Video Annotation for Choreographers
on the NB Platform

by Louis M. Lamia

S.B. Computer Science and Engineering, M.I.T. 2014

Submitted to the
Department of Electrical Engineering and Computer Science
In Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

September 2015

Copyright 2015 M.I.T. All rights reserved.

Author:

Department of Electrical Engineering and Computer Science
September 4, 2015

Certified by:

David Karger, Professor, Thesis Supervisor
September 4, 2015

Accepted by:

Dr. Christopher Terman, Chairman, Masters of Engineering Thesis Committee
Abstract
The NB Platform, originally developed to host discussion on class materials, has been adapted to suit the needs of dance choreographers. A Video Annotator was developed, and features requested by choreographers were added to help suit the platform to their needs. The updated annotator was then subjected to user testing to determine more ways in which software can be developed for the dance community and video annotation software can be improved.
# Table of Contents

Table of Contents .................................................................................................................. 3
Introduction ................................................................................................................................. 4

**Background** .......................................................................................................................... 5
  Previous Work ........................................................................................................................... 6
  A Note on Terminology ............................................................................................................ 8

**The Original NB Platform** .................................................................................................... 9
  Video Annotator ....................................................................................................................... 9
  Initial Choreographer Reactions ............................................................................................ 11

**Features For Choreographers** ............................................................................................. 12
  Comment Thread Duration ....................................................................................................... 12
  Tagging .................................................................................................................................... 14
    Email Notifications ................................................................................................................ 18
    Tag Privacy ............................................................................................................................ 19
  Section Titles ........................................................................................................................... 20
  Miscellaneous Tweaks ............................................................................................................. 22

**User Tests** ............................................................................................................................ 24
  First Round Methodology ........................................................................................................ 24
  First Round Tests .................................................................................................................... 25
  Final Round Methodology ....................................................................................................... 27
  Final Round Tests ................................................................................................................... 28

**Conclusions** ......................................................................................................................... 32
  Future Work ............................................................................................................................. 33

**Appendix A: NB Architecture** ............................................................................................ 35
  A Note on Terminology ............................................................................................................ 35
  Server and Client ..................................................................................................................... 35
  Views ...................................................................................................................................... 37

**Appendix B: Codebase Documentation** ............................................................................. 40

**Appendix C: Django Model Changes** .................................................................................. 69

Bibliography ............................................................................................................................... 71
**Introduction**

NB is a platform originally designed for sharing and collaborating on class materials. Instructors can post PDFs with readings or assignments, and students can annotate and collaborate on them. A more recent feature is a video annotator, which allows an instructor to link to a YouTube video, a lecture video for instance. Students can log in and annotate the video in the same way they can annotate PDF readings.

The video annotator was initially intended for video lectures and academic assignments, but it also holds a lot of potential for dance choreographers. The ability to annotate videos is a more natural interface for the sharing of videos and comments that might normally be done over email. In its initial incarnation, the tool allowed choreographers to provide comments and context together by placing annotations at specific time points and locations.

Choreographers have different needs from teachers. They want to address individual dancers and remind them about their comments, for instance. We tested the initial interface with some dance choreographers and collected feedback on the sorts of features that would help make NB a useful platform for them. We then implemented several of those features and tested an improved version with choreographers.

This paper proceeds as follows: first we describe the motivation for developing these features including previous work in the area, and the initial state of the NB platform with an emphasis on the development of video annotation in NB. Second, we describe initial user tests with choreographers and their reactions to the platform. Then we follow with a full discussion of potential features that could be added, based on the choreographer tests. Next, we present the selection of features implemented for this paper and their development, and describe the state of
NB after developing those features. Finally, we present choreographers’ reactions to the new features and a discussion of possible future work.

**Background**

Hot on the heels of the explosive growth of the internet, recent times have seen a huge expansion in online digital course materials. Sources like Khan Academy and MIT’s Open Courseware initially sought to bring high quality educational materials to the internet for free. They did not seek to offer classes in their entirety, but rather to make the information that people could seek out and study on their own on par with the information that was presented in universities. More recently, the edX platform sought to actually offer full classes on the internet. Class materials, including video lectures and assignments, could be uploaded and completed by students across the globe.

Within universities, there has also been an interest in using the internet to facilitate discussion amongst students and instructors. The most successful such platform is Piazza. On it, students can post questions to be answered by both other students and instructors. Instructors can also post notes to students, and students can be notified via email when a post they are following is updated. Piazza excels at letting all students benefit from others’ questions while also providing a strong class-wide communication platform. However it is completely detached from class materials. It provides a forum for discussion, but no context.

This lack of context is the primary issue that NB\(^1\) set out to address. NB’s goal is to combine a repository for class materials and a Piazza-like discussion forum into one platform. It is centered around the merits of discussing the material in-place rather than in a separate forum. The original designers of NB believed that such in-place discussion would encourage students to
comment by keeping them “in the flow” and lowering the loss of context from moving to another forum. They also hoped that displaying comments with the material would encourage participation in the discussion, and would provide a useful organization not typically provided by discussion forums. Its primary feature is the ability to annotate class materials. This is primarily used for PDF files, for which the annotation functionality is pretty mature. It also has newer and less mature features for annotating HTML documents and YouTube videos. The YouTube annotation feature is of the most interest to choreographers. As such, our primary focus was on developing this tool.

**Previous Work**

Very little work has been done on video tools for dance instruction. One of the earliest tools is called Video Traces\(^2\), a tool that was specifically designed for teaching dance composition classes. This tool revolved around students using their voice and gestures to annotate video clips. The platform is somewhat less applicable to choreographers teaching their students, and at the time it was developed, it required a lot of peripheral devices in order to work. However it did break ground in applying video technology and a specially developed app to improve dance composition.

A more complete platform for choreographers called The Choreographer’s Notebook\(^3\) was developed later by a collaboration between the University of North Carolina at Charlotte and Stanford University. The authors observed the dance production process in an academic setting and developed the tool to help choreographers make better use of their limited access to rehearsal time and space. Their focus was on enabling multiple modalities of annotation along with context based navigation and asynchronous collaboration. Their entire platform is specialized
for choreographers and video annotation, and primarily consists of a UI that communicates with the database through a PHP backend.

Our video annotator shares a lot on common with The Choreographer’s Notebook, but has a different architecture and some different focuses. While The Choreographer’s Notebook is built entirely around the choreography and dance experience, we were explicitly interested in developing these features into our existing platform in the hopes that they would be helpful and applicable to academic instructors as well. We focused much less on multi-modal input in NB. A major focus of The Choreographer’s Notebook was the ability to embed comments based on any combination of text, video, and drawings on the video. In contrast, NB focuses on a single type of comment: text supplemented by a rectangular spotlight box. While less flexible, this single type captures most use cases while being quicker and less complex in both the UI and backend required to handle more complex types like embedded video or drawings. Additionally, many of the features we implemented were not addressed in The Choreographer’s Notebook. The Choreographer’s Notebook anchors each comment to one time stamp and makes the user pause the video to view the comments, while NB allows comments to be observed during playback while still being sensitive to the limited duration of the context by allowing a comment’s author to specify its duration. The notions of being able to tag other users to point out in comments or title different sections of a video were also absent in The Choreographer’s Notebook. There was also no notion of a “thread” of related comments, which NB includes as a way to facilitate two way discussion. Though The Choreographer’s Notebook is similar in concept to the NB video annotator, we leveraged our existing platform to implement new features that were requested by choreographers.
There has been interest in video annotation for classroom instruction for even longer. Microsoft Research took an interest in this, reporting some preliminary results as early as 1999. Microsoft Research implemented MRAS\textsuperscript{4}, an ActiveX control that enables video annotation features in the browser. Though the program never became popular, likely due to the immaturity of the internet at the time, they performed user studies that demonstrated the effectiveness of annotation tools. One study showed that even though using MRAS took longer, all participants preferred MRAS to pen-and-paper notes due to the increased usefulness from context. Another study showed that students found the use of MRAS as a discussion forum helpful for facilitating conversation as well as that text annotations were generally more popular than audio annotations.

Around the same time, AntV\textsuperscript{5} was developed independently. While MRAS focused on studying how students used it for academic purposes, AntV was interested in how annotations could be organized and displayed, and used to create a new media in which the annotations are part of it. AntV groups annotations into annotation sets from a particular user. Another user can open certain annotation sets and display them, and then choose to export the video with the chosen annotation sets as a new video. AntV was also likely ahead of its time, being developed before the full maturity of the web, but it laid the foundation for a way to organize annotations and make them a part of the media for many different types of video.

