

# CSCD01 Deliverable 2

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



# Table of Contents

<b>Table of Contents</b>	<b>2</b>
<b>Promising Bugs</b>	<b>4</b>
Problem opening file name with hash	4
Set keyboard focus initially and in some user actions	4
No spaces between words when copying text	5
Memory Leaking with pdf viewer	6
Security: Ignores protection from Copy	6
<b>Selected Bugs</b>	<b>7</b>
Bugs That Were Not Selected	8
<b>Bug Analysis &amp; Fix Attempts</b>	<b>9</b>
No spaces between words when copying text	9
Further Analysis of Issue:	9
Attempted Fix 1 - altering SPACE_FACTOR variable	9
Attempted Fix 2 - Workarounds - adding spaces at the end of spans	12
Attempted Fix 3 - Narrowing down the actual problem	14
Problem opening file name with hash	15
Further Analysis of Issue:	15
Attempted Fix 1 - Incorrect Regex parsing for File Name	15
Attempted Fix 2 - Failure to encode/escape the hash character: encodeURIComponent/encodeURIComponent	16
<b>Acceptance Test Suite</b>	<b>19</b>
Test Cases: No spaces between words when copying text	19
General Set-up:	19
Test Case 1 - affected PDF (original bug)	19
Test Case 2 - non-affected PDF	20
Test Case 3 - affected PDF (noticed during fix attempts)	20
Test Case 4 - affected PDF (noticed during fix attempts)	20
Test Cases: Problem opening file name with hash	21
General Set-up:	21
Test Case 1: Control Testing	21
Test Case 2: Various special cases	21
Hashtag	21
Percentage	21

At (@)	21
Non-reserved Characters	21
Mixed Characters	21
<b>Development Process</b>	<b>22</b>
Board Labels Legend	22
Development Snapshots	23
Selection of Five Promising Bugs	23
Selection of Two Main Bugs	23
Creation and Assignment of Bug Fix Tasks	24
Completion of Test Plans	24
Verifying Unfixable Bugs	25
Completion of Unfixable Bug Verification	28

# Promising Bugs

## Problem opening file name with hash

**Issue:** 10632

In PDF.js you can route to PDFs using direct links such as <http://localhost:8888/web/viewer.html?file=example.pdf> (this would open a local file example.pdf). However when routing like this the file will fail to open if the file name contains the '#' symbol.

We believe this to be caused by the fact that '#' is a reserved character that is used in requests. In this context the # symbol defines a subsection of the page for the browser to automatically scroll to when loading up the page. Additionally, PDF.js similarly uses the # character in order to navigate to subsections of PDF files that have defined subsections. For example if 'example.pdf' had a chapter called "CH1" the URL would look something like <http://localhost:8888/web/viewer.html?file=example.pdf#CH1>.

The creator of the bug has noted a simple workaround of just removing the # symbol from file names. Thus, the implementation will not take a significant amount of time, but we will still need time to pinpoint the problem location. Based on that, we believe the fix to take us less than 10 hours to complete.

## Set keyboard focus initially and in some user actions

**Issue:** 9558

For screen reader users reading PDFs that contain links that jump to other sections within the PDF, these links cannot be followed using only the keyboard.

An example to illustrate this would be in a PDF of a textbook opened in pdf.js, where clicking a link in the Table of Contents within the PDF would allow you to jump to Chapter 11, page 230. A sighted user can click that internal link and jump to page 230, but a screen reader user who may not use a mouse cannot follow this link.

Currently internal links within HTML pages rendered by pdf.js use the JavaScript click event handler to trigger JS code that scrolls the internal link into view. To be able to trigger the same click event using a keyboard only, the user would need to bring a link into focus, by pressing the Tab key, and then pressing the Enter key on the focused link. However in pdf.js pressing the

Enter key seems to scroll to the next page instead on some documents, it doesn't do anything on other documents. Additionally to be able to set focus on a link, all links must have the attribute `tabindex="0"` added to them.

The creator of the issue also brings up that the `tabindex="0"` should be used on elements where the ARIA attribute `role="main"` is set, to allow screen reader users to immediately jump to the main content, so screen reader users would not need to press Tab many times through the pdf.js view controls to reach the content. However when testing different documents, we were unable to find documents where the `role="main"` attribute was even set.

This bug reveals several accessibility issues within pdf.js, and how Web accessibility is sometimes neglected, however given the limited time in this phase, it is infeasible for us to implement all of these accessibility fixes in such a way that we are sure that these fixes are done correctly and fully meet accessibility requirements, which we have limited experience with.

Useful resources for learning more about ARIA (Accessible Rich Internet Applications):

- <https://www.w3.org/TR/wai-aria/#usage>
- <https://www.w3.org/TR/role-attribute/>
- [https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/Roles/Main\\_role](https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/Roles/Main_role)
- [https://developer.mozilla.org/en-US/docs/Web/HTML/Global\\_attributes/tabindex](https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes/tabindex)
- <https://webaim.org/techniques/skipnav/>
- <https://webaim.org/techniques/keyboard/tabindex>
- <https://webaim.org/techniques/javascript/eventhandlers>

## No spaces between words when copying text

**Issue:** 10640

**Related:** 6657

For some PDF files, when the user opens the file in pdf.js, highlights and copies some text, then pastes it into another program, the pasted text will have no spaces. This causes a significant usability issue, as copying and pasting text is a very common operation amongst all computer users.

When we reproduced this bug, we found that each word in the PDF was being rendered as its own span in the pdf.js' HTML output. For example, "He likes cheese" would be rendered as "`<span>He</span><span>likes</span><span>cheese</span>`". Additionally, the related issue thread on Github states that this may be caused by an optimization setting for rendering whitespace. The main issue's thread also points out that this seems to happen only for justified text.

As such, we believe this will be a promising bug because it significantly impacts users and because the root cause is both highly apparent and seemingly straightforward to fix.

Since we were able to reproduce the bug and find good documentation about it, we expect time for troubleshooting and testing will be short. Most of the work will come from investigating which parts of the rendering process are causing this problem. Based on that, we estimate this bug fix to take no longer than 12 hours.

## Memory Leaking with pdf viewer

**Issue:** 10021

When using the pdf viewer, if an instance of the viewer loaded a document and the instance of the viewer is cleaned up and removed from DOM, there are memory leaks from the viewer. This issue is reported by multiple people who embed pdf.js (with viewer) into their web app.

There had been multiple reports showing the culprit is likely being the cleanup function of the display layer. They are showing that some objects in the backend are keeping the data alive in the frontend through the cleanup function, thus causing a memory leak.

We believe this bug to be feasible because we have a good understanding of the causes of memory leaks. However, this will require more investigation compared to other bugs in terms of pinpointing the problem area and analyzing the affected components, so we estimate the bug fix to take at least 15 hours.

## Security: Ignores protection from Copy

**Issue:** 3471

**Related:** Bug 792816 on Bugzilla

When creating PDF documents there are certain security options. One such option is copy protection, this prevents text from the PDF from being selected and copied when viewed in standard PDF viewers like Acrobat or Google Chrome's built in viewer. PDF.js however completely ignores this protection and when copy protected PDF files are opened it allows users to select and copy text.

This bug was tested by creating a generic PDF document and adding copy protection to it (via an online tool). When the copy protected document was opened in Google Chrome's default viewer, we could not highlight and copy the text inside the PDF document. However when the same file was opened via PDF.js we could freely select and copy text from the document.

We believe this issue is caused due to how PDF.js processes documents. In traditional viewers the PDF document is rendered as is, but PDF.js converts the PDF document into an HTML document. In this document all text elements are converted into `<span>` tags which can be freely copied from. Since circumventing this shortcoming will take a great deal of investigation and testing, we estimate this fix to take at least 15 hours.

