CSCD01 Deliverable 4

Team 18: Sin Chi Chiu, Jason Fong, Jeremy Lai, Mark Padilla, Gavin Zhang

Table of Contents

Implementation of Issue 6706 - Class and ID Conflicts

Implementation Plans (Issue 6706)

Based on our understanding of the issue, we originally wanted to implement a direct solution in case there were conflicting class names and ID elements in the DOM. Because **viewer.js** would generate a configuration object on load that details all HTML elements to be made, we would implement a fix that prefixed **pdfjs** to the class name and ID's of all the elements. To ensure default functionality, a separate **viewer.html** and **viewer.css** would be created with the prefix text in the case of wanting to embed pdf.js on other websites, with the default available to preserve its current operating environment.

Web developers who wanted to embed PDF elements to their website would then use the modified html and css documents, which we detailed in Deliverable 3 as **viewer-prefixed.html** and **viewer-prefixed.css**. Using the suggestion from this Stack Overflow thread https://stackoverflow.com/a/21253793 we would use a **script** tag to pass a boolean parameter into **viewer.js**. This boolean parameter, a logic switch, would tell the function, **getViewerConfig** to generate a configuration object with or without the prefixed string, again as detailed in Deliverable 3.

We then saw a slight design issue with this method; we would be modifying a section of code that would already handle a key function of loading the viewer. Configuration is a key preloading task and we sought to abstract and formalize this. To correct this we decided to modify the solution as follows:

1. Create a new web layer module, a new script called **viewer-config.js**. This script will be a controller script to handle configuration options for loading the viewer.
2. Move the functionality of **getViewerConfig** to **viewer-config.js**.
3. Use a mode/settings file **viewer-config.json** to indicate the mode or options wanted when generating the configuration object
4. Pass object to **viewer.js** and **app.js** to initialize as normal

This solution abstracts configuration into a new module that can then be modified and added upon in the future. A controller for configuration would be able to handle all aspects of it for the web layer. Aside from the HTML and CSS files, developers would only need to modify the JSON file to indicate what specifications they want when embedding the viewer. This makes the logic switch more robust. The end result configuration object will be the same as the current implementation, and as such, shouldn't break how the viewer is currently loaded.

Automated Testing (Issue 6706)

Before we had to switch issues, some initial experimentation was done with automating unit tests:

Using the Jasmine unit testing framework we initially thought to ensure that the ID's matched their expected values with the following idea for a unit test:

```
describe("logicSwitchTest", function() {
  it("check logic switch on prefixed/nonprefixed DOM IDs", function(done) {
    const documentElements = getViewerConfiguration();
    documentElements.toolbar
    expect(documentElements.toolbar.container.id).toEqual("toolbarViewer");
    expect(documentElements.toolbar.print.id).toEqual("print");
  });
});
```

The idea behind this experiment was that it would call the getViewerConfiguration function and obtain the document and check if each element's ID field was the correct value.

Complications arose when we found that the unit tests do not utilize the front end at all so the getViewerConfiguration function returns a set of undefined values. We could potentially create a dummy front end for use in the unit tests but upon further consideration determined this to be an inefficient use of our time. So we abandoned automated testing for this feature.

Manual Testing (Issue 6706)

Since Issue #6706 deals with styling and the appearance of the whole pdf.js viewer, we had planned to use manual testing to ensure that the feature's changes don't alter the look or feel of the interface. While automated tests could be used to ensure that the CSS classes are identical, it is more critical, to the end-user, that the final interface produced still looks and behaves the exact same way they would expect it to.

For this reason, our implementation of the acceptance test suite would use simple web pages that embed a pdf.js viewer. There will be three such web pages, each corresponding to a test case, all located under the pdf.js subfolder test/6706.
The first two cases involve a web page that does not have any namespace conflicts with pdf.js' CSS namespace. These will be used to ensure that the previous behaviour of pdf.js was not broken by the feature changes. The third and final test case involves a web page that has CSS classes and IDs conflicting with those used by pdf.js. This will be used to ensure that the feature changes correctly prefix the namespace identifiers and thus avoid styling issues.

Test cases: Namespace classes and ids

Goal: Ensure that when PDF.js is embedded into a web page, that its CSS classes and IDs do not conflict with others on that page.

Test Case 1: Control (Disabled, No Namespace Conflicts)

1. In test/6706/1, open index.js in an editor.
2. Find the location where 'PDFJS.usePrefixedSelectors' is set.
3. Set PDFJS.usePrefixedSelectors = 0.
4. Launch test/6706/1/index.html in a browser.
5. Wait for pdf.js to finish loading.
6. Verify that there are no styling issues in the PDF viewer's interface:
    a. Toolbar is visible at the top of the page
    b. Toolbar icons are visible
    c. Overflow menu on the top-right opens and its contents are fully visible
    d. When the sidebar (top-left) is clicked, the sidebar appears with all of its contents and the viewer's canvas is properly resized to accommodate the sidebar
    e. Clicking the search button (magnifying glass) opens a small dialog with all search inputs fully visible

Keep this viewer open for the remaining two tests.

Test Case 2: Enabled, No Namespace Conflicts

1. In test/6706/2, open index.js in an editor.
2. Find the location where 'PDFJS.usePrefixedSelectors' is set.
3. Set PDFJS.usePrefixedSelectors = 1.
4. Launch test/6706/2/index.html in another browser tab or window separate from the one used for Test Case 1.
5. Wait for pdf.js to finish loading.
6. Verify that the user interface is identical to the one from Test Case 1.

