pages: the splash page, the create Snipit page, the video page, and the user profile page.

The splash page is at the heart of Snipit. It is the first thing a user will see when they visit the site. The splash page will serve two purposes. Firstly, it will be a login page. Users on the splash page will notice a bar at the top of the page that has the option of logging in or signing up for a Snipit account. When hovered over, these menus will present dropdowns that contain forms that will make ajax posts to Snipit. This is done to avert the need for an entire web page just for logging in. With a dropdown form, we can still do all of the necessary things we need for logging in and signing up, namely form validation. For example, when a user selects “sign up”, they will be presented with a form containing a username, password, and repeat password field. When they enter their new username, the page will send an ajax request to the server checking to see if a user by that username already exists. If it does, that part of the form will turn red, indicating that the user must find a new, unused username before being allowed to continue registering an account. Similar checks will be implemented for things like making sure that the password and repeat password field match.

If a user on the splash page is signed in, instead of having two options in the top bar for “Sign in” and “Sign up”, they will be presented with a message “Hello, [username]”. When they hover over this message, they will be presented with a dropdown of options including “view profile”, “create snipit”, and “sign out”. If they select “view profile”, they will be taken to the page for their profile. If they select “create snipit”, they will be taken

to the create snipit page where they can cut a snippet out of a youtube video. Finally, if they select sign out, the page will be refreshed and the user will see the logged out version of the page with the “Sign in” and “Sign up” options.

In the body of the splash page will be a text search box similar to Google’s search box. When a user types a string into the search box, a javascript function waits for a pause after the typing (indicating that the user has entered their entire search) and then fires an ajax request to the server to retrieve results for that search. This results will be videos, and the videos will be listed under the search box in a table. Each row in the table will be a thumbnail of a video with the video title, description, and poster. In this way, a user can essentially query Snipit without refreshing their page. When the user clicks on one of the rows in the table, they will be taken to the video page for that video.

The create a snipit page will be where users can actually make their 5-second video snippets (aka a “snipit”). This page will have a text box at the top where users can paste a YouTube URL. When a user puts text in this text box, Snipit will attempt to fetch the video at that url and, if successful, display the video in the Snipit editor. At the top of the snipit editor is a text box representing the name of the snipit. It will default to “new snipit.” Directly beneath this text box will be the video editor, which will show the full length youtube video and have a slider directly beneath the youtube video. When a user drags the end of the sniper, they set the in and out points of what will be the snippet. Beneath the slider will be an empty text box where the user can type a short description of the snippet. Once the user has selected bounds that satisfy the constraint of having

the snippet be less than 5 seconds in duration, a fogged out “create” button next to the the slider will become unfogged. When the user clicks this button, the server will invoke a java program that will download the youtube video, trim it to a given length, and then store it somewhere on the server’s file system. Then, the server will create a row in the videos table with any data about that video (id, name, description...) and a location in the file system where that video can be found. This table will be used later to serve videos and video previews.

The video page will select a snipit through a GET variable passed in the URL. Each snipit’s unique URL ID will serve to make each snipit bookmarkable. If no ID is specified, or if the ID is not found in the database, the user will be redirected to the splash page and shown an error message.

The video page will include a flash player that will play the snipit automatically. Below the player, the page will display the snipit’s tags, view count, number of likes/dislikes, and comments. Only signed-in users will be allowed to tag, comment, or like/dislike a video. When a user views a video page, they will have the option of changing their like/dislike or removing their comments or tags. Snipit tags and likes/dislikes will be made using AJAX calls to query the database, removing the need to refresh the page. The video’s view count will increment every time the video is watched, regardless of whether the watcher is signed in.

The user profile page will display all user information, including the user’s email address, favorite videos, and videos created by that user. Anyone can view a user’s

profile, but only that user may edit their email address or remove their created snipits or favorites. Removing created snipits will delete the video from the entire site, while removing a favorite will simply remove it from that user's favorites list.

Users can log in or out from any page through links on the top bar of each page. When logged in, the top bar will show the user's username, which will link to the user's profile page.