# Selected Bugs

For this deliverable, we have decided to implement fixes for the following two bugs:

- 10632 - Problem opening file name with hash
- 10640 - No spaces between words when copying text

Both bugs are very straightforward to test manually. This makes it easy for us to create acceptance tests as well as verify the bug fixes using said tests. As pdf.js is an open-source project used by nearly all Firefox users, it is imperative that we are able to test fixes easily and thoroughly so that users are not negatively affected if our fix is merged in.

In the case of 10640, we felt it was a good bug for us to tackle because of the troubleshooting resources available to us. One of these resources is the detailed issue discussion on Github. Here, contributors mentioned specific constant values in pdf.js files that are likely causing the bug. As well, one user described a workaround they made that seemingly circumvented the issue, although they had not been able to verify it thoroughly.

Both of these details gave us a great starting point for triaging. After surveying the code, we were able to narrow down the potential problem area to two files: “text\_layer\_builder.js” and “evaluator.js” in the core layer.

Thus, since we have a good general idea of which code to look at, we feel that 10640 is a bug that we can fix in 12 hours of work. The majority of the time will be spent on investigation, experimenting with the constants to understand their effects (since they are undocumented), and analysis of how the different components work together in producing the bug.

For 10632, we have a good idea as to what the root cause could be, thanks to our knowledge of web browsers and URI strings. We know that the # character is treated specially by browsers to serve as anchor points linking to certain sections of a page. As well, we know that pdf.js uses the # in the browser URL to mark chapter jumps. As such, we are confident that the fix will simply involve finding the location in code where the URL is broadcast to the browser, and then correctly escaping or reformatting the # before the browser interprets it. The main difficulty will come from finding this exact problem location, but once it's found, we expect the reformatting code to be trivial to implement.

Based on this, we expect the bug to take 8 hours to fix: the first 75% of time will be spent on pinpointing the problem area and determining the flow of data to get there, while the latter 25% will be spent on implementing the fix and verifying it.

## Bugs That Were Not Selected

We believe “Security: Ignores protection from Copy” (3471) to be a significant bug beyond our scope because upon inspecting the PDF.js code, the code seemingly does absolutely nothing to respect PDF document permissions. So this expands beyond being a simple bug to a full on new feature. However implementing a whole new feature will take significantly longer than a bug fix and may not be feasible in this phase, especially when compared to the other promising bugs we found.

For “Memory leaking with the pdf viewer” (10021), with the available information provided, we seem to have a good starting point to debug the issue. However upon reproducing the issue, we noticed that we need to implement a test web app that embeds the pdf.js frontend and backend, with no simpler method to reproduce the issue. Given the amount of time in this phase, it becomes infeasible to create a test web app to debug the issue.

“Set keyboard focus initially and in some user actions” (9558) reveals several accessibility issues within pdf.js, and how Web accessibility is sometimes neglected. However given the limited time in this phase, it is infeasible for us to implement all of these accessibility fixes in such a way that we are sure that these fixes are done correctly and fully meet accessibility requirements, which we have limited experience with.



# Bug Analysis & Fix Attempts

## No spaces between words when copying text

**Issue:** 10640

**Related:** 6657

### Further Analysis of Issue:

On most PDF documents, for text selection to work properly:

The **expected behaviour** of the PDF is that the content of **each line of text is rendered within a single `<span>`**.

For example: `<span>This is part of a sentence that fits on a line</span>`

However, we found text rendered **differently** in the following ways:

Case1) On one PDF, all text was rendered within a single span, but the content within the span had no spaces:

`<span>Thisispartofasentencethatfirstoneline</span>`

Example pdf: <https://jfong701.github.io/test/Issue10640BugCase1.pdf>

Case2) On one PDF, text was rendered with **each word in its own span**:

`<span>This</span><span>is</span><span>part</span>...`

Example pdf: <https://jfong701.github.io/test/Issue10640Buggy.pdf>

Case3) On another PDF, **individual words**, or segments of words were split into **multiple spans**.

`<span>T</span><span>h</span><span>is</span>...`

Example pdf: <https://jfong701.github.io/test/Issue10640BugCase3.pdf>

On further investigation of the issue we traced the code when rendering several different test pdfs, and explored suggestions and previous attempts from fellow developers on Github.

### Attempted Fix 1 - altering SPACE\_FACTOR variable

```
1700  var SPACE_FACTOR = 0.3;           Hengjie, 5 years ago • Lower threshold
1701  var MULTI_SPACE_FACTOR = 1.5;
1702  var MULTI_SPACE_FACTOR_MAX = 4;
```

By setting the SPACE\_FACTOR variable from 0.3 to 0.1, within the construction of textContentItem within /core/evaluator.js (around line 1696), it allows for the textContentItem.fakeSpaceMin variable to be set to a lower value, making it more likely for text

within a span to have its spaces properly applied in the addFakeSpaces() function, as the text would likely meet the conditions of being less than the computed width of the text characters (which are sent in as a parameter to this function)

*Picture:* PDF operations: showSpacedText. There is also OPS.showText, which doesn't call addFakeSpaces()

```
2094 case OPS.showSpacedText:
2095   if (!stateManager.state.font) {
2096     self.ensureStateFont(stateManager.state);
2097     continue;
2098   }
2099   var items = args[0];
2100   var offset;
2101   for (var j = 0, jj = items.length; j < jj; j++) {
2102     if (typeof items[j] === "string") {
2103       buildTextContentItem(items[j]);
2104     } else if (isNum(items[j])) {
2105       ensureTextContentItem();
2106     }
2107     // PDF Specification 5.3.2 states:
2108     // The number is expressed in thousandths of a unit of text
2109     // space.
2110     // This amount is subtracted from the current horizontal or
2111     // vertical coordinate, depending on the writing mode.
2112     // In the default coordinate system, a positive adjustment
2113     // has the effect of moving the next glyph painted either to
2114     // the left or down by the given amount.
2115     advance = (items[j] * textState.fontSize) / 1000;
```

(width calculations take place between lines 2115-2139), set to variable “advance”

```
2139   if (breakTextRun) {
2140     flushTextContentItem();
2141   } else if (advance > 0) {
2142     addFakeSpaces(advance, textContentItem.str);
2143   }
```