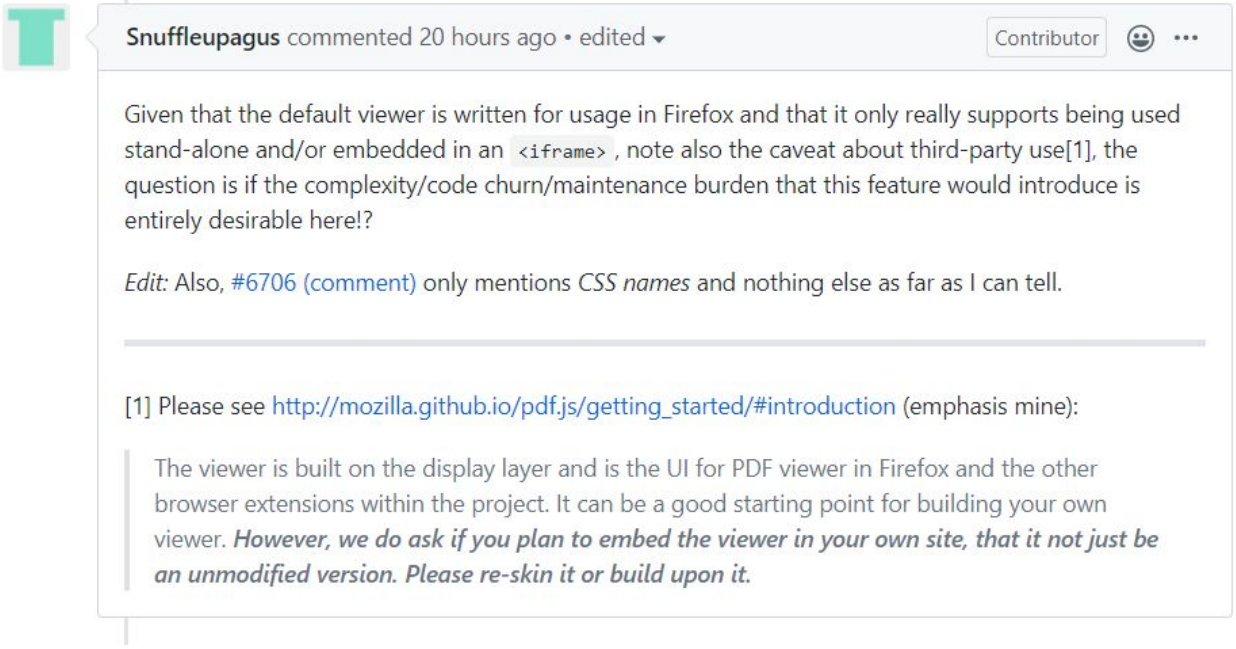
Test Case 3: Enabled, Namespace Conflicts Exist

1. In test/6706/3, open index.js in an editor.
2. Find the location where 'PDFJS.usePrefixedSelectors' is set.
3. Set PDFJS.usePrefixedSelectors = 1.

4. Launch test/6706/3/index.html in another browser tab or window separate from the one used for Test Case 1.
5. Wait for pdf.js to finish loading.
6. Verify that the user interface is identical to the one from Test Case 1.

Complications (Issue 6706)

Unfortunately there were complications during the development process. When we attempted to confirm that the fix we had designed (but luckily not yet fully implemented) was considered desirable by those in charge of the project we received this response:



Snuffleupagus commented 20 hours ago • edited Contributor

Given that the default viewer is written for usage in Firefox and that it only really supports being used stand-alone and/or embedded in an `<iframe>`, note also the caveat about third-party use[1], the question is if the complexity/code churn/maintenance burden that this feature would introduce is entirely desirable here!?

Edit: Also, #6706 (comment) only mentions *CSS names* and nothing else as far as I can tell.

---

[1] Please see http://mozilla.github.io/pdf.js/getting_started/#introduction (emphasis mine):

> The viewer is built on the display layer and is the UI for PDF viewer in Firefox and the other browser extensions within the project. It can be a good starting point for building your own viewer. ***However, we do ask if you plan to embed the viewer in your own site, that it not just be an unmodified version. Please re-skin it or build upon it.***

According to this, PDF.js is designed to operate as Firefox's PDF viewer or in an iframe (in which case it would be completely separate to everything else on the page). The introduction to the project states that if you intend to use PDF.js as part of another application you should modify it yourself. As such, the onus is on the third-party developer to resolve any conflicts.

So unfortunately for us, the fix we have planned would not be required nor accepted by those responsible for PDF.js.

Switching Gears (Issue 3471)

Fortunately for us, we did have an extra feature to fall back on: the copy protection feature we examined in deliverable 3.

Implementation of Issue 3471 - PDF.js Copy Protection

Feature Explanation

PDF documents have had permission settings ever since it was introduced. Some of these permissions allow document creators to set whether or not to allow readers to print or copy contents. While these permissions can easily be removed by third party applications, enterprise environments still rely on PDF readers to respect these permissions in order to comply with company policies. The lack of support in PDF.js to respect these permissions drives many enterprise users away from using Firefox as their choice of browser, and forces them to use alternative solutions such as chrome.

The copy protection feature was requested 7 years ago, and was implemented for a short period of time. The feature was quickly removed due to the conflict of interest between the Mozilla developers and the usefulness of the feature. However the feature is still being requested by many enterprise users. The Mozilla developers decided to make this feature optional to PDF.js, where system admin can set to turn on this feature in Firefox settings. Due to the change in requirements (and change in the overall architecture of the code base), this feature will require a complete redesign.

This copy protection feature works as follows: when Firefox is set to respect PDF user access permissions, PDF.js will check if the document has any permission set. If the document is set to be copy protected, users will **not be able to copy text** from the document. Users are still allowed to **search** and **select/highlight text** within the document. The search function is still useful for searching within the document and highlight text is useful for demonstration/ presentation purposes.

User Guide

Since this feature is intended to be optional in Firefox, it is disabled by default. To enable this feature for use in environments outside of Firefox default PDF viewer, the value of *respectPermissions* option under *web/app_options.js* must be set to true manually.

app_options.js web ●

web > app_options.js > defaultOptions > respectPermissions

```
151   respectPermissions: {
152     value: true,
153     kind: OptionKind.VIEWER + OptionKind.PREFERENCE,
154   },
```

Once enabled, readers will not be able to copy content from the viewer, but still be able to select and search within the document.

Preferred Fix and Alternate Fix

The preferred fix by those in charge of PDF.js was to add an option to the viewer and check for the permission flags in the PDF document to see whether or not to disable the HTML text layer altogether.

However, most of the other PDF viewers allow text selection along with copy protection, and the preferred fix will introduce 2 problems:

1. User will not be able to select text
2. Search will not function properly (search will jump to the page where the text is found, but will not highlight the text)

Due to the preferred fix not adhering to the PDF standard and causing search to malfunction, we decided to implement an alternate fix that allows users to select text and keeps search function working correctly. In fact, the feature of allowing text selection on copy protected PDF documents has been requested multiple times (when copy protection was implemented) but due to the lack of development on PDF access permissions, neither copy protection nor text selection along copy protection have been implemented properly.