3. Database Specifications

Our database will have the following entities: Users, Videos, Tags, Playlists and Groups. The User entity will be used to store information about someone who signs up for an account on Snipit. Users will include a username, email address, and password. The username will need to be unique and thus it will be the primary key for the User entity. The Videos entity will be composed of a url link to the video, a start time, and a stop time. The primary key will be the url link since it is a unique reference. Tags are our last entity. The tag will be composed of a non-case-sensitive string which will be the primary key as well as the sole attribute.

Relationship tables in our database include: Views, Favorites, Comments, Likes/Dislikes, Friends, Members and Messages. Views will be a table that updates a single field: an integer that increments each time a video is viewed. A Views tuple would include the video url, the user who viewed it, and the number of total views. Favorites is similar to the Views relation except in that it keeps track of videos

"favorited" by a user. The tuple for Favorites is identical to that of Views except there is no Total Views attribute and the Viewed_by_User is renamed Favorited_by_User. This relationship will also update a sort of "News Feed" for the User who favorited a video and is accessible from that User's News Feed or Profile. Comments is, again, similar to Views except its Total Views attribute is replaced by a String representation of the comment submitted by a specific User. Likes/Dislikes is practically the same Relationship as Comments except there are two attributes replacing the String entered by the user. One is a likes column of booleans and the other is a dislikes column of boolean values.

Some of the typical queries users would like to use on Snipit would be: getting all the comments posted about a video, search results for the most popular video on Snipit (or in recent weeks, months), a history of likes/dislikes for a video, a user history (similar to Facebook's “News Feed”), the creation of new Users, altering the status of a User's favorited videos, reset to a password given a confirmation key. These are a few of the most important and popular queries Snipit will be using on a daily basis. The database will not be handling immense amounts of users and stored videos, so retrieval time will not be optimized in our queries.

Some constraints include: comments, likes, and dislikes need to be associated with at least 1 video, a video must be tied (whoever uploaded it to Snipit) to a User, and Favorites and Views must be associated with Videos in a many to many relationship.

Triggers include a pre-made set of queries that delete certain types of information from

our database (user accounts, video links, comments) without causing dangling pointer in our database.

4. ER Data Model Design

Our database design consists of the entities Video, User, Tag, Playlist, and Group. A video represents a Snipit video that has been saved somewhere on the filesystem of the server. All entries in any database table will have an integer ‘id’ which is a primary key, as we all createdAt and updatedAt column which are both MySQL datetimes. Videos will have a name which is a string, a duration which is an integer and a path which is a string. The path will just be the unix-style absolute path where the video can be found. Users will have a name which is a string, a password which is an encrypted binary, and a nullable age which is an integer. Tags will simply have a field called ‘word’ which is a string of what the tag is. For instance, if I tag a video with #bodybuilding, Snipit will create a tag relation pointing to a Tag row with ‘word’ = ‘bodybuilding’. Playlists will have a name which is a string and a creator id which is an integer pointing to the User who created the playlist. Finally, Groups will have a name which is a string and a creator which is an integer pointing to the User that created the group.

Our database will also have the relations Belongs to, View, Favorite, Comment, List/Dislike, Member of, and Message. Belongs to is a many to many relationship between videos and playlists. This makes sense because a playlist is essentially a list of videos, and videos can be in more than one playlist. View will describe a user viewing a video. Each time such a view occurs a row will be inserted in the View table that will reference the user who viewed and the video that was viewed. This makes view a many to many relationship. Favorite describes a

user adding a video to their list of favorites. This is also a many to many relationship, because each time a user adds a video to their favorites a row is inserted in Favorite that points to the user and the video. Comment describes a user commenting on a video. Similarly to Youtube, Snipit will have a chronological list of comments appear beneath videos so that our users can banter endlessly about politics. Comment will be many to many, have pointers to user and video, and have an additional field which is a string which contains the actual comment. Like Dislike will essentially be a binary piece of data, either a like or a dislike, relating a user to a video. It is a many to many relationship with the constraint that a user can only like or dislike a video once. So the rows in Like/Dislike are unique on user id and video id. Member describes a user joining a group. It will also be many to many, and will have foreign keys pointing to user and group. Finally, Message will denote a message sent from one user to another. It will be many to many and have two pointers to user and an additional message field which will be a string.

