

# CSCD01: Deliverable 2

Team 32: CCF

Angela Zhu

Dasha Besshaposhnikova

Michelle Pasquill

Mohammed Osumah

# Table of Contents

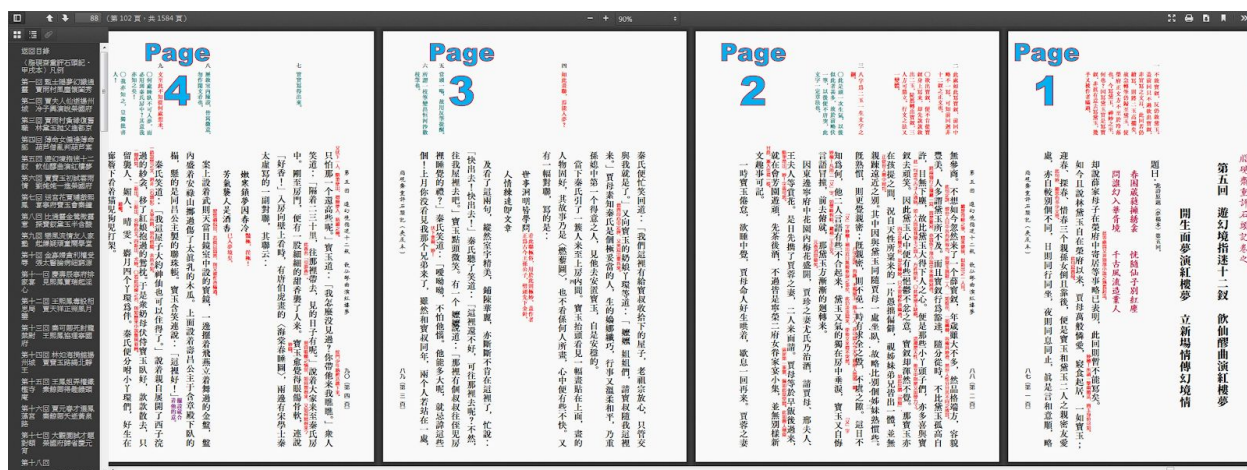
<b>Bug Issues</b>	<b>2</b>
ISSUE #1 - Horizontal Scrolling	2
ISSUE #2 - WAI-ARIA	4
ISSUE #3 - Converting Colors in 'web/viewer.css' to CSS Variables	5
ISSUE #4 - Tab-order	6
ISSUE #5 - CTRL-F Highlighting Problem	7
ISSUE #6 - Copy/Pasting Problem	9
<b>Rationale for selected issues</b>	<b>10</b>
Chosen Issue: Converting Colors in 'web/viewer.css' to CSS Variables	10
Chosen Issue: Tab-order	11
<b>Technical Commentary</b>	<b>12</b>
Chosen Issue: Converting Colors in 'web/viewer.css' to CSS Variables	12
Chosen Issue: Tab-order	13
<b>Software Development Process: Kanban</b>	<b>14</b>

# Bug Issues

## ISSUE #1 - Horizontal Scrolling


Link to issue: <https://github.com/mozilla/pdf.js/issues/11154>

This issue is a new function suggestion for horizontal scrolling. Currently there is an option to enable horizontal scrolling mode in PDF.js, but the direction of the pages goes from left to right. The user who raised the issue was suggesting a right to left scrolling mode for documents that are intended to be read from right to left, such as Chinese/Japanese/Korean vertical texts.




### How to reproduce the issue:

*Firefox (Can run in any other browser using pdf.js as an extension)*

1. Open any pdf with multiple pages
2. Enable horizontal scrolling by clicking on the  button in the top right corner and selecting Horizontal Scrolling

### Source Code

1. Clone the source code from link <https://github.com/CSCD01-team32/pdf.js/>
2. Install Node.js and install the gulp package (more information in README.md)
3. Use gulp server and open <http://localhost:8888>

4. Open any pdf with multiple pages from the test/pdfs folder
5. Enable horizontal scrolling by clicking on the  button in the top right corner and selecting Horizontal Scrolling

### **Estimate of amount of work to complete:**

We estimated that this feature would take 2-3 days of work and at least 2 developers working together since the feature would span multiple files and things like mouse/keyboard events, thumbnail ordering, and UI changes would need to take place. We thought this would be a good issue for our team to work on because it would mainly involve examining the code and implementing the reverse of the existing scrolling feature, which is something our team would be comfortable with. However, we did not select to implement this feature just because of the amount of time it would take and the amount of files that we would need to change.