Preferred Fix

The preferred fix is simpler than the alternate solution. First, we need to check the option of whether or not PDF.js needs to respect the permission. Mozilla developers believe that PDF permission should be optional and not mandatory, thus we will need to add an option in the viewer preference that can be set by Firefox browser. If the viewer is set to respect permissions, then it will check the permission flags on the PDF document. If we find the flag that indicates that the PDF is copy protected we must send that information through the system back to the viewer to disable copy. When the viewer receives this information it will disable the HTML text layer such that no text can be selected from the document. This is illustrated below:

Alternate Fix

The alternate fix would be an extension of the preferred fix. Instead of disabling the text layer once we get the permission flag from the backend, we will create a new type of text layer. Currently PDF.js has 3 options for text layer: disable, enable, enhanced enable. The preferred fix would simply use the “disable” option for the text layer, but as mentioned above it introduces problems.

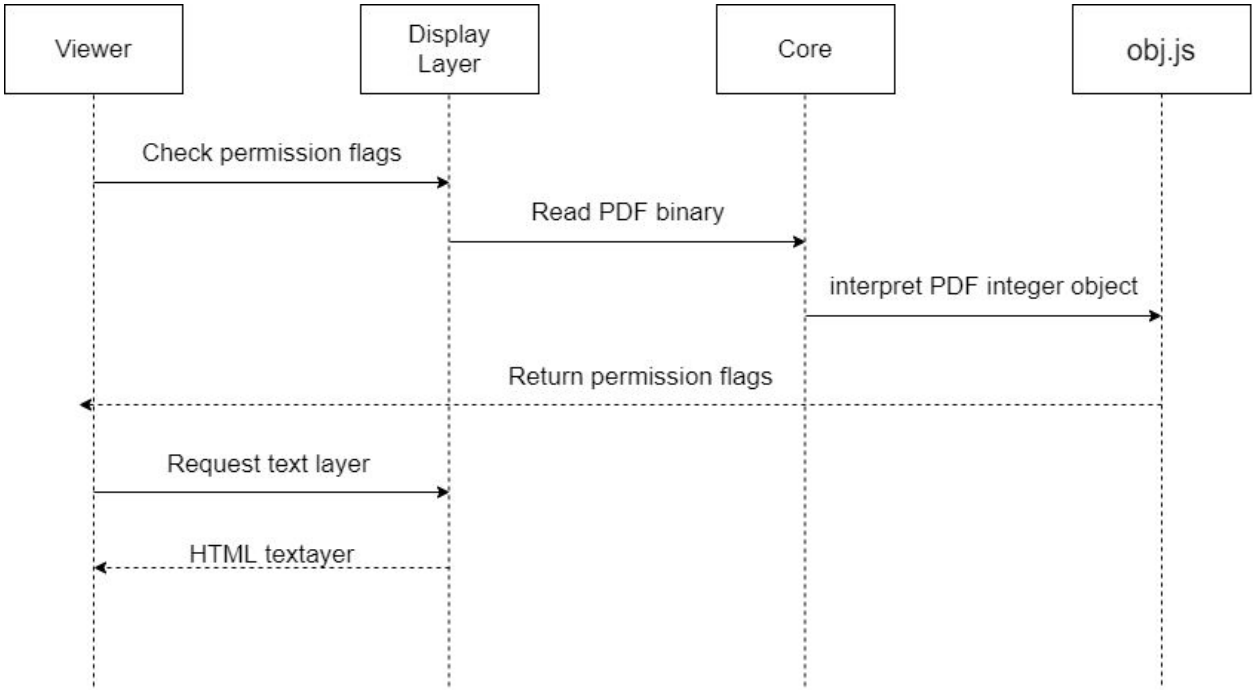
The new text layer would allow the selection of text in the PDF, but disable the ability to copy it. This means that the text layer would be active so that highlighting is available. The usual ways to copy highlighted text, is either from keyboard input or context menu to select the copy option. We can prevent the typical copy methods by catching and modifying the event for copy, and preventing the default behaviour. From there depending on the permissions flag, we can specify how pdf.js should handle the copy function.

Implementation

The copy protection feature is implemented in two parts: reading the permission flags from the PDF binary, and a new text layer that disables copy functionality.

Reading the permission flags

The design of reading permission flags follows the same design used for getting document information in PDF.js. To check permission flags, during viewer startup it will send a request to the display layer (API layer). The display layer will then read the binary and extract the permission flags in the core layer. The permission flags are stored as PDF integer objects. Per PDF specification, this can be interpreted as binary values in a signed two's-complement form, and the position of the bits indicate which permission is set (based on section 7.6.3.2 and Table 22 of the PDF specification[1]). The interpretation of the PDF integer object will be done in obj.js, and the interpreted value will be returned back to the viewer. Once the viewer gets the flags, it will request the appropriate text layer from the display layer. The following would be the sequence diagram of the design:



Upon implementing this feature in the core layer, we noticed there is already code implemented for reading permission flags, and is not used. However, we were not able to find when or any documentation regarding the implementation. The implemented code follows the diagram above and is in the same design as other functions that request document data. This left us with only the viewer to work on.

The changes needed in the viewer are rather minimal. First, we added a new preference in *app_options.js* that can either be set manually by changing the code in file or set by the Firefox browser. The default value for respect permissions is false. The viewer's main thread (web/*app.js*) will check for the preference when loading the document, and only if it is set to true, the viewer will request the API to get the permission flags. When the flag is returned, it will check the two's complement of the returned value and see if copy protection is enabled. If copy protection is enabled, it will override the default text layer from "enable" to "disable". Initial testing was done on this to ensure the mechanism is working correctly, before we move on to implement a new text layer.