**A Note on Terminology**

We discuss making comments frequently as one of the main functions of NB. A comment is a single text annotation written by a single user. Comments are grouped into threads, which are groups of comments by potentially different users that discuss the same context in the video. Each thread has exactly one location, describing the location of the “spotlight box” and the time stamp of the thread, and one or more comments, which are listed in
the order they were added to the thread. We may use the term **thread** or **location** interchangeably to refer to a group of comments associated with that location. The term **comment** refers to an individual post.

We also use several different terms to refer to our **users**. We use **user** to refer to any generic user of the NB platform, whether they are using it for dance or for academic purposes. Users who use NB for dance are referred to as **choreographers** or **dancers**, while users who use NB for academic purposes are referred to as **classroom instructors** and **students**. Additionally, some users are **admins**, in particular choreographers and classroom instructors, while dancers and students have limited privileges, and in particular cannot delete things that aren’t posted by them. Users are members of one or more **ensembles**, which are collections of users who have access to some set of materials. An ensemble may also be called a **class**.

**The Original NB Platform**

**Video Annotator**

The original NB video annotator was a small bare-bones part of the original NB and ran on Django 1.6. This version of the video annotator anchored annotations to a single time stamp in the video and displayed a spotlight box, which specifies the physical location of the video that provides the context for the comment, until the next time stamp appeared. Annotations appeared on the progress bar as red tick marks, and the controls lacked a mute button. Additionally, video upload was available only on the development branch.
Early implementation of NB video annotator:

More recently, the original annotator was replaced with a newer one and video upload was moved into the main server. The improved annotator added a separate bar for ticks, though it still did not support a notion of annotation duration. This version also used a newer YouTube API, added a mute button, and had a replay button that functioned as a five second skip backwards command.

**Tick Mark on Progress Bar:**
Initial Choreographer Reactions

We first tested the newer annotator with choreographers to get feedback on what features choreographers need in the platform. The first choreographer expressed interest in making annotations more powerful as a discussion tool, suggesting making notes private to particular dancers and supporting two way conversation (though that is already an inherent feature). She also wanted more ways to distinguish comments including a way to break a video into sections, annotations with only a time stamp, and the ability to mark important comments. She suggested using an auto-complete box to tag users, and adding a user-based setting that would enable the user to darken the rest of the video during an annotation, called “Spotlight Mode”. She also wanted to be able to send reminder emails to dancers who haven’t seen her comments. This choreographer in particular expressed a lot of interest in using the platform to explore new creative methods. To this end, she suggested the ability to collapse the screen to annotate the soundtrack, and the ability to run two copies of the same piece next to each other, synchronized, so that they can be compared.

The second choreographer wanted finer control over the time of the video, suggesting rewind or fast-forward buttons and the ability to replay from the beginning and/or a particular comment. She also suggested sliders to select a video segment to loop. She expressed a desire for both tagging functionality and comment duration settings. She also suggested expanding the “Collage View” from PDFs to video files, and providing a way to print and email comments.

The third choreographer thought it would be helpful to allow dancers to critique each other, and to define sub-groups that can be tagged. He suggested being able to draw on the video like a “football play video”, and having comments automatically pause the video so dancers could examine them more closely. He also wanted to be able to tag notes to go through at
rehearsal and filter down to them, and to allow dancers to filter down to comments pertaining to them. He was excited about it possibly making it easier to learn from video as well, noting the possibility of annotating a demonstration video.

**Features For Choreographers**

We decided to focus on a subset of these features that we thought would benefit choreographers the most, could also benefit classroom instructors, and fit most cleanly into our platform. We chose to add the notion of durations for comments, the ability to tag users in comments, annotations called “section titles” that split a video into sections with different titles, and improved access to replay functionality. We also added new email reminders attached to the tagging feature to help keep users engaged. Additionally, we added two new filters to make navigating comments easier.

Features were developed in multiple iterations, sometimes changing the interface as well as fixing bugs as they were developed. User tests, described in more detail later, informed the development of these features and their interfaces. Here we present the major features, how they changed throughout development, and basic details regarding their implementation.

**Comment Thread Duration**

One of the most obvious features was the notion of a duration for comment threads. Choreographers unanimously did not like that the spotlight box stayed on the screen indefinitely, and even for classroom instructors where things moved more slowly, annotations were often floating on screen long after they lost their contextual relevance. Our solution was to allow the original author of a thread to specify a duration, typically representing how long the context is in the specified location. Durations are represented to the user in seconds and have precision to
100\textsuperscript{th}s of a second. They are stored in the database along with the location that defines the thread as an integer, which is divided by 100 to get the duration in seconds.

**Original duration input under progress bar:**

![Image of original duration input]

Initial prototypes had the duration entered manually and only stored durations to a precision of whole seconds. The tick marks that represented the time stamp of a comment in the earliest prototype was replaced by a tick interval, which spans a length on the progress bar equivalent to the duration in time. When making a comment that would start a thread, a choreographer had to either manually count how long to make the comment for before making the comment, or guess at how long the comment was. There was also no way to change the duration after starting the thread without deleting it and starting over, making the lack of precise duration selection more problematic.

**Manual duration input moved to editor:**

![Image of manual duration input]

For the final implementation, we included buttons that the user could click to set the start or end of the comment being edited to the current time of the video player. The duration display was kept to display the duration to the user, and it can be used as a shortcut should the user
simply want to enter a duration. This allowed choreographers to be more precise in placing the start and end points of their comments. We utilized the duration by displaying the spotlight box only while the player was between a thread’s start and end point. When the comment’s duration is finished, if there is no other comment to select, the video clears the display and deselects all comments, eliminating a common complaint from earlier versions. If a user needs more time to read comments in a short duration thread, they can pause the video to do so. We also added the ability for users to change the duration of their comments in edit mode, if they started the thread.

**Automatic duration box with buttons to set time range:**

![Automatic duration box](image)

Adding durations added the potential for multiple threads to overlap. We allow threads to overlap in time, though in practice we found users shied away from making too many overlapping comments. Tick intervals would overlap as well, with multiple overlapping tick intervals making a segment appear darker. In the event that the video player is in range of multiple comments, it selects the comment with the latest start time, ensuring that during a playback every comment will be selected at some point. Due to periodic updates of non-video display being controlled by a ticking metronome, this may not hold if multiple comments are clustered within a single tick of the metronome. However, no users during testing made comments close enough to each other to elicit this issue.

**Tagging**

Another key feature is tagging functionality, requested in some form by all choreographers. In some sense this is the central new feature, since many other features can be
built on top of it, some of which we implemented. The features we implemented include filtering for tags, making threads private to only tagged users, and sending email notifications about new tags.

When making a comment, the author can choose any set of class members to tag in the comment. Tags signify that the tagged user should pay special attention to those comments. They are not exactly analogous to “hashtags” on Twitter, they are simply a mapping between users and comments they are tagged in on which additional features can be based. Our intended use case was for choreographers who wanted to tag specific dancers they wanted to see their comments. In user tests we also observed dancers tagging choreographers in questions and, less frequently, dancers tagging other dancers in position clarifications or critiques. Though dancers and choreographers seemed to understand these as the basic uses of tags, they do not have an inherent meaning, and their usage will take shape around the features that are built around the tagging functionality and may change in other contexts.

Our implementation of tagging allows the author to select a set of users to tag when a comment is written. The original author of the comment can edit the tags by editing the comment. The current implementation allows only individual “user” tags, but can be expanded to include other types of tags, such as the ability to define topics to which users can voluntarily subscribe for updates.

**Tag drop-down menu and list:**

<table>
<thead>
<tr>
<th>Select user to tag</th>
<th>Louis Lamia</th>
<th>Remove</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testuser Numone</td>
<td></td>
<td>Remove</td>
</tr>
<tr>
<td>Testuser Numone</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Our initial implementation of the tagging interface utilized a drop down menu from which users to tag could be selected. After selecting a user in the drop down menu, their name was added to a list of pending tags. Users could also be deleted from the pending tag list, and once the comment was saved, the pending tags were added to the database as tags. This had the advantage of taking up less space in the common case where few users are tagged. However, this was criticized in the first round of user tests because it took two clicks to select each user to tag, which was somewhat slow even when selecting few users and could take a very long time if tagging many people.