Some of the files involved in this feature are:

pdf\_presentation\_mode.js - Deals with page switches according to different events.

pdf\_thumbnail\_viewer.js - renders the thumbnail view

ui\_utils.js - Deals with scrolling and scroll mode

## ISSUE #2 - WAI-ARIA

**Link to issue:** <https://github.com/mozilla/pdf.js/issues/9556>

This function was suggested by a user who utilizes screen readers. Pdf.js is not completely up to accessibility standards, and as such they have suggested additional support the viewer needs. Currently the viewer allows one to tab through its toolbar. It was suggested that keyboard shortcuts and arrow key functionality also be added to toolbar navigation. For example, adding a keyboard shortcut for toggling the sidebar navigation.

### **Estimate of amount of work to complete:**

We estimated that this feature would take [insert amount of time]. This is because the feature would require us to modify keyboard events in several files. We looked at this issue because accessibility features are important features and many of them are not very workload heavy. Implementing this accessibility feature requires modifying and adding on to existing features and thus manageable in our time frame. The reason we did not select this issue is because of how long it would take us to modify all the key events. Keyboard shortcuts would require us to modify existing keyboard functionality; the viewer allows for navigating the pdf with arrow keys, and thus would need to switch focus when using a screen reader on the toolbar. Keyboard shortcuts would also require us to work around existing shortcuts and create thorough documentation.

## ISSUE #3 - Converting Colors in 'web/viewer.css' to CSS Variables

Link to issue: <https://github.com/mozilla/pdf.js/issues/11572>

This issue was brought up by one of the top contributors to PDF.js. He was suggesting that the css file web/viewer.css should be altered so that all the colours used are represented by CSS variables, to make it easier for reskinning the viewer.

Currently the css colours are defined like this:

```
#viewerContainer.pdfPresentationMode:-ms-fullscreen {
  top: 0px !important;
  overflow: hidden !important;
}

#viewerContainer.pdfPresentationMode:-ms-fullscreen::-ms-backdrop {
  background-color: rgba(0, 0, 0, 1);
}

#viewerContainer.pdfPresentationMode:fullscreen {
  top: 0px;
  border-top: 2px solid rgba(0, 0, 0, 0);
  background-color: rgba(0, 0, 0, 1);
  width: 100%;
  height: 100%;
  overflow: hidden;
  cursor: none;
  user-select: none;
}
```

Using rgba() to set the colour value, and this method is used over 200 times in the file, even though the amount of colours used is limited, with a lot of colours being reused.

### Estimate of amount of work to complete:

We estimated that the work required for this fix is maximum 1 hour, since CSS variables are not very complicated and only involves declaring the rgba values used as variables in :root. This is one of the issues we chose to work on, and further justification can be found in the *Rationale for selected issues* section of this deliverable.

## ISSUE #4 - Tab-order

**Link to issue:** <https://github.com/mozilla/pdf.js/issues/9557>

This is an accessibility issue with the UI viewer in pdf.js. The issue covers two different problems, there are tabindex values greater than 0, and there are some tabbable elements that aren't encapsulated in an ARIA landmark.

First, the tabindex values allow for keyboard navigation for anyone who requires the use of a screen reader or who has mobility problems. Currently the tabindex values are manually set as different positive values ranging from 0 to around 100. The increasing values dictate the order that the elements should be tabbed through. This is bad practice in general, since inserting a single element into the tab order may require that all tab indexes for later elements be manually increased. For accessibility reasons this is bad since it can cause an unexpected tab order which can cause it to appear as if the items are skipped entirely.