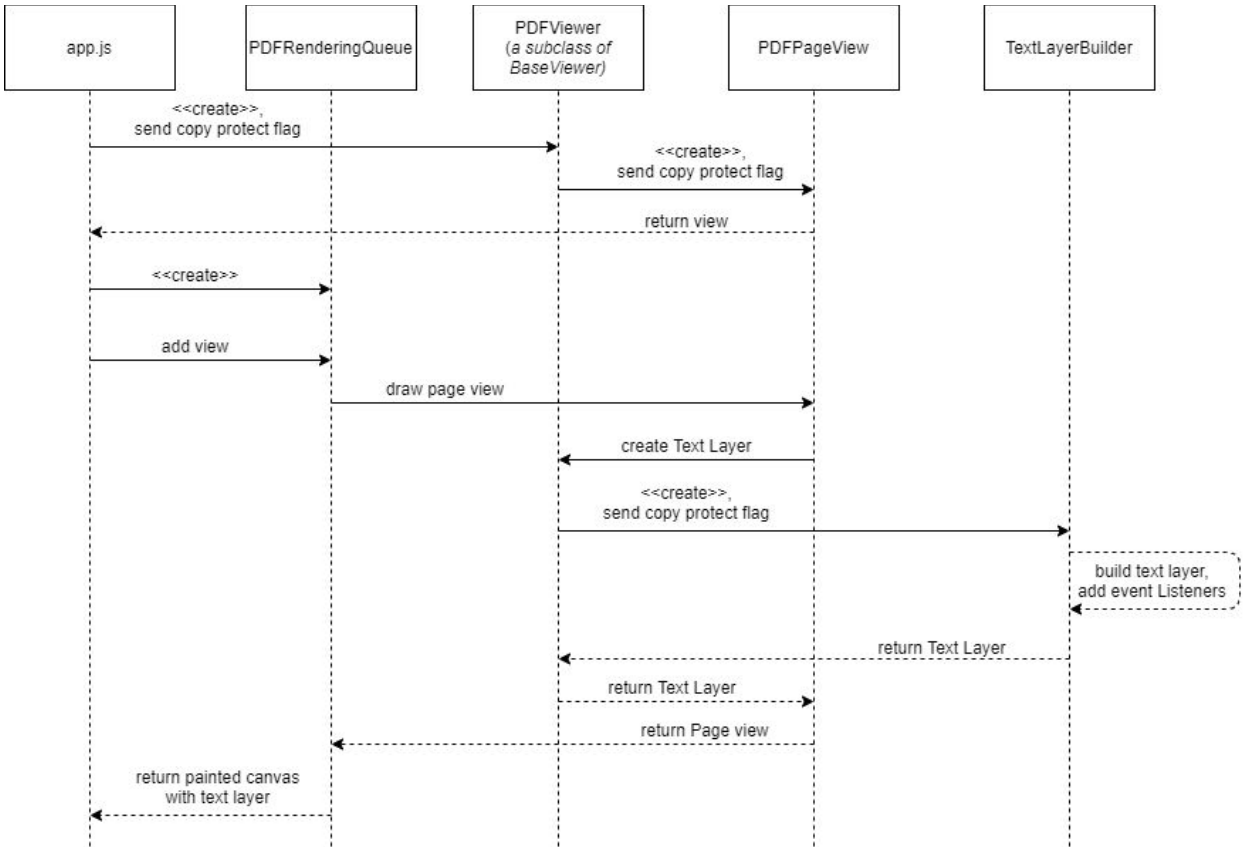
New text layer

In the text layer itself (in the View layer), we can disable various ways users can copy data without affecting their ability to select text and search for text with Ctrl+F. We do this by leaving the text layer enabled, and by adding event handlers for events such as "copy", "cut", "dragstart" (for drag and drop), and "contextmenu", and disabling the default action. This will prevent users from being able to copy, cut, or drag and drop selected text from copy protected documents.

The event handlers for “copy” and “cut” in particular are newer, and while they have had long-time Firefox support, these event handlers in other browsers were added more recently, with Chromium browsers adding support in May 2017.

These event handlers are only added to web/text_layer_builder.js under the condition that the document is copy protected. In implementing the changes to the text layer, after checking the viewer’s main thread in (web/app.js) that a document should be copy protected, most of the work involved was sending this property of “enableCopyProtection” through the various classes inside of the View layer, to reach the TextLayerBuilder.

The sequence diagram below shows how after app.js decides whether the copy protection permission should be used, this condition is sent into the text builder so the event listeners can be used to disable copying actions.



Acceptance Test Suite for Issue 3471

Manual Testing over Automated Testing

Unfortunately, while PDF.js does contain an extensive Jasmine test suite for testing its core functions, it does not have anything set-up for automating front end testing. We had looked into selenium as a potential option for automating front end testing. We decided against this for three reasons. One, none of us are particularly familiar with Selenium. Two, Selenium is not already part of the project and we do not believe we could integrate it in the time we have left due to configuration and having to get approval from project maintainers. Three, although Selenium is designed to automate front end testing, Selenium's current capabilities still make it difficult to simulate selecting and copying text out of the page.

Based on these issues, we decided to use manual testing for all acceptance tests. We believe this is the best way to test Issue #3471, because highlighting text, copying it, and pasting are very simple and quick operations for a human user.

Goal

To ensure that copy-protected PDF files, when opened in PDF.js, cannot have their text copied.

General Setup

1. From our PDF repository https://github.com/jfong701/test take all PDF files with the "Issue3471" prefix and save it locally.

Test Cases

Preferred Fix - Text Selection Disabled

Non-Protected PDF, followed by Protected PDF

1. Open a new instance of the pdf.js web viewer in a browser tab.
2. In the viewer, use the 'Open File' button to load "Issue3471NonProtected.pdf".
3. Once the PDF loads, highlight a few lines on the first page.
4. Copy the highlighted text, either by using Ctrl+C or the right-click context menu.
5. Open a text editor and paste the copied text.
6. **Verify** that all of the highlighted text was successfully pasted.
7. In the same viewer instance (same browser and tab), use the 'Open File' button to load "Issue3471Protected.pdf".

8. Once the PDF loads, attempt to highlight the line of text on the page.
9. Verify that the text cannot be highlighted.

Protected PDF, followed by Non-Protected PDF

1. Open a new instance of the pdf.js web viewer in a browser tab.
2. In the viewer, use the ‘Open File’ button to load “Issue3471Protected.pdf”.
3. Once the PDF loads, attempt to highlight the line of text on the page.
4. **Verify** that the text cannot be highlighted.
5. In the same viewer instance (same browser and tab), use the ‘Open File’ button to load “Issue3471NonProtected.pdf”.
6. Once the PDF loads, highlight a few lines on the first page.
7. Copy the highlighted text, either by using Ctrl+C or the right-click context menu.
8. Open a text editor and paste the copied text.
9. **Verify** that all of the highlighted text was successfully pasted.

Note that the test cases below only apply to the alternate fix and not the preferred fix.