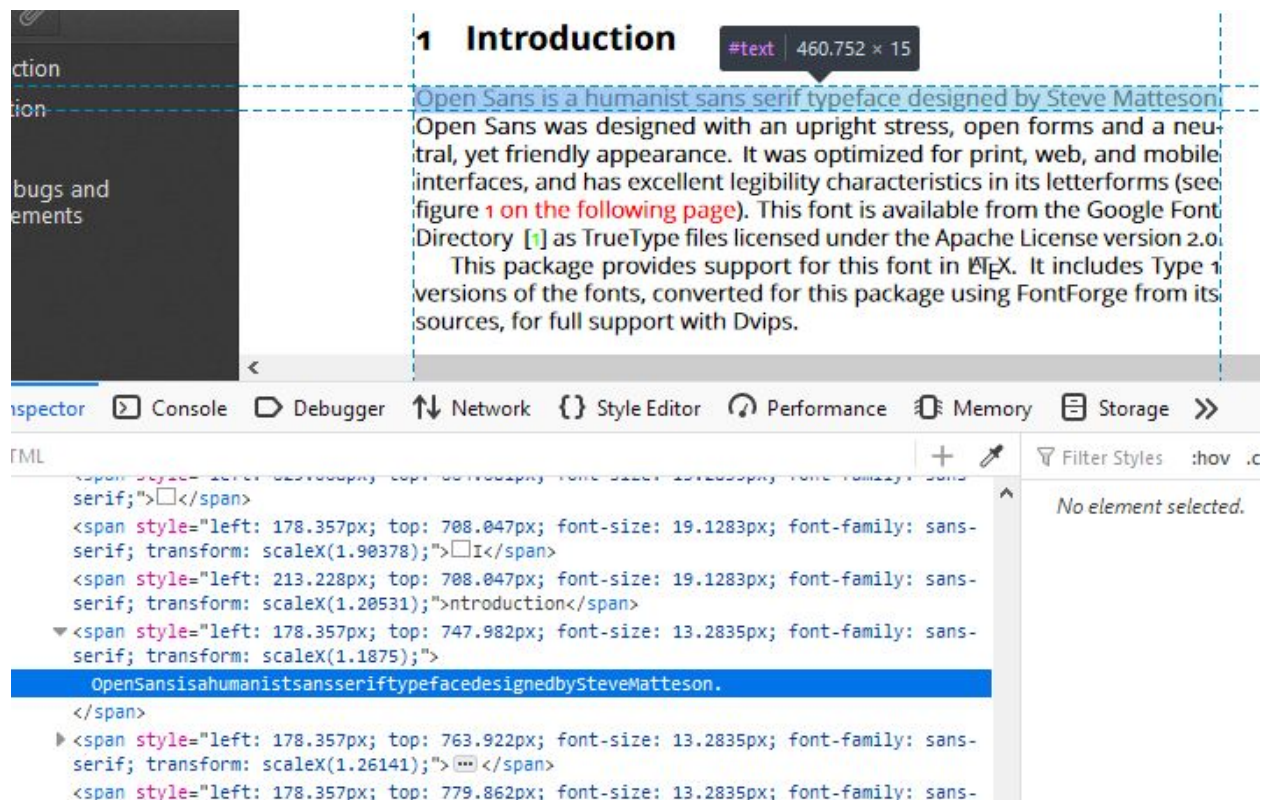
*Picture:* addFakeSpaces() function

```
1898 function addFakeSpaces(width, strBuf) {
1899   if (width < textContentItem.fakeSpaceMin) {
1900     console.log('no space added', width, textContentItem.fakeSpaceMin);
1901     return;
1902   }
1903   if (width < textContentItem.fakeMultiSpaceMin) {
1904     strBuf.push(" ");
1905     console.log('single space added', width, textContentItem.fakeMultiSpaceMin);
1906     return;
1907   }
1908   var fakeSpaces = Math.round(width / textContentItem.spaceWidth);
1909   console.log('n spaces added ', fakeSpaces);
1910   while (fakeSpaces-- > 0) {
1911     strBuf.push(" ");
1912   }
1913 }
```

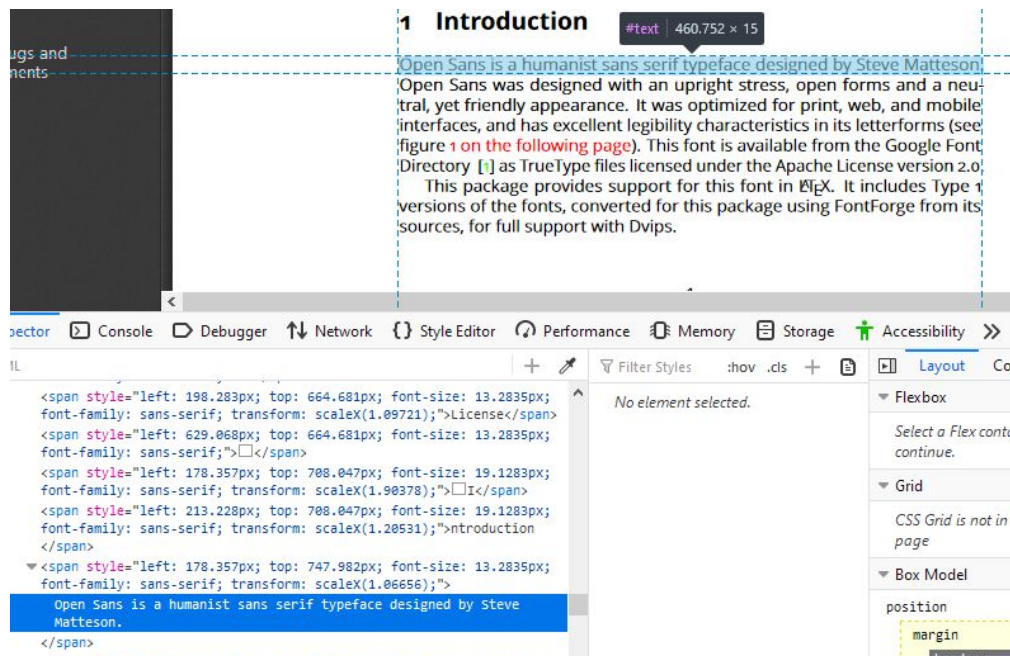
If the PDF operation set in the Evaluator preProcessor was set to OPS.ShowSpacedText, then as part of this, the addFakeSpaces() function would be called, this is where the value of the constants set above affect calculations for adding spaces within a span.

Playing with these parameters was suggested on this Github issue, and related issues by one of the lead developers for pdf.js. We found that this fix is effective, but only on PDFs which encountered the first type of issue, where all text of a line is in one span, but the spaces are missing. Tweaking these parameters had no effect on PDFs where a line was split into multiple spans as in cases 2) and 3).

Picture: Before fix:



Picture: Attempted Fix 1) applied to problem case 1). It correctly applies spaces within the span, allowing for proper text selection. It however has no effect (positive or negative) on cases 2) and 3).



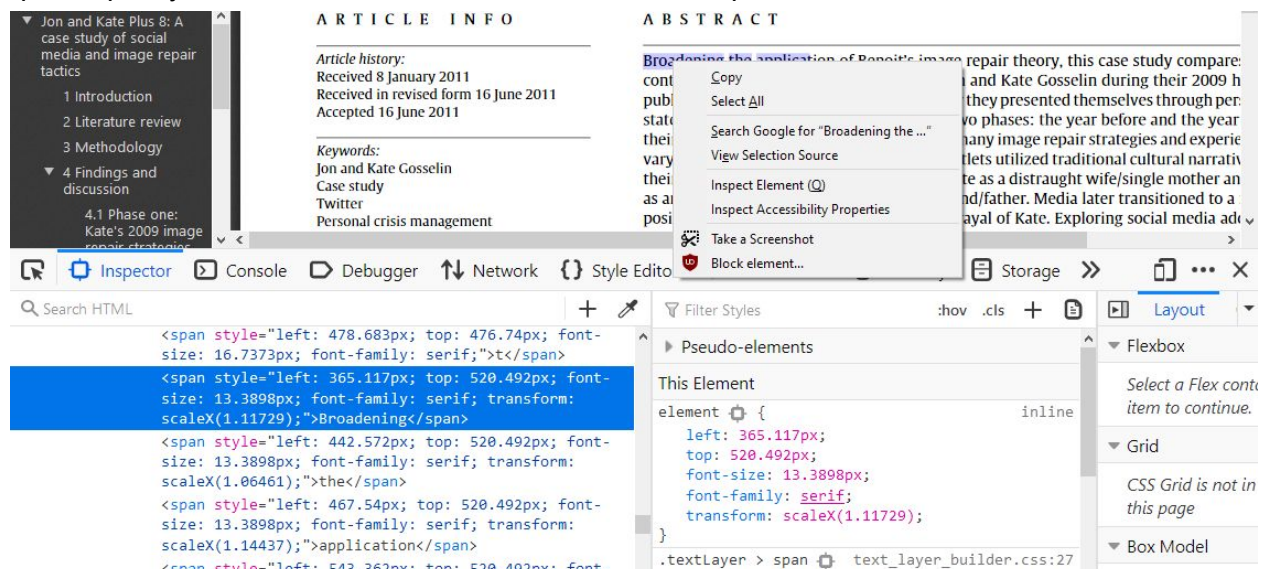
## Attempted Fix 2 - Workarounds - adding spaces at the end of spans

Other users on Github discovered a workaround (<https://github.com/mozilla/pdf.js/issues/7310>) to problem 2), where each word is placed inside of an individual span instead of on the line, and the workaround was adding extra whitespace to the end of each individual span through either the core evaluator, or within the /display/text\_layer. This resolves problem case 2), without negatively affecting working PDFs, or PDFs which were fixed through Attempted Fix 1.