Another problem brought up in this issue, is that there are no ARIA landmark roles in the viewer. ARIA landmark roles provide the screen reader user with the ability to skip to different main sections in the document, rather than having to tab through everything to get to the desired section. Currently there are no landmarks, meaning the user needs to tab through the entire navigation bar before they can actually read the pdf.

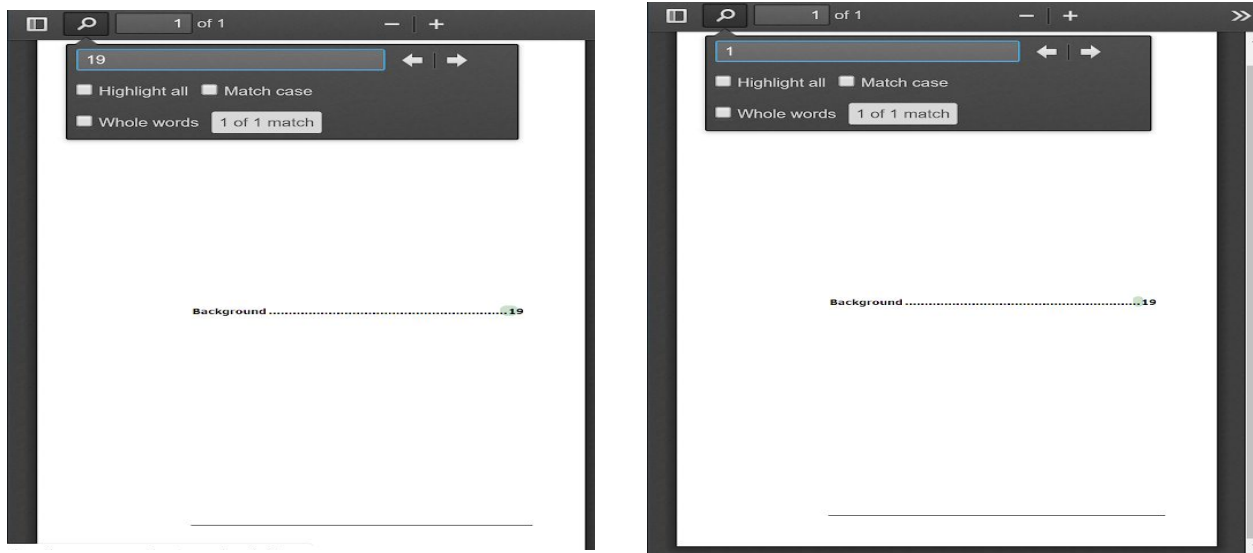
### **Estimate of amount of work to complete:**

The work for this issue would not be too much. Probably no longer than 30 minutes to find screen reader software and learn to use it for testing purposes. Then about 1-2 hours to change tabindex values, add ARIA landmarks, and to change the javascript to ensure that drop-down type menus have focus moved to them on opening.

## ISSUE #5 - CTRL-F Highlighting Problem

**Link to issue:** <https://github.com/mozilla/pdf.js/issues/9752>

There is an issue with the highlighting of text when searching for text in the pdf with the CTRL-F option. You can see in the screenshot of the pdf file below (bottom left) that we opened with pdf.js, when searching for the string “19” with CTRL-F, the highlighting is off. The string “19” in the pdf file should be highlighted, but instead you see that the string “.. 1” is highlighted instead. The highlighting does not match the text in this case. When searching for the string “1” it also wasn’t highlighted correctly. This is evident in the screenshot below (bottom right).



### How to reproduce the bug:

*Firefox (Can run in any other browser using pdf.js as an extension)*

1. Download the file issue9752.pdf in our github folder for deliverable 2
2. Open the file in a firefox browser
3. CTRF-F and search for strings “19” or “1”

### Source Code

1. Download the file issue9752.pdf in our github folder for deliverable 2
2. Clone the source code from link <https://github.com/CSCD01-team32/pdf.js/>
3. Install Node.js and install the gulp package (more information in README.md)
4. Use gulp server and open <http://localhost:8888>



5. Open the downloaded file
  6. CTRL-F and search for strings “19” or “1”
- Estimate of amount of work to complete:**

Initially we assumed it was a functional issue, where the highlighting was a little off. We initially thought it would only take approximately 2-3 hours worth of work to diagnose this issue, but as we continued to dive deeper into finding the areas of the code where the bug might be, we realized it was a bigger issue involving the rendering of the text layer, which could take days to fix.