Alternate Fix - Disable keyboard copy, context menu, and drag&drop

“Respect Permissions” toggled off

1. In web/app_options. js, set “value” inside “requestPermissions” to **false**.
2. Open a new instance of the pdf.js web viewer in a browser tab.
3. In the viewer, use the ‘Open File’ button to load “Issue3471NonProtected.pdf”
4. Once the PDF loads, highlight a few lines on the first page.
5. Copy the highlighted text by using Ctrl+C.
6. Open a text editor and paste the copied text.
7. **Verify** that all of the highlighted text was successfully pasted.
8. In the viewer, highlight a different part of the text.
9. Copy the highlighted text by using the right-click context menu and choosing “Copy”.
10. Open a text editor and paste the copied text.
11. **Verify** that all of the highlighted text was successfully pasted.
12. In the viewer, right-click on the highlighted text.
13. In the context menu, select “Search Google for ‘<highlighted text>’”.
14. **Verify** that the browser opens a Google search for the highlighted text in a new tab.
15. In the viewer, drag and drop the highlighted text onto the tab bar.
16. **Verify** that the browser opens a Google search for the highlighted text in a new tab.
17. In the same viewer instance (same browser and tab), use the ‘Open File’ button to load “Issue3471Protected.pdf”.
18. Once the PDF loads, highlight the line of text on the page.
19. Copy the highlighted text by using Ctrl+C.
20. Open a text editor and paste the copied text.
21. **Verify** that all of the highlighted text was successfully pasted.

22. In the viewer, highlight the text again.
23. Copy the highlighted text by using the right-click context menu and choosing “Copy”.
24. Open a text editor and paste the copied text.
25. **Verify** that all of the highlighted text was successfully pasted.
26. In the viewer, right-click on the highlighted text.
27. In the context menu, select “Search Google for ‘<highlighted text>’”.
28. **Verify** that the browser opens a Google search for the highlighted text in a new tab.
29. In the viewer, drag and drop the highlighted text onto the tab bar.
30. **Verify** that the browser opens a Google search for the highlighted text in a new tab.

Non-Protected PDF, followed by Protected PDF

1. In web/app_options. js, set “value” inside “requestPermissions” to **true**.
2. Open a new instance of the pdf.js web viewer in a browser tab.
3. In the viewer, use the ‘Open File’ button to load “Issue3471NonProtected.pdf”
4. Once the PDF loads, highlight a few lines on the first page.
5. Copy the highlighted text by using Ctrl+C.
6. Open a text editor and paste the copied text.
7. **Verify** that all of the highlighted text was successfully pasted.
8. In the viewer, highlight a different part of the text.
9. Copy the highlighted text by using the right-click context menu and choosing “Copy”.
10. Open a text editor and paste the copied text.
11. **Verify** that all of the highlighted text was successfully pasted.
12. In the viewer, right-click on the highlighted text.
13. In the context menu, select “Search Google for ‘<highlighted text>’”.
14. **Verify** that the browser opens a Google search for the highlighted text in a new tab.
15. In the viewer, drag and drop the highlighted text onto the tab bar.
16. **Verify** that the browser opens a Google search for the highlighted text in a new tab.
17. In the same viewer instance (same browser and tab), use the ‘Open File’ button to load “Issue3471Protected.pdf”.
18. Once the PDF loads, highlight the line of text on the page.
19. With this line still selected, hit CTRL-C.
20. **Verify** that this text cannot be pasted elsewhere.
21. Right-click on the highlighted text.
22. **Verify** that no context menu appears.
23. Attempt to drag and drop the text onto the browser’s tab bar.
24. **Verify** that the browser does NOT open a Google search for the highlighted text in a new tab.

Protected PDF, followed by Non-Protected PDF

1. In web/app_options. js, set “value” inside “requestPermissions” to **true**.
2. Open a new instance of the pdf.js web viewer in a browser tab.
3. In the viewer, use the ‘Open File’ button to load “Issue3471Protected.pdf”.

4. Once the PDF loads, highlight the line of text on the page.
5. With this line still selected, hit CTRL-C.
6. **Verify** that this text cannot be pasted elsewhere.
7. Right-click on the highlighted text.
8. **Verify** that no context menu appears.
9. Attempt to drag and drop the text onto the browser's tab bar.
10. **Verify** that the browser does NOT open a Google search for the highlighted text in a new tab.
11. In the same viewer instance (same browser and tab), use the 'Open File' button to load "Issue3471NonProtected.pdf".
12. Once the PDF loads, highlight a few lines on the first page.
13. Copy the highlighted text by using Ctrl+C.
14. Open a text editor and paste the copied text.
15. **Verify** that all of the highlighted text was successfully pasted.
16. In the viewer, highlight a different part of the text.
17. Copy the highlighted text by using the right-click context menu and choosing "Copy".
18. Open a text editor and paste the copied text.
19. **Verify** that all of the highlighted text was successfully pasted.
20. In the viewer, right-click on the highlighted text.
21. In the context menu, select "Search Google for '<highlighted text>'".
22. **Verify** that the browser opens a Google search for the highlighted text in a new tab.
23. In the viewer, drag and drop the highlighted text onto the tab bar.
24. **Verify** that the browser opens a Google search for the highlighted text in a new tab.

Software Development Process

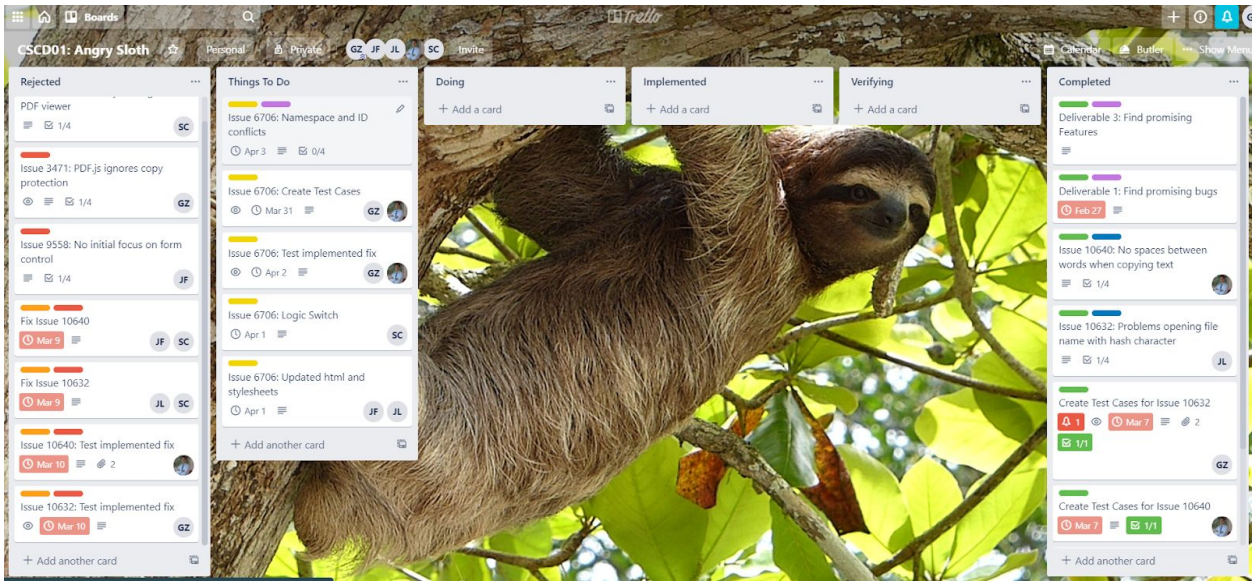
Work in Progress Limits:

Sin Chi Chiu: 10 hours
Jason Fong: 10 hours
Jeremy Lai: 10 hours
Mark Padilla: 10 hours
Gavin Zhang: 10 hours

Development Snapshots

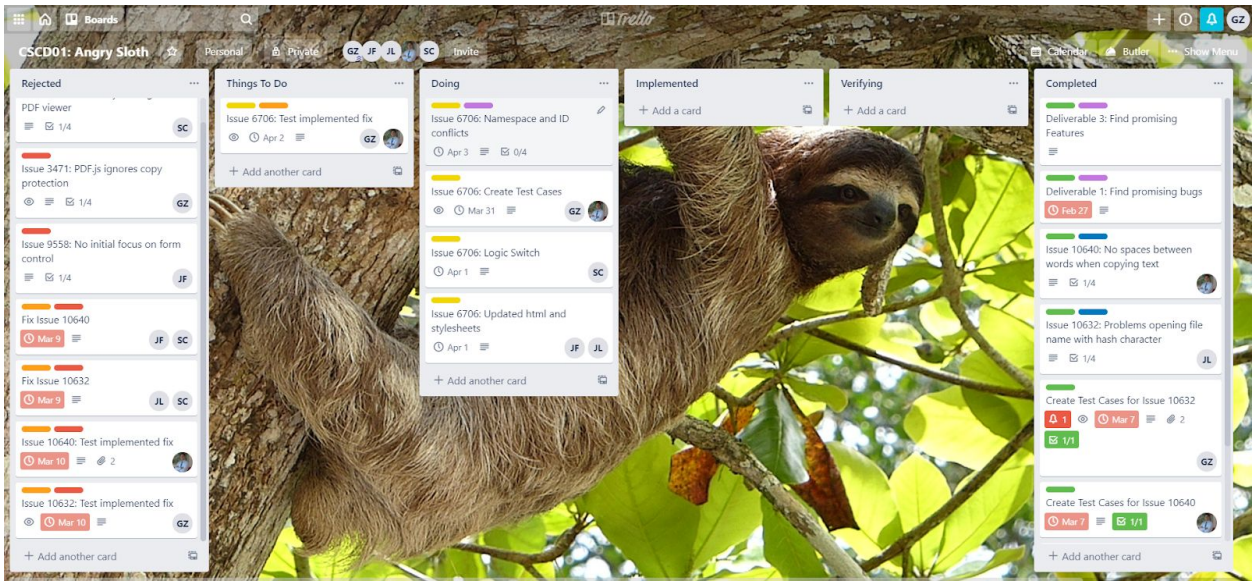
Starting Point

The project starts off exactly where Deliverable 3 left off, we have selected a feature to work on and planned out the tasks required for completing the feature and their estimated costs.



We decided that the feature consisted of three main parts: the logic switch and how it is configured and displayed to the user, the updated style-sheets and HTML documents and the testing for it all.

