Harvard Lecture Helper

Design Document

Developed by Adam Atanas, Promit Ghosh, Wei-Te Mark Ting

Contents

Updated as of December 2014. Harvard Lecture Helper version 0.7.0 (beta)

Powered by Summernote and CS50!

Adam Atanas

Video Manipulation

The video manipulation part of the project uses content scripts (Javascript that is injected into the actual webpage, rather than running inside the Chrome extension) to obtain and manipulate videos on the webpage.

One key aspect was site-to-site variation. YouTube has its own built-in timestamp mechanism (adding an &t=TIME into its URL), so the video manipulator code takes advantage of it. However, most other sites don't offer such a convenient feature, so we had to implement a different set of code for them.

Video Pauser

The video pauser (Get Video Time) works by finding a list of all video tags on the current webpage, then chooses the one that's currently playing (unless we're on YouTube, in which case it chooses the only available video). It then proceeds to pause that video and obtain its timestamp via the HTML5 API, and pop up a message with a URL to resume playback of that video. The URL will consist of the page URL, plus some extra variables. For YouTube, it adds an &t=TIME, where TIME is the time it obtained previously.

A major challenge we faced was attempting to somehow encode the timestamp of a non-YouTube video in such a way that playback could be re-enabled. The URL needed to be copy/paste-able into the notepad, as one of the main motivations of the project was to take notes and insert URLs into them that correspond to various positions in the lecture videos, to avoid having to fumble around with the controls to get the video at the right place (which is especially troublesome with longer videos). Thus, attempting to inject JavaScript that would somehow resume playback was impossible, since the user might access the URL from anywhere (not necessarily the notepad), and there might be many different websites with many different constantly-changing timestamps. What we eventually settled on was adding two extraneous variables (playbackTime and videoSource) to the GET request in the browser (which, being a substring of the URL, will be available to the Video Resumer method, described below). We chose variable names that we thought were unlikely to be used on real websites, so we thus minimized the chances of actually breaking the website. These two variables encoded values for the timestamp and the source of the video to be played back (which was necessary in case there were multiple videos on the webpage; we need to know which one to play back).

Video Resumer

The video resumer (Resume Video) then resumes playback of the video at the timestamp encoded in the URL. YouTube automatically begins replaying the video at the correct time, but other websites won't and the video resume is required to do so. Even so, video resume still works on YouTube, where it simply looks for the t= flag and resumes playback at that time. Originally, this wasn't necessary, but if CS50 2X is run it can reset the timestamp, in which case being able to press Resume Video to get it back is very helpful. For other websites, it parses the URL generated by the video pauser (in particular, it gets the values for the two extra variables in the GET that video pauser inserted). It then gets the video to resume by searching through all video elements on the webpage, and getting the one whose source matches the videoSource variable specified in the GET request. It then sets the time of that video to

playbackTime, the other variable specified in the GET request (which, remember, is a substring of the URL).

Note that this means the two programs only work if the video is in the form of an HTML5 video element. This is where CS50 2X comes in, as it can transform a variety of video players (in particular, JWPlayer, which is featured on isites) into FlowPlayer, which uses the HTML5 video tag and is hence accessible to the two functions above. These programs should thus work on most websites in which CS50 2X works, and will likely not work if CS50 2X doesn't.

A major challenge I faced were attempting to hook into the JWPlayer API. After long hours of fumbling around the API, I eventually realized it was not possible due to security practices. This is when I realized we needed CS50 2X in our project, since it can change JWPlayer into FlowPlayer, which my code can successfully integrate with. At first I tried cross-extension calling (i.e. calling CS50 2X from within Harvard Lecture Helper), but CS50 2X did not give me the permissions in its manifest.json file in order to do this. This led to the decision to bundle CS50 2X directly into Harvard Lecture Helper, which has since worked perfectly. A huge thanks to David Malan, who let us bundle the source code for CS50 2X into our extension. Without him, this project would be impossible.

Promit Ghosh

I was working to implement a feature that would allow the user to save their notes directly to Google Drive. However, there were a multitude of issues I faced. For one, I never had any experience with OAuth or the Google Drive API, and spent a lot of time familiarizing myself with Google's developer APIs as well as how OAuth worked. I spent a lot of time on stackoverflow.com and on Google's various developer help web pages to familiarize myself with how the process worked. I set up a developer account and enabled a client ID for web application. Through a combination of code I gathered from forums and help pages, and code I wrote myself, I created a html file that I ran using the web server from pset 7 to test if I had a functioning authentication check, as well as testing logging in, checking if the user is logged in already, logging out, and manually uploading (ie choosing a file then clicking the upload button) it to Google Drive.