When testing the code we made the text layer visible in order to see whether the highlighting was off on it. We discovered that the highlighting is correct in the text layer, so we're pretty sure the problem has to do with the text layer not being rendered correctly. In the screenshots below, you can see the pdf when the text layer is visible and when it is not. The light gray in the pdf in the bottom right screenshot is the text layer.

CRScube Solutions

**cubeTMF**

安全で、コンプライアントで、  
全てがクラウドに。

CRScube Solutions

**cubeTMF**

安全で、コンプライアントで、  
安全で、コンプライアントで、  
全てがクラウドに。

We suspect the issue has to be in the file `text_layer_builder.js` in the `src/web` folder, which imports `renderTextLayer` from `text_layer.js` in the `src/display` folder. In the images below, we can see the function `renderTextLayer` and we can see it just initializes a `TextLayerRenderTask` object and runs the `_render` method. In the image below to the right you can see the code for `_render`.

```

714  /**
715   * Starts rendering of the text layer.
716   *
717   * @param {TextLayerRenderParameters} renderParameters
718   * @returns {TextLayerRenderTask}
719   */
720  function renderTextLayer(renderParameters) {
721    var task = new TextLayerRenderTask({
722      textContent: renderParameters.textContent,
723      textContentStream: renderParameters.textContentStream,
724      container: renderParameters.container,
725      viewport: renderParameters.viewport,
726      textDivs: renderParameters.textDivs,
727      textContentItemsStr: renderParameters.textContentItemsStr,
728      enhanceTextSelection: renderParameters.enhanceTextSelection,
729    });
730    task._render(renderParameters.timeout);
731    return task;
732  }
733
734  return renderTextLayer;
735  })();
736
737  export { renderTextLayer };
738
739

```

```

796  _render: function TextLayerRender(timeout) {
797    const capability = createPromiseCapability();
798    let styleCache = Object.create(null);
799
800    // The temporary canvas is used to measure text length in the DOM.
801    const canvas = document.createElement("canvas");
802    if (
803      !PDFJSDev.is("unimplemented") ||
804      PDFJSDev.test("NOCENTRAL") ||
805      PDFJSDev.test("GENERIC")
806    ) {
807      canvas.mozOpaque = true;
808      this._layoutTextCtx = canvas.getContext("2d", { alpha: false });
809
810      if (this._textContent) {
811        const textItems = this._textContent.items;
812        const textStyle = this._textContent.styles;
813        this._processItems(textItems, textStyle);
814        capability.resolve();
815      } else if (this._textContentStream) {
816        const pump = () => {
817          this._reader.read().then((value, done) => {
818            if (done) {
819              capability.resolve();
820              return;
821            }
822            Object.assign(styleCache, value.styles);
823            this._processItems(value.items, styleCache);
824            pump();
825            capability.reject();
826          });
827        };
828        this._reader = this._textContentStream.getReader();
829        pump();
830      } else {
831        throw new Error(
832          "Neither 'textContent' nor 'textContentStream' + 'parameters' specified."
833        );
834      }
835
836      capability.promise.then(() => {
837        if (styleCache === null) {
838          // Render right away
839          render(this);
840          // Schedule
841          this._renderTimer = setTimeout(() => {
842            render(this);
843            this._renderTimer = null;
844          }, timeout);
845        }, this._capability.reject());
846      });
847    }
848  },
849

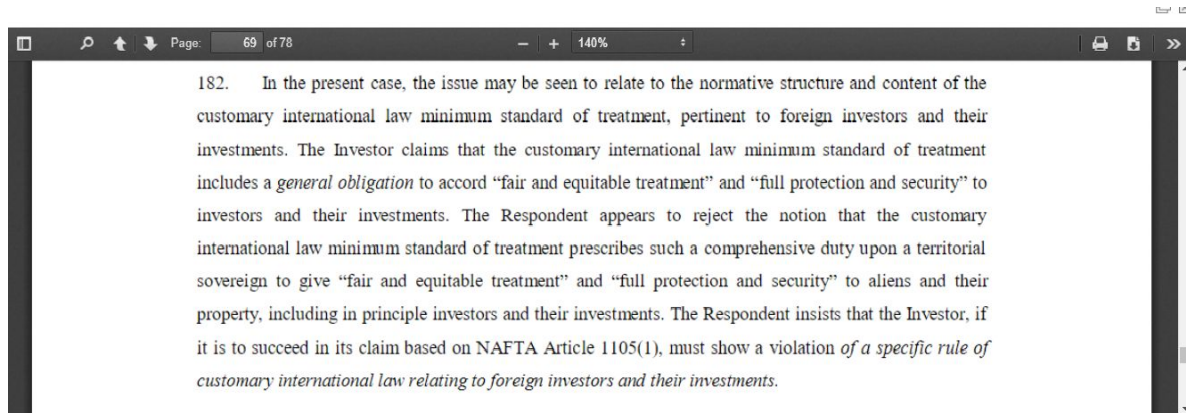
```