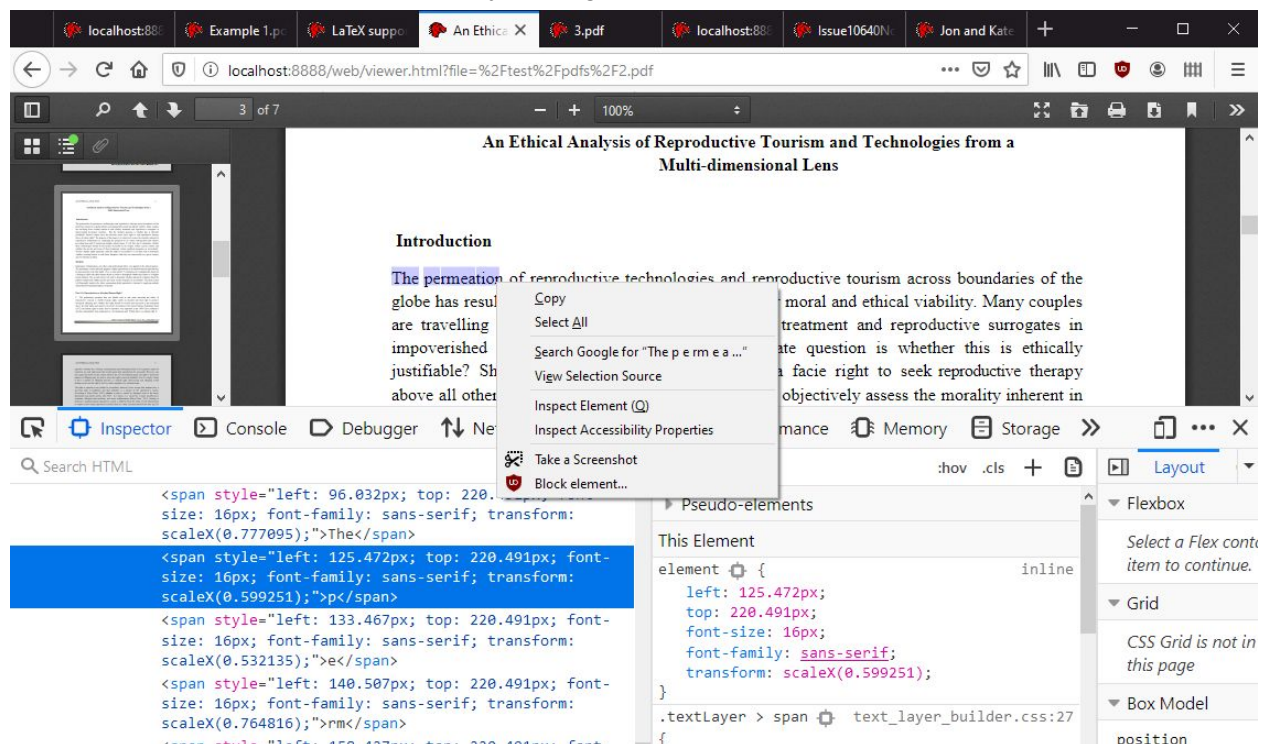
However using this fix would negatively affect PDFs which had words that were partitioned into multiple <span> elements in the style of case 3), and would introduce additional spaces in this case when trying to select and copy text from the PDF.



Picture: Workaround of adding spaces to the end of spans does fix text selection for PDFs with spans split by case 2, where each word was in its own span.



Picture: However, for PDFs in case 3, where individual words were split into spans, this workaround worsens text selection, by adding in non-existent spaces.



### Attempted Fix 3 - Narrowing down the actual problem

When seeing cases 1), 2), and 3) complicating fixes, it seems that the actual problem causing the text selection problem is really how pdf.js applies `<span>` elements to lines. This leads into why fixes have been difficult for open source developers to apply.

The extraction of text content from the stream of bytes and calculations related to font and font width occur within evaluator, as well as font sizing heuristics, based on the PDF specification. They are heuristics because pdf.js cannot possibly be aware of every font, and fonts are not always embedded properly in PDFs. pdf.js has to handle many different fonts, languages, symbols, and even postscript fonts, which are represented as glyphs. Pdf.js also cannot guarantee that the PDF it is trying to read was created fully following the PDF specification. As a result, the evaluator has a large amount of calculations, and sometimes these font sizing calculations to determine proper spacing of strings can be inaccurate. It is difficult to narrow down exactly which part of the evaluator, or if it is only the evaluator responsible for this, especially when different test PDFs encounter different issues, and may also use several different fonts within them. There is also the added complication during debugging of the evaluator receiving data in pieces from the stream declared in `/core/document`.

# Problem opening file name with hash

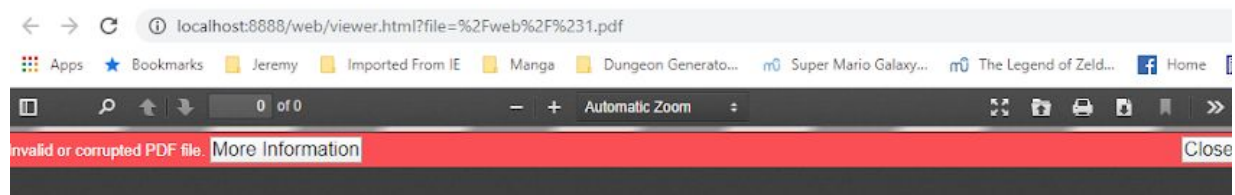
**Issue:** 10632

## Further Analysis of Issue:

While not a common thing to do, PDF names are able to have **special characters** and **reserved characters**.

For example, generating a PDF with the name `#.pdf` or it containing the hash character is allowed. When directly opening a file with the viewer via a direct file open button, such files were accessible.

However, when opening it via a **url redirect**, it would generally display the following error:



The file would not be able to open as it stated plainly, it was an invalid or corrupted file. After further analysis, we came to several hypotheses to why it was failing.

## Attempted Fix 1 - Incorrect Regex parsing for File Name

We looked at the part of the code that handled accepting file names, thinking it might have missed the hash character. For this, we found the regex expression and checked online whether it was correct.

```
(function rewriteUrlClosure() {  
  // Run this code outside DOMContentLoaded to make sure that the URL  
  // is rewritten as soon as possible.  
  const queryString = document.location.search.slice(1);  
  const m = /^(^&)file=(^&)*\/.exec(queryString);  
  defaultUrl = m ? decodeURIComponent(m[2]) : "";
```

Picture: Area of the code, in **viewer.js** we initially believed to be the cause of the faulty file retrieval

Upon confirming the regex, which was basically a generic expression to handle all filename types, we found out it wasn't a regex issue, and began to look at other avenues of error. From there we decided to check if it was even attempting to get the correct file, and proceeded with our 2nd attempt to fix it.

## Attempted Fix 2 - Failure to encode/escape the hash character: encodeURIComponent/ encodeURIComponent

Afterwards we decided to check what the url looked like when the request was sent, between a valid openable file and the problematic file. We then noticed that the proper file had **some** characters like the blank character converted to the **URI encoding**, but others didn't.