**Tag checkboxes:**

<table>
<thead>
<tr>
<th>Select Users to Tag:</th>
<th>Select All</th>
<th>Deselect All</th>
</tr>
</thead>
<tbody>
<tr>
<td>Louis Lamia</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Testuser Numone</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tester Numthree</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For the current implementation, we changed the tag interface to make use of checkboxes instead. The view in which comments are written and edited contains a table of checkboxes with three columns and as many rows as are necessary to fit every member of the class. The author checks the box for each member they want to tag before saving the comment. This allows users to be tagged with only one click each instead of two. Additionally, “select all” and “deselect all” buttons allow large numbers of users to be tagged just as quickly as small numbers and quick recovery from mistakes.

**Section tagging interface:**
We also implemented a prototype group tagging interface based on the existing section interface, which allows a group admin to assign class members to different sections, which can be named. It doesn’t actually implement a separate type of tag, but simply acts as a UI shortcut. The user can select a section from a drop down menu, and the interface will automatically check the boxes of all the students in that section. The author of the comment can then select additional sections or members to tag, deselect particular members to not tag, or click the “Deselect All” button to undo the tags completely before saving the comment to confirm the tags. Sections are disjoint, meaning they cannot share members, which makes the feature not perfectly suited to many choreographers’ needs. However, we believe it is an effective prototype for a group tagging feature until we determine the best interface for defining groups, which could include a separate interface for choreographers or an option to loosen the constraint that sections be disjoint.

Tags alone are largely useless with no features built to take advantage of them. Fortunately, there are many features that can be implemented based on tagging. The first feature we built in the initial prototype was a filter that could filter for comments that the current user is tagged in. Filters are options that tell NB to only display a subset of comment threads that fit the criteria of all active filters. This filter was designed to allow dancers to quickly filter down to comments directed at them, though its usage will obviously vary with the way the group uses tagging. We also styled the filters to look more like buttons, after user testing showed that many users did not realize they were clickable.

**Filters, currently filtering for comments the current user is tagged in:**

![Filter Interface](image-url)
We also found in first round user tests that users wanted to be able to see who was tagged in a thread, so for the current implementation, we added a table under the video that would display a list of users who are tagged in the selected thread. The table is based strictly on the full thread. It does not specify which comment users are tagged in, and users only appear once in the table no matter how many times they are tagged in the thread.

**Tagged user table:**

<table>
<thead>
<tr>
<th>Tagged:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testuser Numone</td>
</tr>
<tr>
<td>Tester Numthree</td>
</tr>
</tbody>
</table>

**Email Notifications**

We built a system of new email notifications around tagging functionality. Choreographers wanted the system to help get dancers to read comments directed at them sooner and more frequently. To help with this goal, we leveraged an existing system that tracks which comments have not been seen yet so that they can be rendered in bold when the video is next accessed. We wrote a command line server command that reads through the database to determine which tagged comments have not yet been seen by the tagged user. It then sends an email to each user with a direct link to each new tag. This command can be scheduled to run daily on the NB server. Additionally, since choreographers wanted to bother their dancers until they see their comments, users cannot opt out of these batch notifications.

Additionally, we wanted a more immediate email system to continuously engage those who want to be more engaged. We implemented an immediate notification system as well, which emails a user immediately when that user is tagged in a new comment along with a link to
that comment. Since this could potentially result in large volumes of email, which is not the case with a daily summary, individual users have an option to turn these notifications off.

User options including choices for new option:

Email Notifications
Comments that I wrote
New reply in a discussion to which I participated
New comments in classes for which I'm a staff member (if any)
New comments in which I am tagged

Tag Privacy

We found during first round user tests that choreographers wanted to use tags to send personal critiques to dancers, but did not want to publicly make such a personal critique for fear of making the dancer feel self-conscious. This could cause valuable feedback to not be made. To address this, we added a “Tag Private” privacy setting. In addition to the options that were available previously, in the current implementation, a new thread author can set a comment to be visible to “Myself and Tagged Users.” Under this privacy setting, only users who are tagged in the thread can read it. Being tagged anywhere in the thread allows access to the entire thread. Thus, any user tagged in a “Tag Private” thread can allow another user access to the thread.

“Tag Private” setting selected:
There is some contention regarding whether it is a good feature to be able to make a comment private to an arbitrary subset of users. On one hand, the feature was requested because choreographers felt awkward publicly making personal comments, suggesting more comments will be made with the feature. However, dance is highly collaborative and even “personal” critiques can benefit others. Some choreographers contend that dancers should be able to make such comments publicly. To resolve this issue, we added an ensemble-level option, which an admin can use to toggle whether “Tag Private” comments should be allowed in that class.

**Ensemble options list with new option underlined:**

- Allow users to write 'staff-only' comments: ✓
- Allow users to write anonymous comments: ✓
- **Allow users to make comments private to tagged users only:** ✓
- Allow guests (i.e. non-members) to access the site: □
- Allow users who have the 'subscribe link' to register by themselves: ✓
- Allow users to download the PDFs: ✓
- Allow users to add any PDF accessible on the internet by pointing to its URL: □

**Section Titles**

The third major feature is “section titles,” essentially an annotation type that focuses on breaking the video into distinct parts rather than discussing some detail of the video. To implement this, we added a Boolean field to the comment location that is shared by all comments in a thread, specifying whether the comment is a section title or not. From the user standpoint, specifying a section title is as simple as checking a box when starting a new thread. Subject titles are subject to restrictions, specifically that they must be placed at a new location (i.e. it cannot be a direct reply to a non-title annotation), which reflects their distinct purpose in contrast to standard comments. When the section title box is checked, the buttons that allow the user to select time stamps are disabled, preventing the user from making “long titles.”
Section titles also have special properties in the way they are displayed, which make them stand out. Standard tick intervals highlight blue while the mouse hovers over them and green when the thread is selected. Tick intervals representing section titles remain dark so that they always stand out against other tick intervals. They turn gray when hovered over with a mouse pointer, but otherwise stay black. They also appear at a constant six pixel width regardless of how long the video is, which changes the scale of regular comments. Section titles are also underlined in the list of comment threads, and users can set a filter to show them only section titles, which can help navigate long or densely commented videos. When the video
selects a section title, the spotlight box that normally highlights the location of the current thread does not display since only the time stamp is relevant. Additionally, a title display was added above the video, which displays the title of the most recent section title that occurs earlier in the video. In the event that all of the section titles are still in the future (or there are none at all), it displays the name of the video that was given to YouTube.

**Section titles next to overlapping standard comments in progress bar:**

![Diagram](image)

**Section title displayed above video player:**

**House Party Beginning**

**Miscellaneous Tweaks**

The original NB implementation had a “replay” button, that functioned as a five-second skip back. Choreographers were confused by the function of this button and wondered why it always skipped back 5 seconds regardless of context. In the initial prototype, we made the replay button skip back to the last comment selected, while double clicking it skipped back to the beginning of the video. The first round of user tests suggested that the double click function was undiscoverable, users were looking for an easy way to skip back to the beginning, and they often thought clicking on the thread in the thread select view was a more natural way to skip to a particular comment. Thus for the final implementation, we set the replay button to skip back to the beginning of the video, which proved a popular decision in final user tests.
Replay and Play buttons:

Choreographers also sometimes had issues with threads being selected late. Part of the NB Javascript code is a metronome, which periodically triggers an event telling each view to get the current video time and update the state of the app based on it. In the initial implementation, the metronome ticks only once per second, so if the metronome ticks when the video is just short of starting the next comment thread, the thread would not be selected until the next tick a second later. To fix this, we changed the metronome to tick five times per second, noticeably increasing the responsiveness of the app.

Integrated Upload Dialog:

We also tweaked the file upload interface to better integrate the different types of files that can be uploaded. Initially, the file upload dialog was specific to PDFs and uploading a YouTube video or HTML document required following a link to a different page. We replaced it with a single dialog with radio buttons that select which upload form is visible.
User Tests

We held two main rounds of user tests for the NB video annotator during development. The first round focused primarily on choreographers and single-user interaction with the system. An initial set of these tests was scheduled on an early prototype, and more tests were run on prototypes of the system as the users’ schedules allowed throughout the course of development. A final round of tests included groups of choreographers and dancers and focused on group interaction. These tests were held on a near-complete version of the system. User tests helped to inform development of the system, as well as our conclusions about the current implementation of the system and suggestions for future work.

First Round Methodology

User Tests were conducted in two parts. In the first part, the users were walked through a scenario in which they had to perform various tasks related to the new features, such as using titles to split the video into 2 or 3 parts, and tagging a user in a comment pertaining to him. Users were observed to see what they would do to complete the task, and could be given a hint once they tried a few times on their own and gave up.