## ISSUE #6 - Copy/Pasting Problem

Link to issue: <https://github.com/mozilla/pdf.js/issues/10003>

This is an issue which seems to be brought up a few times on the issues list. When text from the PDF document is copied and then pasted in some text editing software, it creates random line breaks that do not appear in the original PDF. This is likely an issue with how the PDF is parsed and rendered, because when highlighting the text some spaces between words show up in the highlight while others do not. This problem not only affects people who are trying to copy text from PDF documents, but screen readers also have a problem since it will read text with random breaks between words.

The following images are from the issue on GitHub, and they show what the block of text looks like in PDF.js followed by how it looks when pasted into a Word document. There are random line breaks where there should not be, including in the middle of a word. Other similar issues give more examples of this, and the same occurred with our own testing of the issue.



183.  
The Tribunal considers that the issue relating to the structure and content of the customary international law minimum standard of treatment has not been adequately litigated, and that neither the Investor nor the Respondent has been able persuasively to demonstrate the correctness of their respective contentions. We are not convinced that the Investor has shown the existence, in current customary international law, of a general and autonomous requirement (autonomous, that is, from specific rules addressing particular, limited, contexts) to accord fair and equitable treatment and full protection and security to foreign investments. The Investor, for instance, has not shown that such a requirement has been brought into the corpus of present day customary international law by the many hundreds of bilateral investment treaties now extant. It may be that, in their current state, neither concordant state practice nor judicial or arbitral caselaw provides convincing substantiation (or, for that matter, refutation) of the Investor

### **Estimate of amount of work to complete:**

We are unsure of the exact amount of time it would take to fix this issue, but it would likely take at least a few days or probably longer. As a team we are not too familiar with the structure of PDF documents, or how to parse through them. Trying to fix this issue has the potential risk of breaking some other part of the parsing, which is one of the most important parts of the code. Additionally, it may be hard to completely fix this problem since badly formatted PDF files may still have this problem. Therefore, we chose not to fix this issue since the risks are too high and our unfamiliarity with parsing may lead to inaccurate time estimates.

# Rationale for selected issues

## Chosen Issue: Converting Colors in 'web/viewer.css' to CSS Variables

*See Issue #3 for more information*

We ended up choosing this issue instead of the Ctrl+F highlighting problem when we discovered that the cause of the highlighting problem was more complicated than we initially thought. This CSS variables issue, on the other hand, is very straightforward and allowed us to make a useful fix for PDF.js. With CSS variables, the code becomes much cleaner and easier to build upon for future developers. It also allowed us to fix a second issue on time for this deliverable.

The anticipated risk for this issue is that a developer had left a comment on the issue expressing concern that the CSS variables might add overhead to the loading of the pdf viewer. Upon doing some research we discovered that many sources state that CSS variables function well, and are much better than reusing the same values many times throughout CSS files. Since we won't be adding a large amount of variables anyway, we decided it wouldn't make much of a difference to the overhead.