5. Relational Model Design

```
---------------------
--- ENTITIES ---
---------------------

CREATE TABLE IF NOT EXISTS `video` (
        `id` INTEGER NOT NULL auto_increment ,
        `name` VARCHAR(255) NOT NULL,
        `duration` INTEGER NOT NULL,
        `path` VARCHAR(255) NOT NULL,
        `creator` INTEGER NOT NULL REFERENCES users(id),
        `createdAt` DATETIME NOT NULL,
        `updatedAt` DATETIME NOT NULL,
PRIMARY KEY (`id`)) ENGINE=InnoDB;

CREATE TABLE IF NOT EXISTS `user` (
        `id` INTEGER NOT NULL auto_increment ,
        `name` VARCHAR(255) NOT NULL,
        `username` VARCHAR(255) NOT NULL,
```

```
	`password` BLOB NOT NULL,
	`age` INTEGER,
	`createdAt` DATETIME NOT NULL,
	`updatedAt` DATETIME NOT NULL,
PRIMARY KEY (`id`)) ENGINE=InnoDB;

CREATE TABLE IF NOT EXISTS `tag` (
	`id` INTEGER NOT NULL auto_increment ,
	`word` VARCHAR(255) NOT NULL,
	`createdAt` DATETIME NOT NULL,
	`updatedAt` DATETIME NOT NULL,
PRIMARY KEY (`id`)) ENGINE=InnoDB;

CREATE TABLE IF NOT EXISTS `playlist` (
	`id` INTEGER NOT NULL auto_increment ,
	`name` VARCHAR(255) NOT NULL,
	`creator` INTEGER NOT NULL REFERENCES users(id),
	`createdAt` DATETIME NOT NULL,
	`updatedAt` DATETIME NOT NULL,
PRIMARY KEY (`id`)) ENGINE=InnoDB;

CREATE TABLE IF NOT EXISTS `group` (
	`id` INTEGER NOT NULL auto_increment ,
	`name` VARCHAR(255) NOT NULL,
	`creator` VARCHAR(255) NOT NULL REFERENCES users(id),
	`createdAt` DATETIME NOT NULL,
	`updatedAt` DATETIME NOT NULL,
PRIMARY KEY (`id`)) ENGINE=InnoDB;

----------------------
-- RELATIONS --
----------------------

CREATE TABLE IS NOT EXISTS `belongs_to` (
	`id` INTEGER NOT NULL auto_increment ,
	`video_id` INTEGER NOT NULL REFERENCES video(id),
	`playlist_id` INTEGER NOT NULL REFERENCES playlist(id),
	`createdAt` DATETIME NOT NULL,
	`updatedAt` DATETIME NOT NULL,
	UNIQUE (`video_id`, `playlist_id`),
PRIMARY KEY (`id`)) ENGINE=InnoDB;

CREATE TABLE IS NOT EXISTS `view` (
```

```
        `id` INTEGER NOT NULL auto_increment ,
        `video_id` INTEGER NOT NULL REFERENCES video(id),
        `user_id` INTEGER NOT NULL REFERENCES user(id),
        `createdAt` DATETIME NOT NULL,
        `updatedAt` DATETIME NOT NULL,
PRIMARY KEY (`id`)) ENGINE=InnoDB;

CREATE TABLE IS NOT EXISTS `favorite` (
        `id` INTEGER NOT NULL auto_increment ,
        `video_id` INTEGER NOT NULL REFERENCES video(id),
        `user_id` INTEGER NOT NULL REFERENCES user(id),
        `createdAt` DATETIME NOT NULL,
        `updatedAt` DATETIME NOT NULL,
        UNIQUE (`video_id`, `user_id`),
PRIMARY KEY (`id`)) ENGINE=InnoDB;

CREATE TABLE IS NOT EXISTS `comment` (
        `id` INTEGER NOT NULL auto_increment ,
        `video_id` INTEGER NOT NULL REFERENCES video(id),
        `user_id` INTEGER NOT NULL REFERENCES user(id),
        'comment' VARCHAR(2048) NOT NULL,
        `createdAt` DATETIME NOT NULL,
        `updatedAt` DATETIME NOT NULL,
PRIMARY KEY (`id`)) ENGINE=InnoDB;

CREATE TABLE IS NOT EXISTS `like_dislike` (
        `id` INTEGER NOT NULL auto_increment ,
        `video_id` INTEGER NOT NULL REFERENCES video(id),
        `user_id` INTEGER NOT NULL REFERENCES user(id),
        `like_dislike` VARCHAR(16) NOT NULL,
        `createdAt` DATETIME NOT NULL,
        `updatedAt` DATETIME NOT NULL,
        UNIQUE (`video_id`, `user_id`),
PRIMARY KEY (`id`)) ENGINE=InnoDB;

CREATE TABLE IS NOT EXISTS `member` (
        `id` INTEGER NOT NULL auto_increment ,
        `user_id` INTEGER NOT NULL REFERENCES user(id),
        `group_id` INTEGER NOT NULL REFERENCES group(id),
        `createdAt` DATETIME NOT NULL,
        `updatedAt` DATETIME NOT NULL,
        UNIQUE (`user_id`, `group_id`),
PRIMARY KEY (`id`)) ENGINE=InnoDB;
```

```
CREATE TABLE IS NOT EXISTS `message` (
        `id` INTEGER NOT NULL auto_increment ,
        `sender_id` INTEGER NOT NULL REFERENCES user(id),
        `recipient_id` INTEGER NOT NULL REFERENCES user(id),
        `message` VARCHAR(2048) NOT NULL,
        `createdAt` DATETIME NOT NULL,
        `updatedAt` DATETIME NOT NULL,
PRIMARY KEY (`id`)) ENGINE=InnoDB;
```