In the second part, we discussed the features with the user for their reactions to them. We asked how they thought the features might be used in practice and if they wanted to be able to do anything else with them that they couldn’t do. We also discussed any challenging moments or glitches encountered during the test and possible ways to improve the interface. Additionally, we solicited their feedback on the interface to find ways we might make the platform more usable.
Prototype used for first round testing:

First Round Tests

The first round of tests revealed that several basic aspects of using the annotator were not obvious to users unfamiliar with the platform, and created a high barrier to entry. For instance, some users needed to be told to click and drag to create an annotation, despite the help menus that explain it. Some clicked the tick bar expecting it to create an annotation, though on commented videos it might raise the issue of the tick bar playing too many roles. It was suggested that an improved help menu or help page, perhaps one specific to the video annotator,
could help make it easier to learn how to annotate. However, given that a help menu exists, it is unlikely that simply improving the help menu would fix the problem.

Some users were similarly confused by filters. Nothing explicitly identifies the buttons that act as filters except the tool tip that only appears if the user hovers over it with the mouse for a short time, which few users actually did. One user activated a filter, which filters for comments the user is tagged in, in an attempt to tag another user. They then didn’t realize the filter was on, which led to comments unexpectedly not displaying. This suggests filtering should be made more obvious, and perhaps the filters themselves should be more clearly labeled.

Choreographers in particular indicated that precise control of time was helpful to them. They wanted to be able to pause immediately without clicking a separate button and to easily control the position of their player and their annotations at least down to the second. Controlling the position of the player by only clicking the progress bar becomes less precise as the video gets longer.

We also determined that the behavior of the replay button was needlessly complex. For these tests, the replay button was configured to skip back to the last selected comment. Double clicking the button would go back to the beginning. Unfortunately, the double click feature was undiscoverable, and many users thought it was more natural to skip back to a particular comment thread by clicking it in the thread list. This also raised the question of whether the video should have an option for automatic stopping at the start or end of threads, which may help users focus on particular comments.

Finally, tagging needed some refinement. Users thought tagging more than just a couple of people was clunky, requiring each person to be selected individually in a drop down menu. One suggestion was to use a grid of checkboxes with select all and deselect all options instead of
a drop down menu. Users also wanted to be able to see the tags that were applied to a comment, not just filter for their own tags. Additionally, some choreographers were looking for a “private tag” feature, where they could make a comment private to only the people who are tagged, in particular so that they could make private criticisms. These requests were all addressed in later iterations.

Final Round Methodology

The final round of user testing was conducted as an end-to-end test involving groups of choreographers and dancers. Choreographers taught their choreography to dancers as they normally would, taking videos of the performances. The choreographers then uploaded those videos to an NB class and invited their dancers to join the class. Choreographers and dancers then collaborated on the videos for review.

Afterwards, the choreographers and dancers were invited to give feedback via an online form, and those that agreed were also interviewed in person about their experiences. They reported on bugs and confusing UI features. They also gave feedback on which features they found most helpful, and other features that they wanted. We also examined their usage of the system to draw conclusions on how they used the system and determine which features are most used.

Participants in this test used a near-complete prototype with all the major features present, but some features were further expanded or had the interface tweaked since this test for the current implementation. In particular, the duration of a thread was manually entered and only maintained precision to the whole second. Additionally, tick intervals representing section titles represented one second in duration rather than a constant six pixels, the instant email
notification for new tags was not implemented, and the metronome that updates the display
ticked only once per second.

Near-complete prototype for final round testing:

![Final Round Tests](image)

Final Round Tests

Variable comment durations were universally appreciated by testers. Choreographers in
particular appreciated the ability to set durations for the comments they were making, and
dancers appreciated how the selection and deselection logic attached to that made watching the
video a clean experience where it was easy to see where comments were and skip between them.
Though they liked the feature overall, choreographers wanted a bit more fidelity in setting
comment times. Some choreographers found that comments were usually selected slightly after
the time that they wanted it selected, due mostly to the slowly ticking metronome in the version
being tested. They also disliked having to manually enter the duration of a thread, forcing them to either guess or count a comment’s duration ahead of time. Additionally, this version did not have a way to change the time or duration of a comment thread after it is made, forcing the user to delete it and rewrite the comment. Both the inability to edit the time of a comment and the imprecision in placing comments would be addressed in the current implementation.

There was also some confusion about how to use some basic features. Several users couldn’t figure out to click and drag to make a new annotation. A few tried to start threads by clicking on the tick bar or directly on the video, but many simply remained confused until they asked for help. Other users couldn’t figure out how to delete or edit a comment, though the feature exists. Others couldn’t figure out how to tag other users, missed the ability to filter, or misunderstood how certain filters worked even when a correct definition was technically available by mousing over it. This suggested that our current help system, consisting of a panel with short directions and a link to a larger help menu, was inadequate, and being missed by many users.

**Choreographer answers a dancer’s question:**

```
I wasn't supposed to end up so close to [redacted], was I?
[Redacted] 1 – 20 Aug, 01:45AM

No, you should have been further forward in the empty space
[Redacted] [me] – 05:17AM
```

In practice, we observed that most threads are short, with the majority of threads containing only a single comment. These threads were typically a choreographer making a correction. Single-comment threads created by dancers usually either pointed out a mistake of
their own, or complimented another dancer. Dancers tended not to critique each other. Amongst the threads that are longer, most of them consist of a question for a choreographer, and their response. We never observed a multi-comment thread that did not revolve around questions and answers of some sort. Though not all comments were “strictly business”, those that were more social in nature were primarily single-comment “shout out” compliment threads targeted at a particular dancer.

We also observed the usage of tags in a group setting. In addition to informing us about how we can improve the currently implemented features, it can suggest new features we may want to add around tagging. Private tagging and email notifications were planned primarily for the choreographers, but played just as strong a role for dancers. Dancers frequently used the tool to tag choreographers, knowing that they would receive email notifications about them. Some dancers appreciated the sense that the system would notify the choreographers until they read them. They utilized the private tag feature to target questions at only the choreographer and a couple of other dancers that the question pertained to (e.g. “Should I be behind that person or in that window”). The email notifications were originally developed for choreographers who wanted to prod their dancers into checking their comments, but the dancers appreciated the ability to prod their choreographers into looking at their questions as well. We also received a suggestion to allow users to tag users in the text of a comment similarly to a Twitter tag, though this is technically challenging given our current representation of comments. This suggested that while we may have found a good input format with checkboxes, that it is not optimal for all uses and that additional flexibility may be desirable in some cases.

Some choreographers seemed to employ section titles more than others. While some immediately appreciated the ability to split videos into parts, others didn’t make much use of
them even once explicitly told about the feature. This seemed to be driven largely by personal choreographer preference, but some expressed that they thought it was more useful for longer videos, while shorter videos showing a single section of the dance were less likely to have section titles written for them. Those choreographers that did use section titles appreciated having the title displayed above the video, and once they saw that feature work, they understood the concept of section titles. However, they also thought that dragging a box to start a thread that would ultimately become a section title seemed strange given that the box would not be displayed. Some users suggested that clicking on the tick bar should begin a thread, possibly defaulting to being a section title. Some also thought it would be helpful to be able to change whether a comment is a section title by editing it, though this would be at odds with removing the need to click and drag a selection for section titles.

There was also some disagreement about the “Tag Private” setting, and how available it should be to users. Some users thought that a tag private setting made sense for making personal critiques, while others felt that dance is a team activity and that even personal critiques benefit everyone. Some users suggested a compromise, that making comments private to tagged users should be something only choreographers can do. Some users, particularly those that didn’t like email notifications, also expressed concern about the frequency with which other dancer’s comments can trigger email notifications. Though notifications in this prototype were sent no more than once per day, it’s clear some users don’t like receiving much email, and that there is some tension between choreographers wanting to engage their dancers often, and dancers not wanting to be bothered excessively.

Some users also suggested that navigating throughout the app as a whole was confusing and could get cluttered. In particular, they disliked the fact that opening new files in NB would
always open a new tab. While some other users find this helpful, these users dislike not being able to use the back button to return to the desktop from a video, and noted that without user vigilance, NB can easily open many tabs at once.