### **Acceptance test**

Since there is no way of coding an acceptance test for css variables, it suffices to open web/viewer.js and in the :root pseudo class with the variables under the color comment, search each of the variables in the file. If you see the color variables being reused, and the pdf viewer looks unchanged from the previous version, then the implementation has been successfully completed.

## Chosen Issue: Tab-order

*See Issue #4 for more information*

We chose to work on this issue since it would not take too long to fix and some of our team members have had experience with making accessible web content before. We also believe that accessibility is an important issue, and this problem could make PDF.js frustratingly unusable for people who need to use screen readers. Additionally, it is bad practice to put positive tab indexes, so by fixing this it makes the viewer easier to change in the future.

Anticipated risks of fixing this issue are that we needed to change all tabindex values to 0, meaning the tab order may not follow expectations if the DOM elements aren't organized well. Additionally, some javascript may modify tabindex values directly, and changing them could cause other problems. These risks can be mitigated by reordering DOM elements to follow a logical order, as well as checking the javascript for any potential tabindex or ARIA landmark manipulations.

### **Acceptance testing**

From our research, there does not seem to be any way of automating testing for ensuring a logical tab order or checking ARIA landmarks. Instead, we manually used a screen reading plugin for Google Chrome called "ChromeVox". This plug in allowed us to check that the tab ordering makes sense and also to test that the ARIA landmarks are working. Using this to test, we were able to find any issues where the DOM does not flow in a logical left to right order and modify it to fix this. This also allowed us to find an additional problem with the fact that opening a sub-menu sometime does not move focus to that sub-menu. This means that someone with a screen reader may click "Enter" to open a menu, expecting the next tabbable element to be from the sub-menu when it is not, leading to confusion.

Additionally, we used another plug-in called "axe" which is a tool with automated accessibility testing which can be used in the in-browser developer tools. This tool checks any violations of web accessibility guidelines and tells you how many there are and what element in the DOM violates it. The tool provided feedback regarding the positive tabindex values and lack of ARIA landmarks. Screenshots of the tool being used can be found on GitHub under the deliverable 2 acceptance test folder.

# Technical Commentary

## Branches

The fixes we made can be viewed in their respective branches in the forked repository.

Issue 3: Converting Colors in 'web/viewer.css' to CSS Variables: **css-variables-fix**

<https://github.com/CSCD01-team32/pdf.js/tree/css-variables-fix>

Issue 4: Tab-order: **accessibleTabIndex**

<https://github.com/CSCD01-team32/pdf.js/tree/accessibleTabIndex>

## Chosen Issue: Converting Colors in 'web/viewer.css' to CSS Variables

*See Issue #3 for more information*

For this issue, the only file we needed to update was web/viewer.css. We converted the numerous repeated rgba() to define colours as variables. These changes make the design of the code for this file in the project more cleaner and easier for reskinning.

It shouldn't affect the functionality of the project because everything should still work as it is, seeing as we just converted the repeated colours to variables, but didn't change the colours for the classes and IDs.

## Chosen Issue: Tab-order

*See Issue #4 for more information*

For this issue, we had to modify the HTML file for the viewer as well as the javascript.

All tabbable elements (with the exception of the page thumbnail viewer which uses aria labels) were managed using tabindex. However, with the amount of elements, the values were going to incredible high numbers. Notably, elements in the find bar had values in the 90s while the find button for toggling the bar had a value under 10. This caused inconsistency with tabbing order. For example, the elements in the sidebar