File Name

[test#hashtag.pdf](#)



▼ General  
Request URL: http://localhost:8888/test  
Request Method: GET  
Status Code: 🟡 301 Moved Permanently  
Remote Address: 127.0.0.1:8888  
Referrer Policy: no-referrer-when-downgrade

Missing #hashtag

[test%percent.pdf](#)



▼ General  
Request URL: http://localhost:8888/test%percent.pdf  
Request Method: GET  
Status Code: 🔴 400 Bad Request  
Remote Address: 127.0.0.1:8888  
Referrer Policy: no-referrer-when-downgrade

[test@at.pdf](#)



▼ General  
Request URL: http://localhost:8888/test@at.pdf  
Request Method: GET  
Status Code: 🟢 200 OK  
Remote Address: 127.0.0.1:8888  
Referrer Policy: no-referrer-when-downgrade

[test space.pdf](#)



▼ General  
Request URL: http://localhost:8888/test%20space.pdf  
Request Method: GET  
Status Code: 🟢 200 OK  
Remote Address: 127.0.0.1:8888  
Referrer Policy: no-referrer-when-downgrade



The above screenshot showed that when the URL contains the hash character, it is treated as the fragment identifier and therefore anything starting from # is not sent to the server. In addition, we found that the % symbol also caused the file to fail to open with a different error. This led us to believe the URL is not being encoded properly. According to the [documentation](#), PDF.js does not perform URI encoding when sending URL, and users must implement `encodeURIComponent()` when specifying the file in URL.

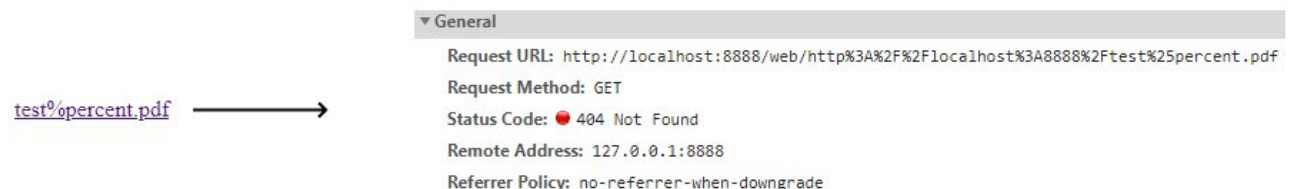
#### ▼ Request call stack

```
PDFFetchStreamReader      @ fetch_stream.js:149
getFullReader             @ fetch_stream.js:82
eval                      @ api.js:1900
eval                      @ message_handler.js:324
_createStreamSink         @ message_handler.js:323
MessageHandler._onComObjOnMessage @ message_handler.js:135
```

#### ▼ Request initiator chain

```
▼ http://localhost:8888/src/display/fetch_stream.js
```

Looking at the call stack we know that the error is raised from `fetch_stream.js`. After some inspection, we found the culprit and added `encodeURIComponent()` to the fetch function in **PDFStreamReader** in `fetch_stream.js`, the GET request becomes the following:



The server took the URL as literal string without decoding, causing a 404 status due to the file not being found. However using `encodeURI()`, filename with % will work without issues, but since `encodeURI()` **does not encode #**, it does not solve the issue with #.

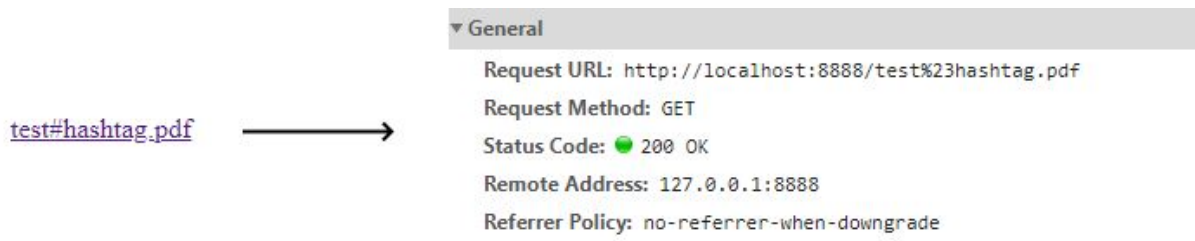
The above debugging steps showed that the problem also comes from the backend. Therefore we implemented a simple string replace with regular expression in **fetch\_stream.js** to verify the issue:

```
this._headers = createHeaders(this._stream.httpHeaders);
const url = encodeURI(source.url).replace(/#/g, '%23');
fetch(
  url,
  createFetchOptions(
    this._headers,
    this._withCredentials,
    this._abortController
  )
)
```

And on the server end, in `./test/webserver.js` we implemented the same string replace with it converting back to #:

```
_handler: function(req, res) {
  var url = req.url.replace(/\\/\\/g, "/");
  url = url.replace(/%23/g, '#');
  var urlParts = /([^?]*)(?:\?(.*)?)?.exec(url);
  try {
    // Guard against directory traversal attacks such as
    // `../../../../../../etc/passwd`, which let you make GET requests
    // for files outside of `this.root`.
    var pathPart = path.normalize(decodeURI(urlParts[1]));
    // path.normalize returns a path on the basis of the current platform.
    // Windows paths cause issues in statFile and serverDirectoryIndex.
    // Converting to unix path would avoid platform checks in said functions.
    pathPart = pathPart.replace(/\\/g, "/");
  } catch (ex) {
    // If the URI cannot be decoded, a `URIError` is thrown. This happens for
    // malformed URIs such as `http://localhost:8888/%s%s` and should be
    // handled as a bad request.
    res.writeHead(400);
    res.end("Bad request", "utf8");
    return;
  }
}
```

With the # encoded in the frontend and decoded in the backend, filename containing # can now be opened without error:



The diagram illustrates the process of handling a URL with a hash character. On the left, the URL `test#hashtag.pdf` is shown. An arrow points to a box titled "General" which contains the following details:

- Request URL: `http://localhost:8888/test%23hashtag.pdf`
- Request Method: GET
- Status Code: 200 OK
- Remote Address: 127.0.0.1:8888
- Referrer Policy: no-referrer-when-downgrade

This workaround on encoding and decoding # in filename exposes a problem; if the file name contains "%23" (the UTF-8 encoding of "#"), it will convert it to "#" in the backend and causes the server to not find the correct file. However Based on our findings in the [FAQ](#) and reason on closing issue [11683](#), this technically should not be considered as a bug. The reasoning is that this is the expected behaviour of pdf.js. If an user wants to specify a different PDF in the default viewer through URL, the user is expected to handle any special characters in the URL from both the frontend and backend.

# Acceptance Test Suite

Both of the selected bugs, “No spaces between words when copying text” and “Problem opening file name with hash” require interaction with the browser that lies outside the scope of the testing framework provided by pdf.js.

For “No spaces between words when copying text”, the user needs to highlight and copy words from a document in pdf.js’ web viewer. To accomplish this, we would need to add a GUI testing framework like Selenium to the project. But this would require us to get permission from the project maintainers. Additionally, the time needed to set up and learn Selenium or a similar library would likely take almost as much, if not more time, than implementing the fix itself.

We have a similar reasoning for “Problem opening file name with hash”. This bug involves the browser’s URL bar, which again lies outside the scope of the pdf.js testing suite. We could use Selenium or a similar framework to simulate this, but again, this requires a significant amount of time to obtain permission and set everything up.

For these reasons, we have decided to use manual test cases for both of our fixes. This is ideal because both bugs are easy to recreate and verify for a human user.

## Test Cases: No spaces between words when copying text

**Goal:** Ensure that when text is highlighted in the pdf.js viewer, then copied and pasted into a different application, that the pasted text preserves the spaces from the PDF.

*Note:* The original test cases for this issue were Test cases 1 and 2. Test cases 3 and 4 were added upon further investigation of causes of issue 10640 affecting text selection.

### General Set-up:

1. From our PDF repository <https://github.com/jfong701/test> take all PDF files with the “Issue10640” prefix and save it locally

### Test Case 1 - affected PDF (original bug)

→ *Case 2 in section: Bug Analysis & Fix Attempts for issue 10640*  
(text selection has no spaces, each word was in its own <span>)

1. In a non-pdf.js PDF viewer (e.g. Chrome, Acrobat), open “Issue10640Buggy.pdf”.
2. In pdf.js’ web viewer, use the ‘Open File’ button to load “Issue10640Buggy.pdf”.