6. Creating Database

We downloaded and installed MySQL 5.529 on Ubuntu 12.04. The installation went flawlessly and our application was able to connect to MySQL through our connector library.

7.

5 Toughest queries are listed below in RA & TRC:

i. SQL:

```
select * from
  (select V.id as videoId, V.createdAt, U.id as userId, U.name,
     U.username,U.age, V.name as videoName, V.uploader as uploader
   from user_to_video_favorites F, videos V, users U
   where F.videoId = V.id and F.userId = U.id and U.id = :userid)  X,
  (select U.id, U.name as uploaderName
   from users U) Y
 where Y.id = X.uploader;
```

Tuple Relational Calculus:

$$\{t[10] \mid (\exists f)(\exists v)(\exists u)(F(f) \wedge V(v) \wedge U(u) \wedge f.videoId = v.id \wedge f.userId = u.id \rightarrow (\forall u2)(U(u2) \wedge u2.id = v.uploader \wedge t[1] = f.videoId \wedge t[2] = v.createdAt \wedge t[3] = f.userId \wedge t[4] = u.name \wedge t[5] = u.username \wedge t[6] = u.age \wedge t[7] = v.name \wedge t[8] = v.uploader \wedge t[9] = u2.id \wedge t[10]\ u2.username))\}$$

RA:

$$\pi_{*}\left(\sigma_{id = uploader}(\pi_{v.id\ as\ videoId,\ v.createdAt,\ u.id\ as\ userId,\ u.name}\left(\sigma_{f.videoId = v.id,\ f.userId = u.id,\ i.id = :userid}(video\ x\ user\_to\_video\_favorites\ x\ users)\right)\ x\ \pi_{u.id,\ u.name,\ uploaderName}(users)\right)\right)$$

ii. SQL:

```
SELECT distinct F.id as recipientId, F.name as recipientName, P.senderId,
P.senderName, P.message, P.createdAt
```

```
        FROM (SELECT Y.id as senderId, Y.name as senderName, X.recipient,
                X.message, X.createdAt
                FROM (SELECT m.message, m.sender, m.recipient, m.createdAt
                        FROM messages m, users u
                        WHERE u.id=m.sender OR u.id=m.recipient) X, users Y
                WHERE Y.id = X.sender) P, users F
        WHERE P.recipient= F.id
        ORDER BY createdAt
```

TRC:

RA:

iii. Find the 10 most popular videos in Snipit (popularity is defined as likes - dislikes).