**Conclusions**

Our users unanimously agreed that the NB video annotator was a useful tool compared to how they might otherwise review a video and comments, with many users appreciating the ease at which they could skip between comments at different times and specify specific parts of the video to discuss. Some of them also said that the email reminders and the increased ease of skipping to relevant parts of the video compelled them to check the comments more often than they normally would. Additionally, our users appreciated the community aspect of the tool. While the traditional “comments and time stamps” method of sharing notes is typically one directional, a choreographer writing notes for dancers, NB enables dancers to critique other dancers and have conversations with the whole group in context.

The NB platform is in constant iterative development. While the current version of NB is neither perfect nor complete, and the list of helpful features we could add is indefinite, we believe we have successfully adapted a platform originally developed for the classroom to support the dance studio as well. By focusing on the video annotator and developing features to make the experience cleaner and engage users with each other over time, we have developed NB into a useful tool for choreographers and dancers.
Current implementation of video annotator:

![Image of video annotator interface]

**Future Work**

Final testing in particular seemed to suggest that several users were initially confused by how to use certain features, or thought that certain things were impossible even if they weren’t. Furthermore, different users were confused by different things, and some users were confused by things that were explained in the help menu. This suggests that the current help menu, consisting of a small splash screen with short instructions and a link to a more detailed help page, is insufficient. Simply because the information exists on how to use the system doesn’t mean the user will access it or completely understand it. The NB help system could be expanded to be more prominent and helpful. This may be accomplished by expanding the original page and explicitly notifying the user about it, but more novel solutions could produce better results.
Possibilities include a help dialog for first time users, or a “help mode” where users can mouse over different elements and read tool tips about how to use them.

There is also a lot of room to improve the discoverability of many NB features. While we focused on adding new features to support choreographers, there is a lot of room for purely UI focused work to improve the usability of the current features. Ensuring that all features are discoverable is the most basic goal, but even when users could find how to do things, they often tried to do them in other ways first. This suggests that there are more UI shortcuts that could be added to give users another way to discover a feature, or help them access it in a more natural and efficient way when they want to.

Additionally, users wanted finer control over the time of the video player. Buttons could be added to scrub through the video, though the ideal behavior would likely depend somewhat on the length of the video and the context. Since finer control was more possible in smaller videos, some suggested including an ability to zoom in on the progress bar so that they could click a location to seek to in a longer video with the same fidelity as a shorter video. It was also clear that users generally did not know and could not discover the keyboard shortcuts, which could help address these issues as well.

One question our user testing couldn’t definitively answer is the permissions that should surround tagging and its associated “tag private” visibility setting. Choreographers differ in opinion, with some feeling that more public discussion is always better and others feeling that privacy makes sense in some cases. In the current implementation, we allow the choreographer to choose via an ensemble-level option. However, it is unclear whether one is quantitatively better than the other, and answering that would prove very interesting.
There are many more features that could be built on top of the tagging functionality. Designing new types of tags and the features that access them is one area for future development. One suggestion we received early was to create subscription topics that users can voluntarily subscribe to. Such a feature seems like it may benefit classroom instructors more than choreographers, though it could potentially be adapted for tagging functional groups of dancers. While the tagging feature was developed at the request of choreographers, it may serve some use to classroom instructors as well, though it would likely serve a different role. More work is needed to determine how tagging functionality can benefit the classroom setting, and what further features could benefit choreographers.

Appendix A: NB Architecture

A Note on Terminology

The term model is used to refer to the representation of the application data, backed by a Postgres database. There are two types of model, one on the server and another on the client. The Django model is on the server, contains all the data in the application, and is provided by the Django framework in Python. The Javascript (JS) model is on the client, only contains the subset of the application data which is necessary for the current view and allowed to be accessed, and is a custom part of NB. We will specify which model we are referring to when it is unclear from the context.

Server and Client

The server runs on the Django web framework, which provides a Python API for interacting with the database. The database itself runs on Postgres. NB was built on Django 1.6, which required third party projects or manual database manipulation to make changes to the
model. We updated Django to version 1.7 at the start of updates to the video annotator, which included features for Django to automatically make changes to the database schema.

The application is primarily the display of data and client side interaction with that data. Thus the bulk of the NB user interface is in Javascript compiled by grunt. When the user loads a page, the NB server sends the Django template page, which in NB is a lightweight HTML page that does little more than load the Javascript file corresponding to that page. The Javascript file is where the direct UI interaction occurs.

**Diagram of client-server interactions for page setup:**

Since the Javascript runs on the client, it cannot directly query the Django model. To give the client access to the data it needs for the current session, a copy of a subset of the data from the Django model is sent through an RPC call to the client. The client keeps this data in a model API that was built specifically for NB, but heavily resembles the Django API for querying the model. This subset of the model will be referred to as the Javascript model, and it runs in the
client’s memory. Since it runs on the client it cannot be trusted. Access control is performed on the server, and data that a particular client should not have access to is simply not sent to it. Once the RPC call to the server returns the data it needs for that session, it loads it into the Javascript model for access by the client’s UI. The client then only needs to communicate with the server if it modifies the server’s state in some way, and it only needs asynchronous RPC calls as a mechanism to do so.

Views

Once the page is initialized and the Javascript model is loaded, with the data for that session, the actual interaction with the data all occurs client side (with the exception of RPC calls the client may make). The actual video is streamed through the YouTube API, and all other data is stored in Javascript on the client. A page usually consists of multiple views, each taking up a part of the screen and fulfilling a particular purpose. Views are independent, but can communicate via the concierge and share a Javascript model. When the user makes a change that affects the database on the server, the client makes an RPC call to the server to ensure that the changes are reflected in the database. Once the RPC call returns, a callback function updates the Javascript model with the same changes that the server made.

Most changes are communicated between views via updates to the model, but sometimes when a view is responsible for sending an RPC call that updates the server, it needs to collect information from other views first. When views need to directly communicate with each other in a way that doesn’t directly affect the data, they pass events using a custom NB object called the concierge. While the event passing functionality in some sense acts like an RPC server between views, it has no callback mechanism and is best suited for pushing data rather than pulling it.
The following diagram illustrates how data might be passed between views when uploading a new annotation for a video.

**Diagram of client view interactions**

Since the docView has the YouTube player and therefore the info on the current location and time, that information is saved in html attributes by the docView so that it can be accessed by the editorView. Once the editorView assembles the complete annotation, it sends an RPC call to inform the server of the model update, and in the callback, updates the Javascript model.
This interaction is typical of the way NB makes changes to its data, assembling all the relevant data in a single view, which makes an RPC call to the server. The views are loosely coupled. While they each have their own context allowing them to be developed independently, all the views on a single page usually pertain to the same data. However, they both refer to the same data in the shared Javascript model, and communicate primarily through updates to the model. In the case where views need to communicate directly, the concierge allows simple and direct event and data passing, though it lacks a callback mechanism.
Appendix B: Codebase Documentation

The Hitchhiker’s Guide to the NB Codebase
Table of Contents

Introduction
Contributors
Setup
  Linux Install Notes
  Mac OS X Install Notes
    Install Python 2.7.5
    Install apache2 + mod_wsgi
    Install Postgres
    Install Python packages + Django
    Install Grunt
    Install mupdf
  Other
  Follow INSTALL part 2:
Where Things Are
Working With The Client: Javascript
  Architecture
  Model, View, and Concierge
    View
    Model
    Concierge
  Models in Javascript
    RPC Server
    Setup
    Query
Models: Database Schema
HOWTOs: Quick Instructions You May Not Know
  HOWTO: Make Model (Database) Changes
  HOWTO: Run The Server Constantly
Everything is Broken! (Debugging Tips)
Introduction

This is the definitive quick and dirty guide to the NB codebase. As I write this, I am a grad student who was tasked with building something new into the NB system, and found that the deeper I dug into this system, the more pieces I found I needed to dig into to figure out how they work. This is my attempt to gather this information so that future students working on the NB codebase can spare themselves many of the headaches I encountered.

This is NOT meant to be an exhaustive documentation. It is a quick and dirty guide to what you most probably need to know to get working on the NB codebase. It will not cover things that can be found in other documentation, such as jQuery or Django, but should let you know when such things should be referenced. It is also meant to evolve over time. If you find yourself needing to figure out something I didn’t cover, please feel free to add the information about it and add yourself to the contributors section.

- Louis Lamia
  
  *original author*

Contributors

**Original Author**

Louis Lamia (August 2015)

**Special Thanks**

David Karger - Supervising Professor

Sacha Zyto - System Construction and Technical Guidance

**Other Contributors**

Carolyn Chang, Jen Liu, Birkan Uzun, Carolyn Zhang - Mac Install Notes

Carolyn Zhang - Working With The Client: Javascript
Setup

Linux Install Notes

See INSTALL file

Mac OS X Install Notes

These instructions are for users who want to set up and run their own instance of NB on OS X. However, you may develop on the ubuntu server as well.