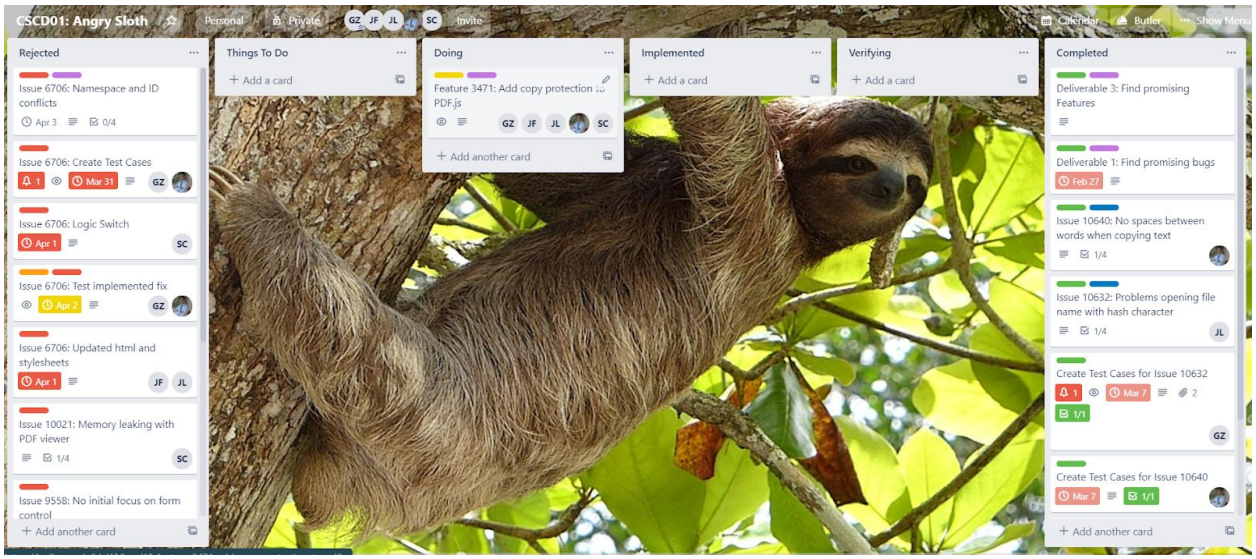
Development Begins



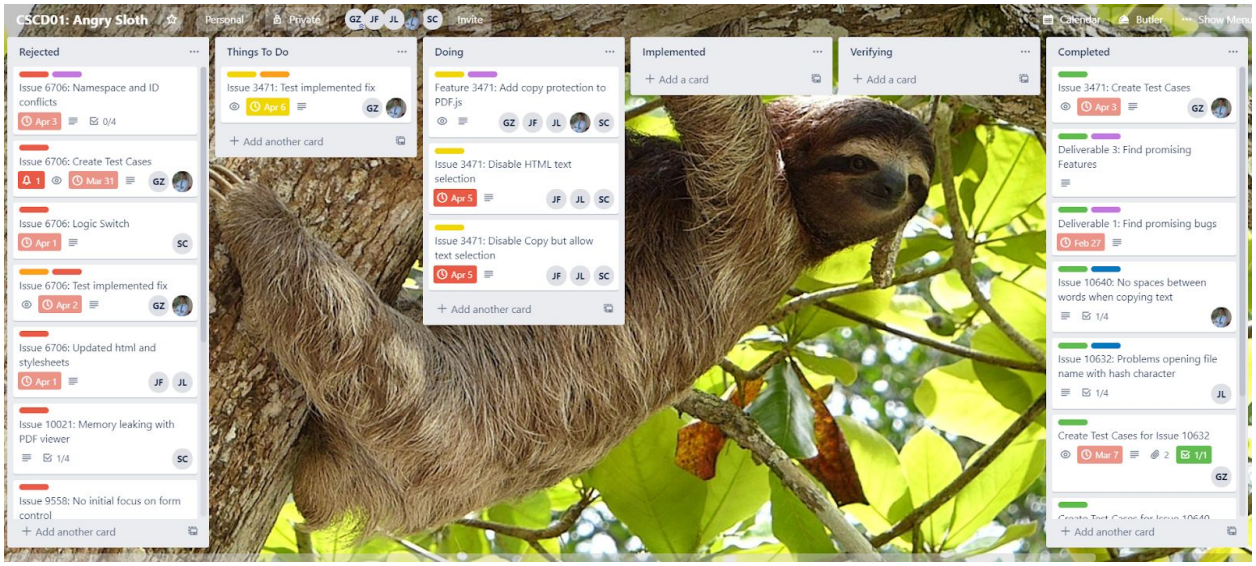
Our estimated costs for the tasks are below our work in progress limits for the duration of this phase, so all possible tasks are pushed into the pipeline with the exception of the actual testing (which is not possible until the actual fix is completed).

Complications

Due to complications explained above, we had to scrap our plans for issue 6706 and switch to issue 3471:

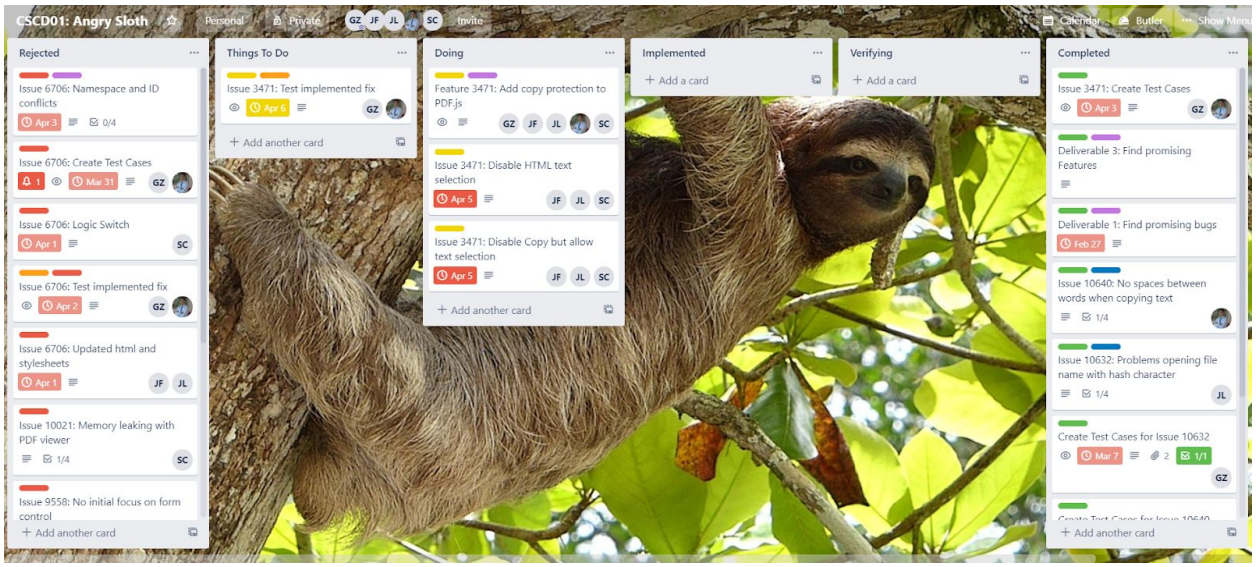


We held an emergency meeting and planned out what we thought would be required for tackling this issue:



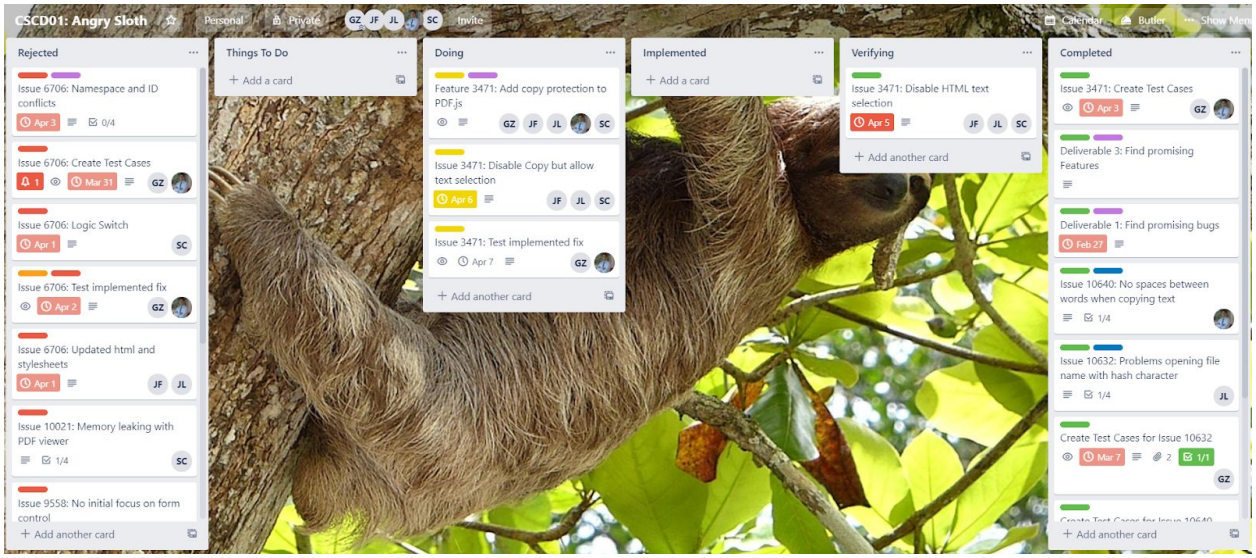
Fortunately for us, we had not been too far into the process for our original plan and still had enough resources to fully switch over to the new issue. So we managed to end up back at the same point we were originally.