SQL:

```
SELECT P.video AS vid, V.name, V.path,
(P.likes - P.dislikes) AS popularity
FROM videos V, (
      SELECT L.video,
      SUM(CASE WHEN L.likedislike = "like" THEN 1 ELSE 0 END) AS likes,
      SUM(CASE WHEN L.likedislike = "dislike" THEN 1 ELSE 0 END) AS dislikes
      FROM likedislikes L
      GROUP BY L.video
) P
WHERE P.video = V.id
GROUP BY P.video
ORDER BY popularity DESC
LIMIT 10
```

This query uses aggregate functions, and thus cannot be expressed in TRC or RA.

iv.

SQL:

```
select V.id, V.name, V.path
from likedislikes L, videos V
where L.likedislike = 'like' and L.user = :userid  and V.id = L.video
```

Tuple Relational Calculus:

$\{t[3] \mid (\exists v)(\exists l)(V(v) \wedge L(l) \wedge L.likedislike = \text{‘like’} \wedge L.user = :userid \wedge L.video = V.id \wedge t[1] = V.id \wedge t[2] = V.name \wedge t[3] = V.path)\}$

RA:

π id, name, path(σ likedislike = ‘like’ ^ user = :userid (L $\bowtie$ V))

v. Find the 10 most viewed videos in Snipit.

SQL:

```
select id, name, path from
   (select id, name, path, count from
      (select v.id, v.name, v.path, count(*) as count from
        user_to_video_views utv, videos v
        where v.id = utv.videoid
      group by v.id)
   order by count)
limit 10
```

Tuple Relational Calculus:

Relational Algebra:

8. The integrity constraints on this database by table are:

‘comments’

- ‘id’ is required
- ‘comment’ is required
- ‘user’ is required
- ‘video’ is required

‘groups’

- ‘id’ is required
- ‘name’ is required

‘likesdislikes’

- ‘id’ is required
- ‘likedislike’ is required
- ‘user is required
- ‘video’ is required

‘messages’

- ‘id’ is required
- ‘sender’ is required
- ‘recipient’ is required

‘playlists’

- ‘id’ is required
- ‘name’ is required
- ‘creator’ is required

‘tag_to_videos’
‘videoId’ is required
‘tagId’ is required

‘tags’
‘id’ is required
‘word’ is required

‘user_to_group’
‘groupId’ is required
‘userId’ is required

‘user_to_video_favorites’
‘videoId’ is required
‘userId’ is required

‘user_to_video_views’
‘videoId’ is required
‘userId’ is required

‘users’
‘id’ is required
‘name’ is required
‘username’ is required
‘password’ is required
‘age’ is required
‘email’ is required

‘video_to_playlists’
‘videoId’ is required
‘playlistId’ is required

‘video’
‘id’ is required
‘name’ is required
‘path’ is required

9.

When designing the database, we tried to eliminate as many dependencies as possible.

The only functional dependencies that exist in our database are the primary key’s determination

of the rest of the entity's attributes. Checking every one of our thirteen tables, we do not have any that are not in Boyce-Codd Normal Form. If we in the future choose to have entries in our database able to be modified by user input, our schema may need to face some decompositions.

Examples would include: users editing their comments and users altering their profiles (passwords, age, etc). An example of a dependency in our database can be taken from any one of our entities, since all of them have a similar single dependency. For comments, the attributes are id, comment, createdAt, updatedAt, user, video. Shortening these attributes to ICRUSV the functional dependency is I -> ICRUSV.

10.

Since our database is already in BCNF, we have little reason to alter our schema when queries and triggers will process efficiently with redundancies non-existent.

11.