Install Python 2.7.5
(http://www.python.org/download/releases/2.7.5/)
(2.7.2 causes issues with Django 1.5.1)

Download the Mac Installer for your system. After installing, go to the "Python 2.7" subfolder of the system Applications folder, and double-click on the "Update Shell Profile" to use 2.7.5 from the command line. After doing that, type python --version from the command line to confirm you're using 2.7.5

Install apache2 + mod_wsgi
(http://heisel.org/blog/2009/09/25/mod-wsgi-mac/)

Install MacPorts if you haven't already. Then:
sudo port install apache2 mod_wsgi

Alternative—if you're using Homebrew on Mavericks, run:
$ sudo ln -s
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/
/Applications/Xcode.app/Contents/Developer/Toolchains/OSX10.9.xctoolchain

$ brew install mod_wsgi

Using your favorite text editor, add the following to
/opt/local/apache2/conf/httpd.conf:

*In Mavericks, the file is located at
/private/etc/apache2/httpd.conf
*if file is read only, first run: sudo chmod 777 httpd.conf

LoadModule wsgi_module modules/mod_wsgi.so

In the same file, uncomment:
Include conf/extra/httpd-vhosts.conf

Install Postgres
Install Postgresapp: http://postgresapp.com/

Add the following to your PATH (in .profile, .bashrc, .bash_profile, or .zshrc, in your home directory):
cd ~/PATH="/Applications/Postgres.app/Contents/MacOS/bin:$PATH"

Start Postgres, then run the command-line interface by typing psql
If successful, enter the following to create a local NB database:
# CREATE ROLE nbadmin WITH SUPERUSER;
# CREATE USER nburop WITH PASSWORD 'ur0p34501';
# CREATE DATABASE nburop;
# GRANT ALL PRIVILEGES ON DATABASE nburop to nburop;

Install Python packages + Django
sudo easy_install pip


sudo pip install numpy
sudo pip install pypdf2
sudo pip install pycparser
sudo pip install xlwt
sudo pip install python-openid
sudo pip install setuptools
sudo pip install --pre pytz

sudo pip install Django==1.5.4
sudo pip install django-openid-auth
sudo pip install django-facebook-oauth

(NOTE: Currently ‘facebook’ module is missing, so it’s commented out in the INSTALLED_APPS list in settings.py. Might be resolved if we install Sacha’s fork of django_facebook_oauth instead.)
Install Grunt
Install nodejs: http://nodejs.org/download/
Then install Grunt: http://gruntjs.com/getting-started

sudo npm install -g grunt-cli

Install mupdf
sudo port install mupdf
Change instances of “pdfdraw” in the codebase to “mudraw” (currently in apps/upload/views.py, line 93).

NOTE: To ignore this file on git, go to your working directory, and edit .git/info/exclude to add this line (you might have to create it as a new file):
apps/upload/views.py

Then in the terminal, type:
git update-index --assume-unchanged apps/upload/views.py

Other
Install imagemagick: http://www.imagemagick.org/script/binary-releases.php#macosx
Install ConTeXt (for rich, i.e. annotated pdf generation):
http://wiki.contextgarden.net/Mac_Installation (I did it via MacText distribution, as recommended)

Start postfix (to enable sending mail if faced with “Connection refused”):
sudo postfix start

--------------------------------
NOTE: Things on dependency list not installed:
g++ (used to compile node) → comes with xcode? (on Mavericks, need to install different gcc from xcode default)
-----------------------------------------

Follow INSTALL part 2:
2- Installation commands:
Once you’ve satisfied the dependencies listed above, you need to run the following installation commands:
    cd nbproject #(or whatever name you may have chosen for the root NB code directory).
npm install #in order to install specific grunt modules, such as grunt-css, execSync

make django #create configuration files. You can safely ignore the "Error 127" message

In settings_credentials.py, set DEBUG = True
Also set the following for DATABASES (to match the table created in Postgres above):

```python
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2', # Add 'postgresql_psycopg2', 'postgresql', 'mysql', 'sqlite3' or 'oracle'.
        'NAME': 'nburop', # Or path to database file if using sqlite3.
        'USER': 'nburop', # Not used with sqlite3.
        'PASSWORD': 'ur0p34501', # Not used with sqlite3.
        'HOST': 'localhost', # Set to empty string for localhost. Not used with sqlite3.
        'PORT': '', # Set to empty string for default. Not used with sqlite3.
    }
}
```

```
sudo make create_dirs #create root folder and some folders below that for nb data
make django #one more time...
# [If you're deploying a production environment, use the cmds given in output in order to configure apache]
# [then configure the cron jobs (cf 5)]
grunt #Compiles js and css.
```

Then from the apps directory, you should be able to:

`./manage.py syncdb`
`.manage.py runserver`
In your browser visit http://localhost:8000

You should be able to see the home page, but the server will throw an error if you try to create a group without configuring an endpoint for the authentication email

**Configuring the Email Endpoint**

From the apps directory, open nbsite/settings_credentials.py and uncomment the lines that set the values of EMAIL_BACKEND and EMAIL_FILE_PATH (~Line 26 currently). Then set:

```python
EMAIL_BACKEND =
'django.core.mail.backends.filebased.EmailBackend'
EMAIL_FILE_PATH = '/tmp/nb_emails'
```

**Accessing the NB Desktop**

Open [http://localhost:8000](http://localhost:8000) in a browser again and click the Log In/Sign Up link in the top right, then choose to create a new group. Fill out the form (make sure you remember your username and password so you can log in later) and submit it. It should tell you it sent the verification email. There should now be one file in the directory /tmp/nb_emails. This is the verification email. Read the file and look for the line:

- To start using NB, please visit [http://localhost?ckey=XXX](http://localhost?ckey=XXX)

XXX here is the verification key. In your browser visit: [http://localhost:8000?ckey=XXX](http://localhost:8000?ckey=XXX)

(So be sure to insert the port number into the URL if you copy/paste)

You should now be on the NB Desktop!

**Where Things Are**

- Django (server) code is primarily found in **apps**.
- **apps/base** contains most of the Python Django code. **models.py** contains the model info. Many of the other files in that directory act as libraries for other code (like the RPC server)
- The RPC server is a major exception to the above rules, that code is found in **apps/rpc**.
• Javascript (client) code is primarily found in **content**.
  • **content/ui/admin** contains several Javascript init files that load first and set up the views. This is where the Javascript model should be primarily populated.
  • **content/modules/dev** contains most of the individual views that are loaded. Any given page likely consists of views from one or more of these files, and this is where direct UI interaction occurs.

**Working With The Client: Javascript**

This section is for those who want to get up and running developing the Javascript client. UI programmers may wish to start here after getting set up. The focus here is the architecture of the interface and the basics of interacting with the application data.

**Architecture**

by Carolyn Zhang

The NB desktop is a single-page application that relies heavily on client-side rendering. NB uses the Python-based Django web framework in a client-server pattern. The server mostly stores models and templates for static pages (such as the landing and settings pages). When a user visits the NB desktop app, the server sends a mostly-empty HTML file and a **heavy** Javascript file that handles application layout and logic. This Javascript is organized in a roughly model-view-controller pattern, though view rendering is sometimes tightly coupled with controller code.
**Figure 2.** Diagram indicating the primary Javascript controllers for the main parts of the desktop interface.

The main components responsible for rendering the desktop front-end are:

- **init.desktop.js**
  Initializes the view by setting the appropriate start state (the Home view) and setting layout parameters for the ui.perspective.js layout wrappers.

- **ui.perspective.js**
  A controller/view hybrid that wraps major components of the page in resizable sandboxes and lays them out in the viewport (see Figure 2).

- **ui.treeview.js**
  Controller/view hybrid that determines what to display in the class tree left sidebar, and how to handle click events in the left sidebar (see Figure 2). *The Concierge* makes sure that the state of the left sidebar stays in sync with the view in the right filesview pane.

- **ui.filesview.js**
Controller/view hybrid that determines what to display in the list of files on the right part of the screen (see Figure 2), and how to handle click events in the files list.

- **pers.js**

  Controller for many shared components and actions across all of NB, including user authentication. pers.js inserts and controls the login dropdowns in the upper right-hand corner of the page (see Figure 2), which also appear in many other NB application pages.