3. Verify that the PDF is rendered identically in both viewers.
4. In pdf.js, highlight at least one full paragraph from the body text of page 1 and copy it.
5. Open any text editor and paste the clipboard.
6. Verify that the pasted text has spaces in the right places (i.e. spaces should be in the same places as in the PDF).

## Test Case 2 - non-affected PDF

1. In a non-pdf.js PDF viewer (e.g. Chrome, Acrobat), open "Issue10640NotBuggy.pdf".
2. In pdf.js' web viewer, use the 'Open File' button to load "Issue10640NotBuggy.pdf".
3. Verify that the PDF is rendered identically in both viewers.
4. In pdf.js, highlight at least one full paragraph from the body text of page 1 and copy it.
5. Open any text editor and paste the clipboard.
6. Verify that the pasted text has spaces in the right places (i.e. spaces should be in the same places as in the PDF).

## Test Case 3 - affected PDF (noticed during fix attempts)

→ *Case 1 in section: Bug Analysis & Fix Attempts for issue 10640*

(text selection has no spaces, whole line in one <span>)

1. In a non-pdf.js PDF viewer (e.g. Chrome, Acrobat), open "Issue10640BugCase1.pdf".
2. In pdf.js' web viewer, use the 'Open File' button to load "Issue10640BugCase1.pdf".
3. Verify that the PDF is rendered identically in both viewers.
4. In pdf.js, highlight at least one full paragraph from the body text of page 1 and copy it.
5. Open any text editor and paste the clipboard.
6. Verify that the pasted text has spaces in the right places (i.e. spaces should be in the same places as in the PDF).

## Test Case 4 - affected PDF (noticed during fix attempts)

→ *Case 3 in section: Bug Analysis & Fix attempts for issue 10640*

(text selection no spaces, words and sentence fragments were split in the middle with spans)

1. In a non-pdf.js PDF viewer (e.g. Chrome, Acrobat), open "Issue10640BugCase3.pdf".
2. In pdf.js' web viewer, use the 'Open File' button to load "Issue10640BugCase3.pdf".
3. Verify that the PDF is rendered identically in both viewers.
4. In pdf.js, highlight at least one full paragraph from the body text of page 1 and copy it.
5. Open any text editor and paste the clipboard.
6. Verify that the pasted text has spaces in the right places (i.e. spaces should be in the same places as in the PDF).

# Test Cases: Problem opening file name with hash

**Goal:** Ensure that local files with hashes (#) in their filenames can be loaded into pdf.js' viewer.

## General Set-up:

- 1: In the root directory of the pdf.js project navigate to /test/pdfs/
- 2: From our PDF repository <https://github.com/jfong701/test> take all PDF files with the "Issue10632" prefix and copy then into /test/pdfs/
- 3: Locally run the PDF.js project using the "gulp server" command
- 4: Navigate to <http://localhost:8888/test/pdfs/> and attempt to open each of the files starting with the control files

## Test Case 1: Control Testing

The Issue10632 files labeled "Control" do not have any special characters in their names, verify that they open correctly.

## Test Case 2: Various special cases

### Hashtag

The Issue10632 files labeled "Hashtag" contain a hashtag in their name which is a URL encoding reserved character, open them to verify that they open and are the same as the 2 control PDFs.

### Percentage

The Issue10632 files labeled "Percent" test the percentage symbol, which is a URL encoding reserved character, open the files and ensure they are the same as the control PDFs.

### At (@)

The Issue10632 files labeled "At" test the @ symbol, which is a URL encoding reserved character, open the files and ensure the PDF's are the same as control.

### Non-reserved Characters

There are various Issue10632 files labeled with various non URL encoding reserved characters to test as well such as various punctuation marks, underscores, spaces, and tildes.

### Mixed Characters

The Issue10632 files labeled "ALL" contain all special characters above, open them to verify that they open and are the same as the control PDFs.

# Development Process

During our meetings, we decided that our WIP Limit will be based on number of work hours available per member per week. This limit was decided on implicitly between all members because Trello does not provide a way to represent the WIP limit visually.

These are the WIP limits for each member during this deliverable:

- Sin: 4 hours
- Jason: 8 hours
- Jeremy: 6 hours
- Mark: 6 hours
- Gavin: 6 hours

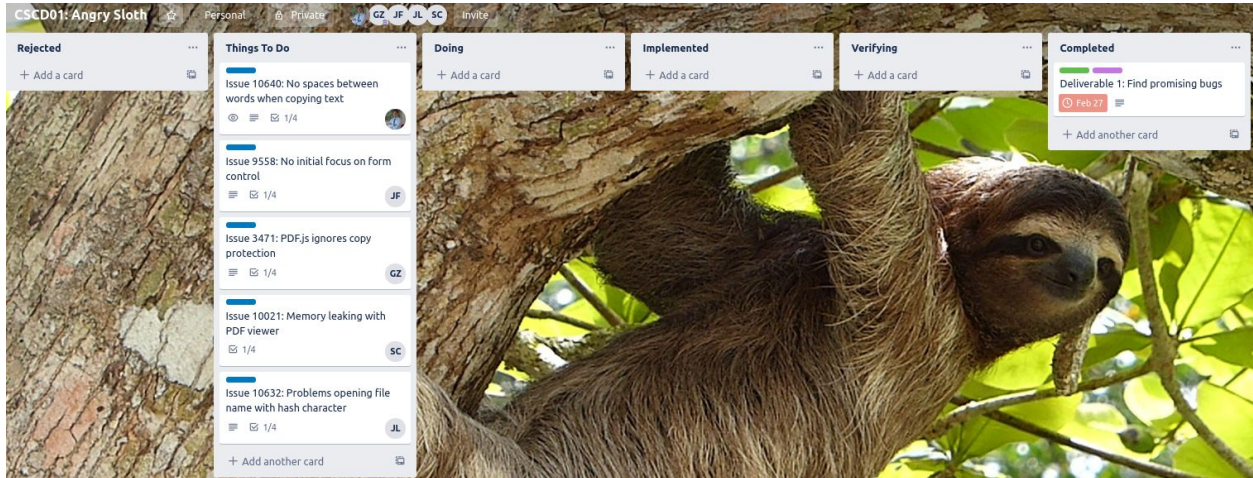
## Board Labels Legend

- **Red** = rejected
  - Feature / Bug rejected for current iteration
- **Blue** = accepted
  - Feature / Bug accepted for current iteration
- **Yellow** = in progress
- **Orange** = blocked
- **Green** = done
- Special case - **Purple** = assigned to all members