We began using MySQL with no issue. Up-to-date versions were available from apt-get, so everyone was able to do a one line install. The only issue encountered was that to make a connection, login credentials were needed in the code. Because we were using git, we did not want to have strings of our personal passwords in the code, so we created a function that reads in credentials from a json file on a conf/ directory and then uses those credentials to create a database connection. The conf/ directory is not checked into source control, so nobody has to share their password.

12.

We used a variety of packages to make web development smoother, cleaner, and faster. Packages include Coffeecup, Coffee-script, and Sequelize. Coffeecup and Coffee-script were chosen to make our front-end code easier to read by compiling the code we type into javascript

code while we deal with less brackets, parentheses and such from javascript syntax. Sequelize is an ORM that was used to generate SQL queries without hard-coding SQL into our views. This makes our code easier to understand and quicker to write.

13, 14

See individual submissions.

15.

- The project turned out to be quite fun to program and resulted in a working application. Better yet, it is something I/we will use on a regular basis. We finished what we intended to get done and did so without getting on each others' nerves. Overall, the process was a great experience and we had excellent and receptive team members. Respect was given to all the members regardless of familiarity with coding projects and that is a godsend in CS classes.

16.

Our project is hosted on a remote server at the following address:
You can access it by entering that address in your address bar. If you would like to host the app on your own machine, simply download the zip file we have submitted, create an empty database named 'snipit', and use the package nodejs to run our project with the command:
$ node /filePathTo/app.js You can then go to localhost:8080 and view our project.

17.

User's Manual:

Most of the project will be mostly self-explanatory considering that our navigation is streamlined and will report errors to the user if something is done incorrectly. Regardless, there are 5 pages available to the user. The index (home) page where popular videos and newly snipped videos are displayed as thumbnails the user can click to view. The new page where the

user can create a profile to use with the site. The profile page displays user favorites, user playlists, and user uploads along with information about the user. The snip page is used to snip videos into 5 second and under chunks and the result is stored in our database. The video page is used to watch videos when a user wished to view a snipit video on the site.

How to Snip a Video

1. Click on the "Snip a Video" button on the navigation bar to get to the Snip page.
2. Copy and paste a YouTube URL into the top text field.
3. Specify start and end times and give your Snip a name.
4. Click submit. You will be taken to the new video's page, which will supply an error message if they video is not yet ready. If you see this message, refresh the page after a few seconds to view your video.
   Note: We would like to add a JavaScript progress bar to indicate how close the video is to being created, as well as to automatically load the video when it is done. However, as this is not a vital feature, it has not been our top priority.

18. Programmer's Manual:

I. **Server Requirements**

a. **Node.js**

This project uses Node.js version 0.10.3 with the following modules:

node-coffee-script 1.4.0
node-coffeecup 0.3.19
node-cookies 0.3.6
node-express 3.0.6
node-keygrip 0.2.2
node-mysql 2.0.0-alpha7
node-sequelize 1.6.0

b. **Video Player**

Snipit uses Flowplayer Flash to play its videos. Full documentation on Flowplayer can be found at http://flash.flowplayer.org/documentation/installation/index.html. When running Snipit from a local machine, you must ensure that Flash allows videos to be played from your hard drive. You can do this at http://www.macromedia.com/support/documentation/en/flashplayer/help/settings_manager04.html by adding the directory containing your videos to the list of trusted locations.

c. **MySQL**

The server expects MySQL 5.x.x with a database called snipit

d. **Java**

The video snipper requires Java 7.

II. **Functions**

Snipit has very few app-wide functions. All of them are contained in the util.js file.

a. **spawn** is a wrapper for the built-in Node.js child_process utility child.spawn.
b. **uuid** generates random Java-style UUIDs. It is used to generate file names for videos and thumbnails.
c. **getPopular** queries the database for the 10 most popular videos. It is called on the index page.
d. **getRecent** queries the database for the 10 most recently created videos. It is also called on the index page.

III. **Models**

We used models to represent our database entities. Each model is defined in its own file within the *models* folder, and the relationships between models are defined in dao.js.

Our models are User, Video, Tag, Playlist, Group, Message, Comment, and LikeDislike.

19.

Will be presented on May 8th.