Next, I had to pass in the note file the user would create while using the extension to a JavaScript function that would check if the user was logged into Google Drive, check the OAuth, then upload the file. This is the part that became an unassailable mountain. I searched through multiple help forums and Google Drive development help pages, but couldn't find anything I would be able to use to automatically pass the note file the user had created to the upload function of Google Drive. While checking for login status and logging out worked fine as well as the authentication for the extension, the file upload could not be uploaded due to the limitations of our file system and that we were making an extension and not a Chrome web store app. Even after spending over 20 hours doing research and searching through stackoverflow.com and other coding sites, I wasn't able to find a way to overcome this hurdle, and Google Drive integration was discarded as an option.

Wei-Te Mark Ting

Distribution of Workload

From the offset, this project was going to be at least a two person project: so distribution of the coding as well as versioning was going to be a problem. We originally wanted to create a shared folder on Dropbox, but lack of fine-grained version history and rather imprecise conflict resolution led us towards another prospect: GitHub. Two disastrous master branch commits and recoveries later (we ended up needing to reset our branches and authenticated IDs to fix the mishap), we were well on our way to developing our project.

App or Extension?

Initially, we waffled back and forth between creating a Chrome packaged application and a Chrome browser extension. Halfway through looking through the extension requirements, and building two or three manifest file versions, we saw that there were many note editors—but none of them (or very few) were rich text editors AND extensions. We weren't sure why at first, but during the last few weeks, the reason became very evident. The lack of file management APIs available to purely JavaScript and HTML were abundantly clear. The few easily pluggable ones that were available (such as Google's Save to Drive) had limited file targets, and none could interact with dynamic content stored in temporary `innerHTML` or were simply broken (like Dropbox's Dropins API).

Ultimately, because the core functionality of our project was to pause/play videos with time-stamped URLs, we elected to create an extension to have content-script functionality (as well as easier integration with 2x—prior to embedding it within our code entirely).

Summernote: Notes for Thought

The idea to create hyperlinked timestamps, or even just get a URL to control video playback was supplemented by the notion that we could create a rich-text notepad. It would be quite useful to have all the tools at your disposal after all. So, looking for ways to store formatted text and cleverly-disguised `href`'s, I found the `editableContent` tag standard. Unfortunately, there was no easy way to generate such hyperlinks, so we opted to implement some form of text editor instead. As a result, we found Summernote, and though it was not the first WYSIWYG in-line editor based on Bootstrap that we found, it was by far the most applicable to our situation: it had all the necessary API and functions documented, and was just as easy to plug in as they had advertised it to be.

Summernote allowed easy access to its editor content through the use of the `.code()` function call and jQuery selectors. Furthermore, it was easily customizable and able to open and close instances on a whim, allowing for more (semi) permanent storage of dynamic HTML content in a browser session.

Chrome and the Cloud

While we originally thought it would be ideal to have cloud integration, Adam and I were busy working on the video detection and notepad respectively. Several weeks in, about right after the proposal was submitted, Promit inquired about joining our team. This granted us the leeway to explore potential cloud integration options. He was able to research almost everything from OneDrive to Google Drive and several other more obscure storage solutions. Though he tried until the very end, due to the nature

of Chrome's content security policies as well as the lack of PHP in Chrome extensions, we were met with several unrelenting impasses that prevented any cloud storage.

Without relaxing security standards, Chrome's CSP prevented use of external JavaScript files, meaning that all of our APIs (such as jQuery) were bundled internally with our extension.

File (System) or Lack Thereof

A text editor without the ability to save is pretty useless—particularly if it "forgets" its contents if the window is closed, so we looked for ways to store the data on a user's file system. Putting it nicely, JavaScript doesn't play nice with accessing files. As a result, we tried cookies and local compact databases, but ultimately, it was only through fairly recent HTML5 proposed standards and jQuery/AJAX that we were able to successfully implement local storage (albeit with some documented limitations).

By parsing the extension ID (`extID`) from Chrome's API, we were able to create a consistent identifier token for the `localStorage` using HTML5's API. This allowed consistent recall and synchronization between browser windows and persistent storage for user sessions.

CS50 2x—Inspiration and Solution

Originally, the idea for this app was inspired by CS50 2x. If you could speed up a video at will beyond its normal playback limitations, why not control playback further? Why not create JavaScript content hooks that allowed retrieval of more detailed playback information? Alas, the solution (nay, the problem) was not so simple. We looked into CS50 2x's source code and discovered that it wasn't a straightforward hook-and-play that we had believed it to be. Instead, we saw that 2x extracted the playable content and used FlowPlayer as an alternative media wrapper to control playback.

During the Hackathon, we spoke to David Malan, who generously allowed us to include the entirety of the CS50 2x extension, allowing us to build off of the FlowPlayer wrapper.

By allowing us to include all of CS50 2x's functions and use it as a pre-requisite, we were able to guarantee that we had a unified platform to work off of, as well as making it possible to interact using the FlowPlayer API (as JWPlayer's API is not exposed to external calls).