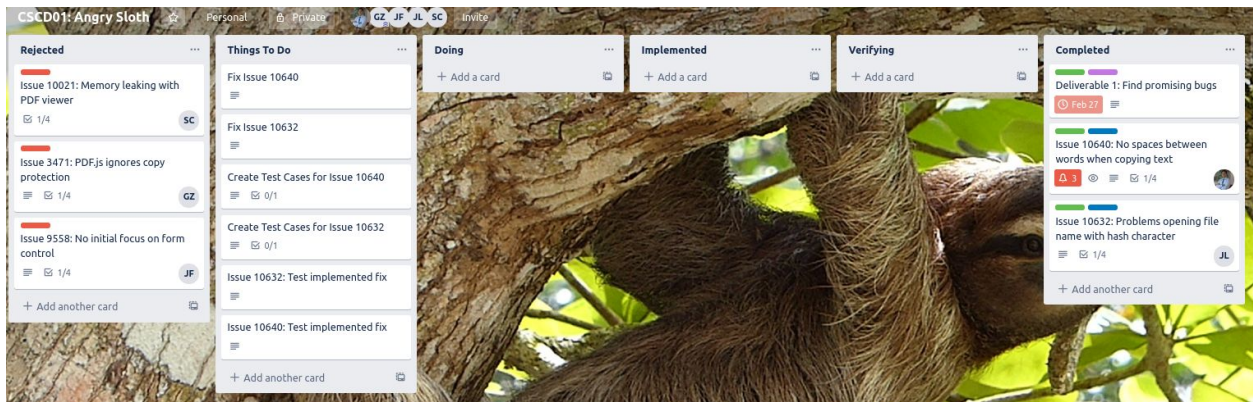
# Development Snapshots

## Selection of Five Promising Bugs



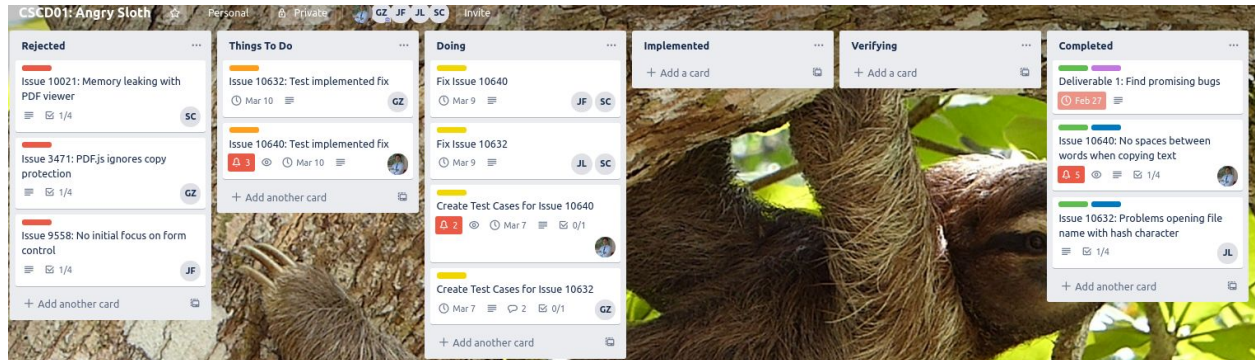
These are the five promising bugs we initially chose for the first part of the deliverable.

## Selection of Two Main Bugs



In this snapshot, we've narrowed down the bugs to the two we plan to implement fixes for. The other three have been rejected.

## Creation and Assignment of Bug Fix Tasks

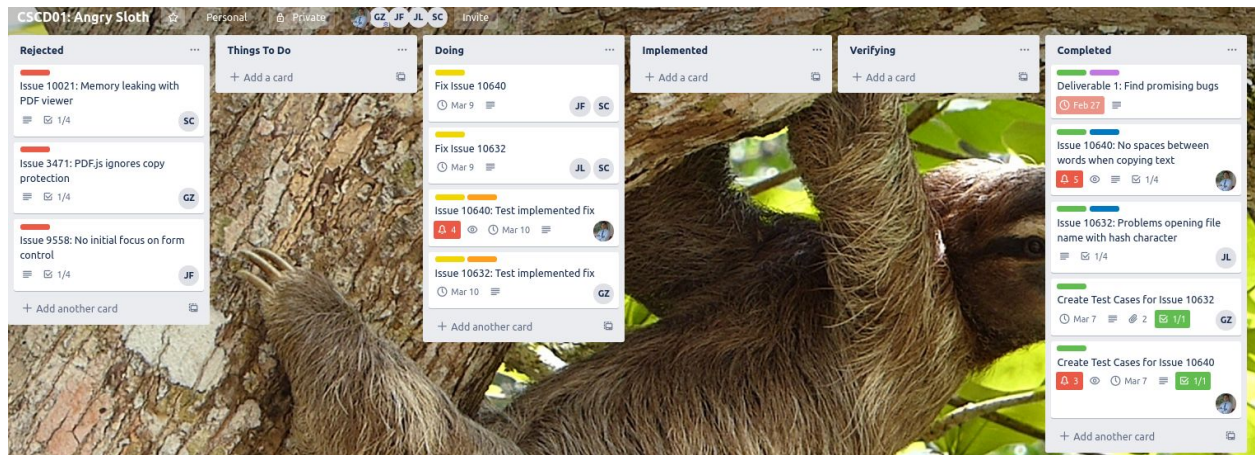


We agreed that two members would focus on testing tickets, while the other three members will implement the bug fixes.

The reason for this split is because we are doing Test-Driven Development. As such, we need to devote more time and resources to testing. This also helps with communicating to the fixers when the test writing is done so that they can begin working on fixes.

Since we're fixing two bugs, one of the three members will help with both tickets.

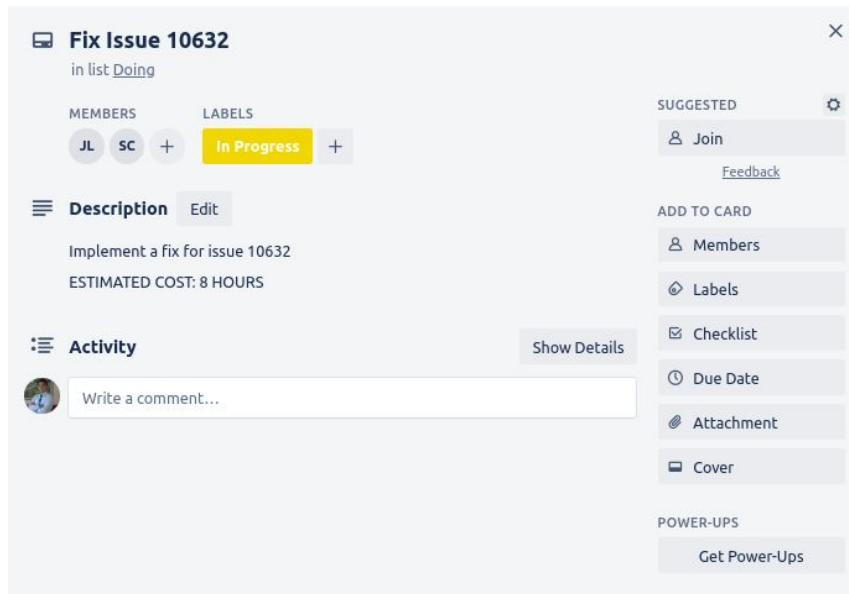
## Completion of Test Plans



In this snapshot, test plans for both bugs have been created and distributed amongst the team. Now the bug fixes are in progress for implementation.

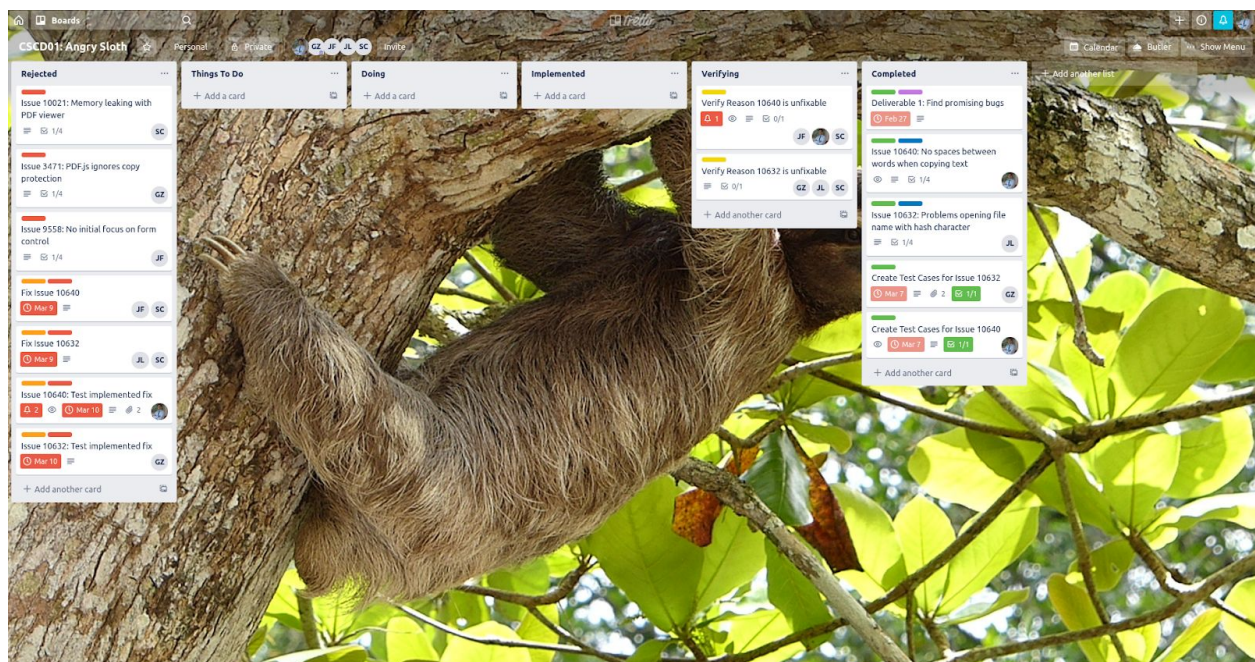
The use of the 'Blocked' label on the two verification tickets helps show everyone that testing is being waited on until the fix implementations are done.