### Model, View, and Concierge

**View**

A view is an independent, often reusable interface that makes up part of a page. For example, the video annotator page is made of a docView that displays the video content, a notepaneView that displays the comments, and an editorView that lets a user write new comments. Having reusable views is good code design. It enables, for instance, the video annotator being turned into a pdf annotator by swapping the docView and leaving the notepaneView and editorView as is. It also means some information you want may be generated elsewhere, and you need to use background machinery such as the model and concierge described below to access it. A view is just a client-side
JavaScript script (perhaps with associated CSS) that generates HTML. Thus any information stored in the view itself is local to that view and disappears when the view is closed.

Model

The model is invoked when things need to be stored persistently. It is provided by the Django web framework and backed up by a database. For example, each video has a model that stores the YouTube ID and information for all its annotations. When the docView containing the video loads, it loads the model and uses that to determine which video to load and what comments to display. Views can receive notifications based on models with an “update” function. The docView, for instance, receives a model event when the page loads to tell it where to place the tick marks. However, models are intended for persistent data storage, not event passing. It doesn’t allow for simply passing events between other components, but instead allows you to listen to events generated by the model.

Since the views can’t directly fetch data from the model, we also have a copy of the model in Javascript. It only contains the data that’s necessary for the current session, not the whole NB database. It’s not
SQL based (Firefox doesn’t support in-browser SQL) so Sacha used an object stored approach inspired by the backend from the Exhibit project, and added separate “tables” for objects and indexes for performance.

**Concierge**

The concierge is a custom NB script, though I was unsure if it was part of the Django framework initially because it provides what seems to be a commonsense functionality. However, it is distinct from Django in that it does not necessarily require the server, and it is initiated by the client. Its function is to allow different views on the client to communicate with each other and to make requests from the server. It passes events between different views, and allows them to communicate using “trigger” (sending for example, the position of an annotation to the editorView). This is great for allowing views to communicate directly in a non-persistent manner, implementing a sort of RPC server between them. Unlike an RPC server, this lacks any sort of reply mechanism to get data back to the view that initiated the call, making it unsuitable for basic queries. In general, the concierge is better suited to pushing information rather than pulling it. There is also a “get_component” method that is used in user identification and a “get_state” method, both
helping to implement a shared data store. get_component typically returns a function that makes an RPC call to the server. get_state enables access to session information.

The model and concierge have distinct roles. The model is meant to mirror the Django model. When a page is first loaded, all the info from the Django model that is relevant to the current session is loaded into the Javascript model. That data can then be queried and accessed by the application. More detailed use info can be found in the next section.

In contrast, the concierge doesn’t directly mirror any particular Django element. The concierge is a singleton, and grants access to session-wide variables and an event passing mechanism. The most basic use of the concierge is using the concierge.trigger method to pass events between views. The event object passed into trigger is passed into the default_handler of every view, and should at least have a type property specifying which part of the handler should handle the event. If defining a new type of event, make sure every view that needs to know about it lists it as a property in listens in prototype.options or the handler will never receive the event. The method concierge.get_state is used to access session-wide state variables.
The method `concierge.get_component` returns functions that make RPC calls, providing a way to make a request from the server from within the code. New components can be added (usually in a Javascript init file) with the `concierge.addComponents` method.

### Models in Javascript

Django models are native to Python. The typical way to access models in Django would be to pass them through to a template. Since the application is heavy grunt-compiled Javascript, NB has its own home-made interface for accessing the model through Javascript. This allows the server to ship all the relevant data at once when a view is loaded, then the client can query its own copy of the model when performing local operations.

`GLOB` is a global object passed into all the Javascript and `GLOB.pers.store` represents the models. `GLOB.pers.store` is passed into individual components via the `set_model` function and typically saved as `self._model` or something similar.

The accesses to the JS model can be broken into two categories: **Setup** operations, which load the JS model with appropriate data; and **Query** operations, which query the model and return a QuerySet.
representing a subset of the data in the model. Additionally, for the actual interfacing with the Python Django interface, an **RPC server** is run. The methods it supports calling can be found in **apps/rpc/views.py** and it is usually invoked with a Javascript call to

```
GLOB.pers.call(method, P, cb),
```

where `method` is a string containing the name of the Python method, `P` is an object containing the payload passed to the method, and `cb` is a callback function called when the RPC call returns.

**RPC Server**

The RPC server is a separate app that the main application can talk to. It provides a way to call Python methods from Javascript code. The entirety of the RPC server is in **apps/rpc/views.py**. A list of methods that can be called via the RPC server called `__EXPORTS` is found near the top of the file. If adding a new RPC method, make sure to list it there.

The RPC methods are defined below, with each method taking two arguments. The first is the `payload`, a dictionary representing the arguments of the function. This is explicitly passed into the Javascript code that calls it as an object. The second is the `request`, a dictionary
that is automatically passed into the RPC call containing information like the User making the request.

A library called **utils_response** handles a lot of the manipulation of the requests. It can be found at `apps/base/utils_response.py` and is usually imported as **UR**. The most frequently used methods:

- **UR.getUserld(request)** - takes a request as an argument and returns the ID of the User who made the RPC call
- **UR.prepare_response(payload, errcode, errmsg)** - An RPC method should always return a call to this function. **payload** is a dictionary that is passed into the callback function given to the original Javascript call. The other arguments are optional, and are typically omitted unless there was an error. If there was an error, **errcode** should be something other than 0 and **errmsg** should be a message describing the error

To call the RPC method from Javascript, you should use the method

- **UR.qs2dict(qs, names, pk)** - A method that converts a QuerySet into a Dictionary mapping a record’s primary key to the record. A dictionary returned by this is suitable for being passed back to
Javascript. **qs** is the only mandatory argument and is the QuerySet to convert. **names** is an optional dictionary that maps the name you want a field to have in the output to the name it has in the Django model. In files like `annotations.py`, the names are defined in a global variable at the top called **__NAMES**. **pk** is an optional field that defaults to “id”, and specifies what the primary key field is called according to the output. This is the field that will be used to index into the dictionary.

`GLOB.pers.call(method, P, cb, eb)`.

**method** should be a string with the name of the RPC method you are calling. **P** is the payload that is passed into the Python RPC method as a dictionary. **cb** is a function that takes a single payload object as an argument. When the RPC call returns, the Javascript callback function **cb** is called, and the payload that was passed to `UR.prepare_response` in the Python RPC method is passed in as the payload argument to that callback. **eb** is an optional argument, which provides an alternate Javascript callback function for the case in which there is an error (an error code other than **0** is passed into `UR.prepare_response`).
RPC methods are also called using
$.concierge.get_component(component_name). These calls resolve to functions that call GLOB.pers.call. The mapping from components to RPC calls is generally located in content/admin/ui/init.*.

Setup

Setting up the model should typically be done in the Javascript init file. Initial setup is done by GLOB.pers.store.create (called by GLOB.pers.createStore), which specifies which tables should be set up in the store. The first argument is the payload passed to it by the RPC server, since createStore is set as a callback for an RPC call. The second argument is a schema object, which maps a String type_name (used to fetch the table later with get) to a tabledef object. The tabledef object tells the Store how to get the data from the payload and how the tables relate. It can have up to three fields: pFieldName, the name of the field in payload that corresponds to it; obj_type, which can specify a constructor to use; and references, a mapping from String tablename to String target_tablename that specifies when you should be able to reference one table from another. The tablename should be the name of the foreign key (the field which references another table’s primary key).
The targettablename is the table whose primary key is being referenced.

The GLOB.pers.store.add(tablename, objects) call lets you actually store things in the JS database. For each table, you should assemble an object that maps id to an object consisting of all relevant fields, and then pass that in as the objects argument. Later calls to get will query the objects in that table. Calls to add will not delete objects, if you want to replace the objects you should call set instead.

Query

The GLOB.pers.store.get method is used to query the store and return a QuerySet with the relevant methods. The first argument, from, is a string telling which table to fetch the data from. The second argument, where, is an object containing filters that the QuerySet should match. The object is a mapping from field names, to values that the field should have.

A QuerySet represents a set of records that match some query. Supposing we have a QuerySet q, we can loop through the records in q by looping through q.items. You can also use the following methods:
• **q.is_empty():** Returns true if the q contains no records and false otherwise.

• **q.length():** Returns the number of records in q.

• **q.sort(sortfct):** Returns an array of the records in q sorted by the function sortfct.

• **q.min(attr):** Returns the primary key of the record with the minimum value of attr.