Note that there is an error in the above screenshot as the final task that reads “Issue 3471” was not properly renamed yet, it should look like below (although without the passed due dates)



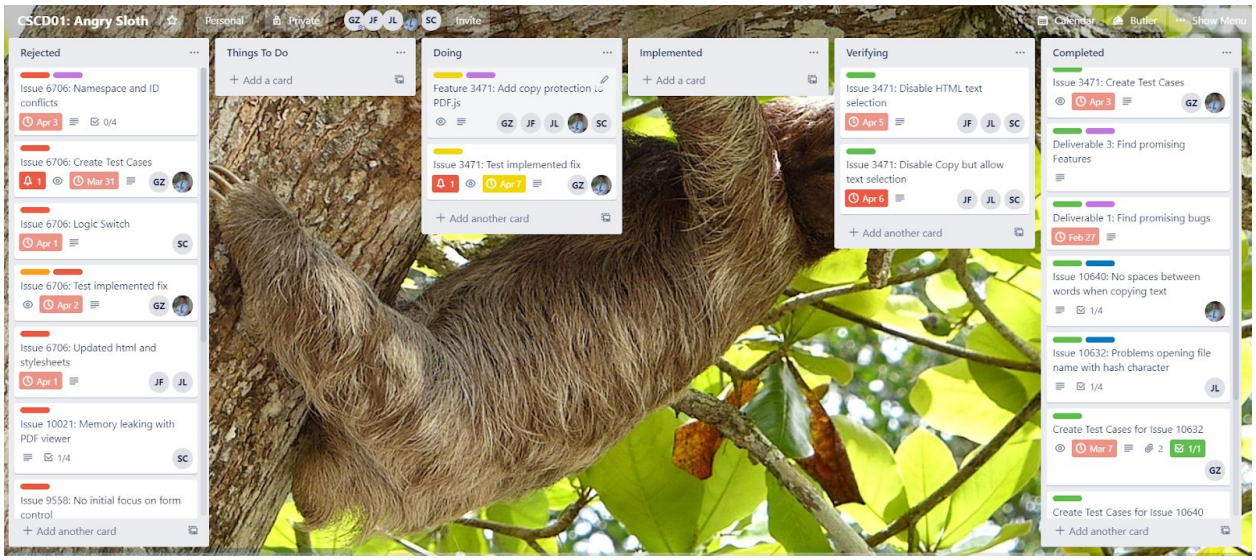
Implementation

As shown above, we came up with two potential fixes for this problem, our preferred fix is the one that still allows text selection while disabling copying. However, the people in charge of PDF.js prefer the much simpler solution of disabling the text layer entirely.

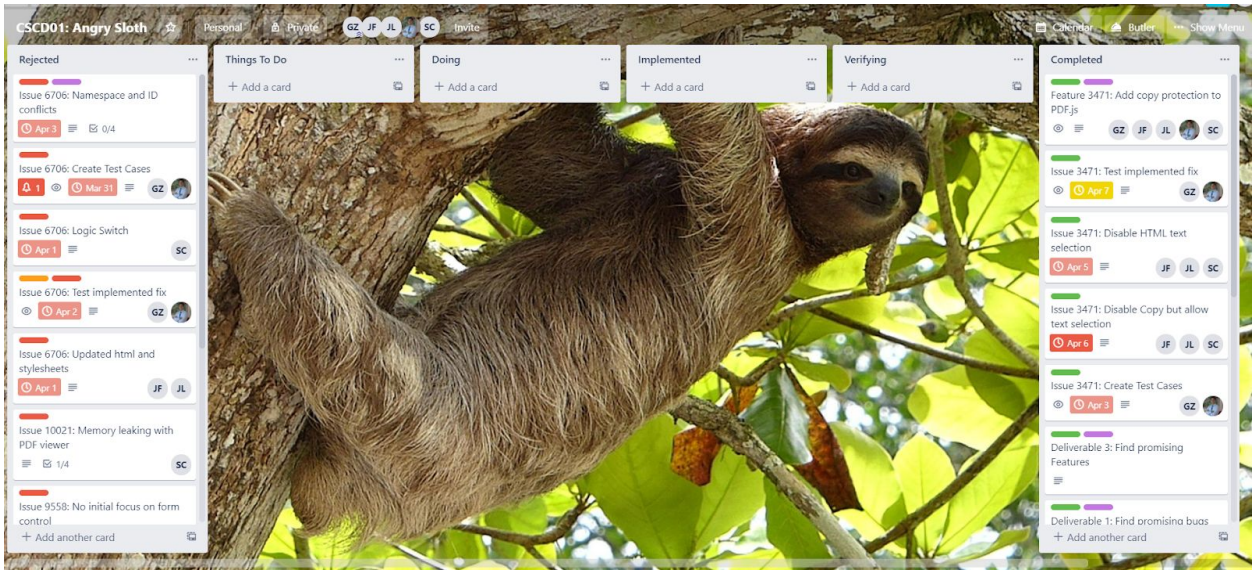


The preferred fix was fixed on time, so we decided to allow for some extra time to be put into our alternate solution.

Simultaneously, the other fix that has already been finished is being verified.



Both fixes managed to be completed on time and the verification process for the alternate fix begins.



Once both fixes have been verified, the code is checked in, the feature is marked completed and the work for this phase is completed.