Here is the 'back' of one of the Kanban cards in the 'Doing' column. Note this part of the description: 'ESTIMATED COST: 8 HOURS'. Each ticket has this cost associated with it, and this is what counts toward the WIP Limit.

## Verifying Unfixable Bugs






After working on the bug fixes, we determined that neither bug was fixable (as explained in previous sections). Due to this, we blocked all of the implementation and verification tickets

since they are no longer doable (not the move to the 'Rejected' column and the orange 'blocked' labels). Additionally, we started two other tickets verifying that the bugs are indeed unfixable. These changes to the board clearly communicate that we've encountered unsolvable issues with the bugs and that we have changed gears to instead prove that they are indeed unfixable.

Verify Reason 10640 is unfixable


in list [Completed](#)

MEMBERS

JF  SC  

LABELS

Completed



Description

Edit

Verify given reasons of why 10640 is unfixable:

Root cause seems to be how pdf.js separates text into `<span>` elements. Desired behaviour for proper text selection is to render one line of text in one `<span>` element. Some PDFs don't render this way.

Workaround fixes exist, but these can't be applied universally, because it depends on how the specific PDF splits its text incorrectly.

To fix text selection spaces would mean reworking how pdf.js renders PDF `<span>` elements.

pdf.js uses the PDF specification to help calculations, which may or may not be correctly followed by PDFs. It is unclear how to fix this, or if it can even be fixed.

Checklist

Hide completed items

Delete

100%

Confirm the inconsistent results

Add an item

Activity

Show Details

JF

Write a comment...

ADD TO CARD

Members

Labels

Checklist

Due Date

Attachment

Cover

POWER-UPS

Get Power-Ups

ACTIONS

Move

Copy

Make Template

Watch

Archive

Share

This is the inside of one of the verification tickets. It contains a description of what we found during our prior investigation, which allows us to use this knowledge to guide our verification.

show that we have show that we have show that we have

Verify Reason 10632 is unfixable

in list [Completed](#)

MEMBERS

GZ

JL

SC

+

LABELS

Completed

+

Description

Edit

Confirm the reasons 10632 is unfixable:

UPDATE 03/10/20:  
This occurs because the server is not configured to decode the hashtag element correctly, we have found a fix for it locally because we can modify the webserver.js file but in actual applications this must be configured by the other application and not PDF.js

Server cannot receive # symbol and it is not properly encoded, but ALL other special characters are properly encoded (except %, but this is an expected behaviour)

Checklist

Hide completed items

Delete

100%

Confirm the above reasons

🕒 👤 ⋮

Add an item

Activity

Show Details

JF

Write a comment...

SUGGESTED

⚙️

Join

Feedback

ADD TO CARD

Members

Labels

Checklist

Due Date

Attachment

Cover

POWER-UPS

Get Power-Ups

ACTIONS

→ Move

📄 Copy

📄 Make Template

👁 Watch

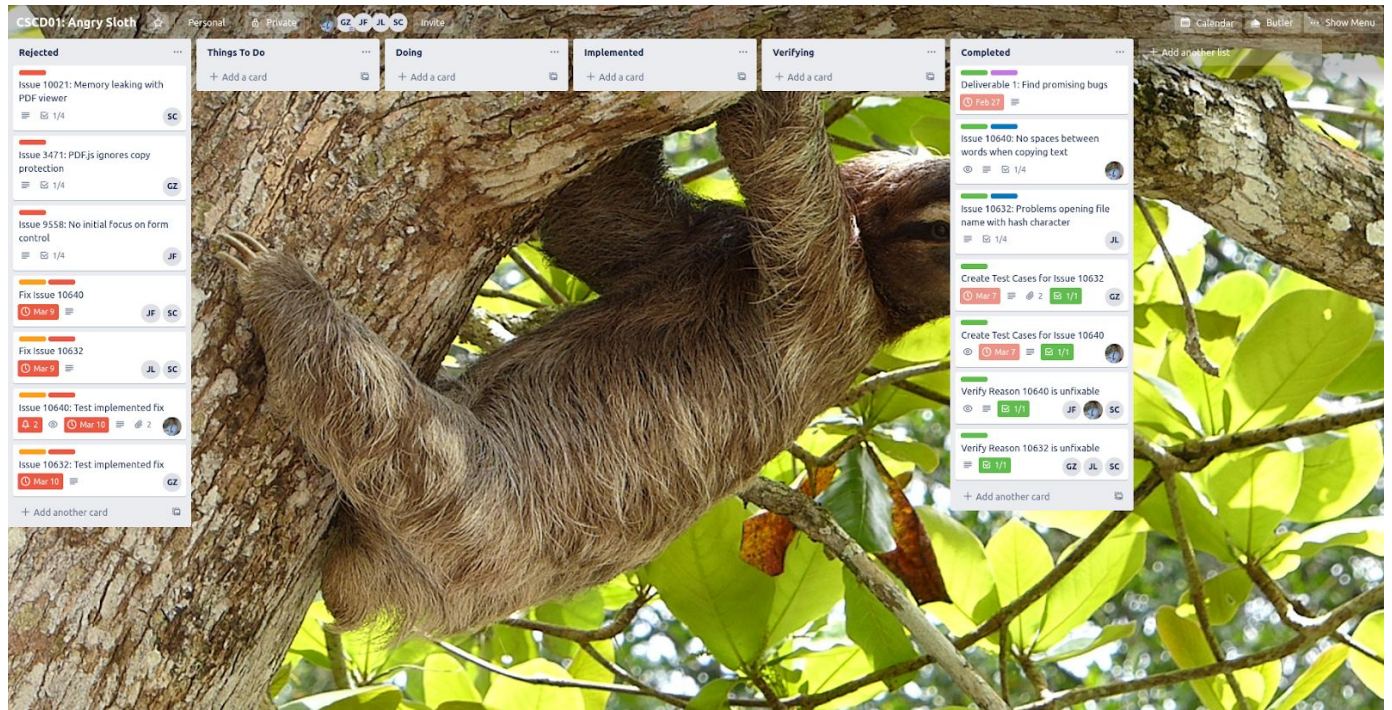
📁 Archive

🔗 Share

This is the inside of the other verification ticket. Notice the update to it due to new information that was found.



## Completion of Unfixable Bug Verification



This is the final snapshot from the end of our deliverable work.

This board state clearly communicates the whole story of what we've done: we selected 5 promising bugs, picked 2 to implement, and made test cases for them. All of these have corresponding tickets in the 'Done' section.

Then it turned out that we couldn't fix the bugs, and this is reflected in the 2 fix tickets in the 'Rejected column'.

Afterwards, we verified that the bugs are indeed unfixable, and this is shown in the two verification tickets in the 'Done' column.