• **q.max(attr):** Returns the primary key of the record with the maximum value of attr.

• **q.first():** Returns the first record found in q. This does not imply that q has a particular order.

• **q.values(fieldname):** Returns an object where the properties are the values of the fieldname field for each record in q.

• **q.intersect(ids, field):** Returns a QuerySet representing q with additional filtering. If field is undefined, filters for records whose id is in ids. If field is defined, filters for records whose value for that field is in ids. ids should be an object with keys being the ids filtered for. The value does not matter.
- **q.exclude(where)**: Modifies q so that it no longer includes any records with filter values specified in where. where should be an object where keys are field names and values are values to exclude for that field. Returns q

### Models: Database Schema

Django models correspond to the database schema. We define models and Django gives us a Python interface for interacting with it. Check the [Django Models Documentation](https://docs.djangoproject.com/) for more info about how to interact with it. Here I will focus on explaining what the models mean in terms of the application. My initial explanation will likely be incomplete, contributions are welcome.

**User** - Represents a unique user on NB with a unique email address. Also contains fields for the user’s name, an optional pseudonym, and authorization info.

**Ensemble** - Represents a class set up by an instructor. When an instructor registers, they create an Ensemble, which can optionally contain multiple sections. Students are then registered as members of particular Ensembles that they were invited to, which gives them access
to materials owned by that Ensemble. Also contains fields for a name and description, and options involving access control.

**Folder** - A grouping for materials in a particular ensemble. Folders always belong to an ensemble, and can be contained inside another folder (via the parent field)

**Section** - A grouping of Users in an Ensemble. Has fields for its name and the ensemble it belongs to.

**Invite** - Tracks invites that are sent, what ensemble they are for and who they were sent by.

**Membership** - A mapping between Users and the Ensembles they have access to along with what section they belong to and what levels of access they have to that Ensemble. A User can have Memberships in multiple Ensembles, and Ensembles can have multiple Users.

**Source** - Represents a file that was uploaded. The type field encodes what type of file it represents. The title and submittedby fields contain the title and submitter of that file. There are also fields for the number of pages, and for encoding the position and size of the material.
**YoutubelInfo** - A Source can optionally have a one to one mapping to a YoutubelInfo, if it is a YouTube file. This contains the key that can be used to fetch the video from the YouTube API.

**HTML5Info** - A Source can optionally have a one to one mapping to an HTML5Info, if it is an HTML5 page. This contains the url at which the page can be accessed.

**Ownership** - A mapping between Sources and the Ensemble that owns them. Each Source is owned by only one Ensemble, though there’s no particular reason we couldn’t share documents across ensembles. Also contains fields for the Folder that the object is in, the publishing DateTime, whether the Source is deleted, and assignment information.

**Location** - Represents the Location of a Comment in its Source. Includes position coordinates (x, y) and size information (w, h). Page tells what page of a PDF the comment is on, but in a YouTube video is used to represent the beginning timestamp of the comment. Duration was added recently to handle video comments spanning a time duration. By default it is null, and null values should be handled explicitly in the code. Non-video Sources can ignore this field.
**HTML5Location** - A Location can optionally have a one to one mapping to an HTML5Location, if it is an HTML5 page.

**Comment** - Represents a Comment and is created every time a new Comment is made. It contains fields for the Location, author, the time the comment was written, the text of the comment, and who the Comment should be visible to. It also has a parent field to represent comment threads. The first comment in a thread has a null parent and creates a new Location, and replies to that comment set the previous comment as its parent and share the same Location.

**Tag** - Recently added to support tagging Users in Comments. Currently a mapping between Users and Comments. The type field can allow it to map Comments to other things, like groups of users or subscription topics.

**ThreadMark** - Represents marks that users place on comments, such as stars.

**HOWTOs: Quick Instructions You May Not Know**

**HOWTO: Make Model (Database) Changes**
Since Django 1.7, the ability to make changes to the database are built into Django. To change the models, first make the desired changes to `apps/base/models.py`, then run the following commands from the apps directory:

```
./manage.py makemigrations
./manage.py migrate
```

Make sure to check the migration file that is created by makemigrations into git!

**HOWTO: Run The Server Constantly**

From the `apps` directory, run the following command, replacing `<my-port>` with your chosen port:

```
nohup ./manage.py runserver 0.0.0.0:<my-port> &
```

The server will automatically update its code when you make changes to it. Python changes are applied immediately. Javascript changes require that the code is compiled by grunt before changes will appear. If you are having trouble seeing Javascript changes, try clearing your browser's cache.

**HOWTO: Run Jobs on the Server**
Jobs are run from a CLI, so they can be scheduled with CRON. From the `apps` directory, run the following command, replacing `<ACTION>` with the command that you wish to execute:

```
python base/jobs.py <ACTION>
```

If you wish to see the actions listed, simply leave out the action.

There are also `jobs.py` scripts in other apps besides `base` that can be accessed similarly. New jobs can be defined directly in the file by placing a mapping from the action name to the function in ACTIONS at the bottom of the file. You MUST actually be in the `apps` directory due to the way the script sets the Python Path. It will probably fail if you run it from another directory.

**HOWTO: Add New Options**

The way options are handled depends on whether it is a user-level or ensemble-level option. Ensemble options are set by admins of an ensemble and apply to the entire ensemble. These are easier to understand, to add an option you add a field to the Ensemble model.

The generation of the forms that allow admins to set these options are handled by the Django ModelForm class. Be sure to define a default
value and a verbose_name, which is the description of the option that will be visible to the admin setting the option.

User settings are more complex. Rather than being fields that a user sets, they are actual items in the database. This means that to add user level options you need to manually add it to the DB or write a command line tool that does it in Django (the latter is probably easier if you are not experienced with databases). The 3 models involved are:

1. DefaultSetting - Defines the option and its default setting. The name field should match the id_item of the select element in the template that controls it. The value should be the value of the default option. The description may be null, and if it is used I have not found where.

2. SettingLabel - Defines each choice available to the user. The setting field should point to the DefaultSetting it is an option for. The value should be a unique integer identifying that option. The label is the text that will appear to the user selecting the option.

3. UserSetting - Unlike a DefaultSetting and a SettingLabel, a UserSetting does not need to be manually entered into the DB to create a new option. A UserSetting represents a user selected
override to a DefaultSetting. When checking for a user’s option, you should check if a UserSetting exists for that user and option. If it does, the one with the latest time should be used to access the value of the option. If it doesn’t, the DefaultSetting should be used instead.

Everything is Broken! (Debugging Tips)

• This may sound obvious but the first thing you should check is the Javascript console printout. Feel free to add debug messages using `console.log()`.

• If things are broken and there is no sign of it in the Javascript console, the error is likely in the Python code.

• You can find Python stack traces and error printouts in `/tmp/`. Consider searching through the logs (which don’t have human-understandable names) using `grep`. The command `grep -r "my_search_here" /tmp/` will do that. Consider searching for the name of the function that you were editing when things broke.

• Similarly, the command `grep -r "my_search_here" ./` will search the current directory and subdirectories. Try using this command to look for functions defined in other files.
• Edit one thing at a time and then check to make sure everything is still working. If you change a lot of different parts at once and then something breaks, it might be difficult to figure out exactly what went wrong. This has caused me to redo decent amounts of work.

Appendix C: Django Model Changes

Below are the Django model objects that were modified, with added fields and objects highlighted:

class Comment(models.Model):
    TYPES = ((1, “Private”), (2, “Staff”), (3, “Class”), (4, “Tag Private”))
    location = ForeignKey(Location)
    parent = ForeignKey(‘self’, null=True)
    author = ForeignKey(User)
    ctime = DateTimeField(default=datetime.now)
    body = TextField(blank=True, null=True)
    type = IntegerField(choices=TYPES)
    signed = BooleanField(default=True)
    deleted = BooleanField(default=False)
    moderated = BooleanField(default=False)

class Location(models.Model):
    source = ForeignKey(Source)
    version = IntegerField(default=1)
    ensemble = ForeignKey(Ensemble)
    section = ForeignKey(Section, null=True)
    x = IntegerField()
    y = IntegerField()
    w = IntegerField()
    h = IntegerField()
    page = IntegerField()
    duration = IntegerField(null=True)
    is_title = BooleanField(default=False)

class Tag(models.Model):
TYPES = ((1, “Individual”),)
type = IntegerField(choices=TYPES)
individual = ForeignKey(User, null=True)
comment = ForeignKey(Comment)
last_reminder = DateTimeField(null=True)
Bibliography