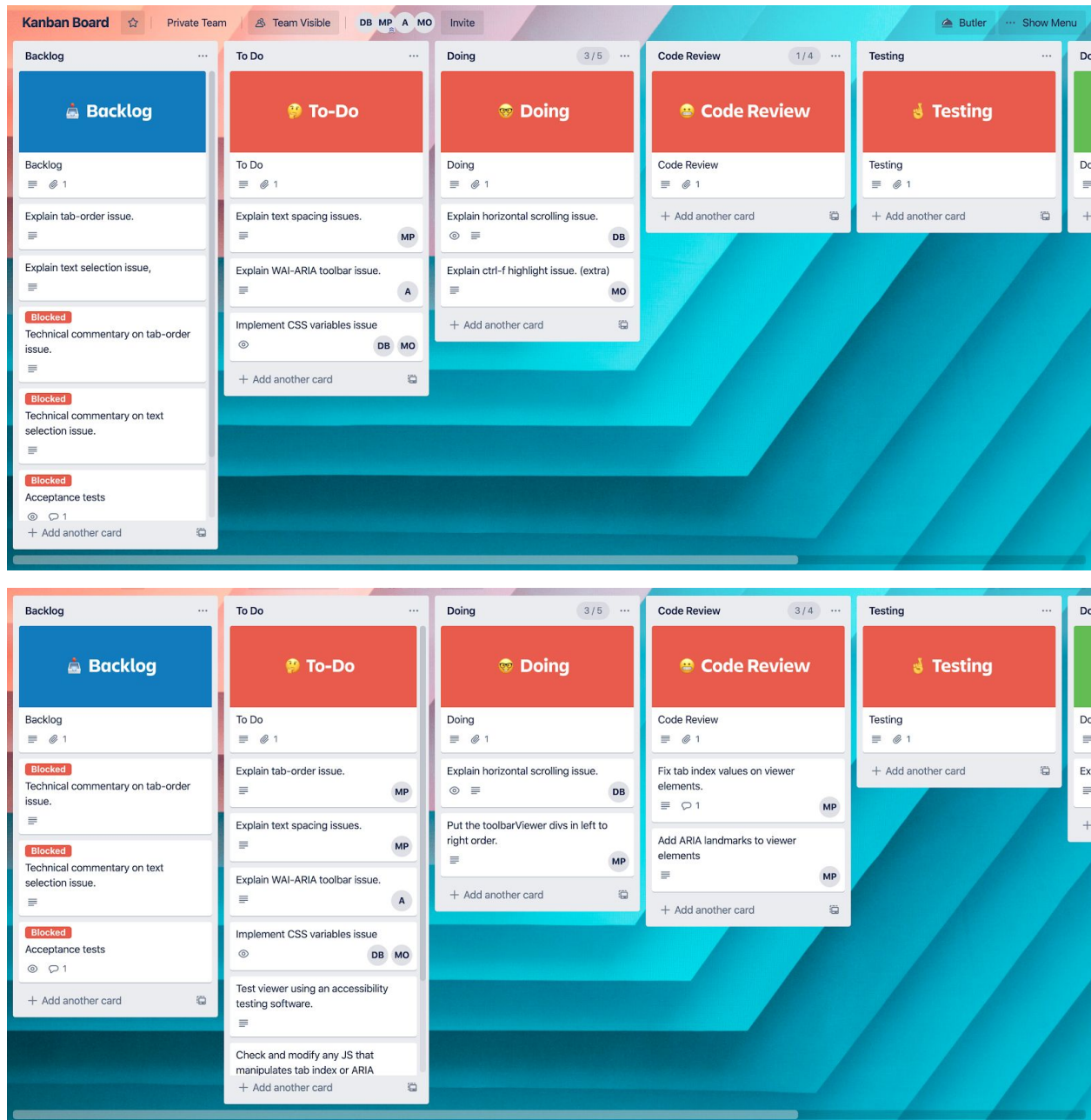
menu had high numbers, and when opened, the tab would continue through the toolbar instead of going into the newly opened sidebar menu.

The HTML file modified was web/viewer.html. We needed to change the 'tabindex' values to follow WAI-ARIA guidelines. There were 43 instances where 'tabindex' was set and needed to be changed to "tabindex=0". Making them all "tabindex=0" brings them all into the tab order on the page without specifically specifying an order, the elements now just follow the order which they appear in the DOM. An additional change needed to be made for the organization of the DOM. Their toolbar has three sections but they were not represented in order in the DOM, so they needed to be reorganized so the tab order makes sense.

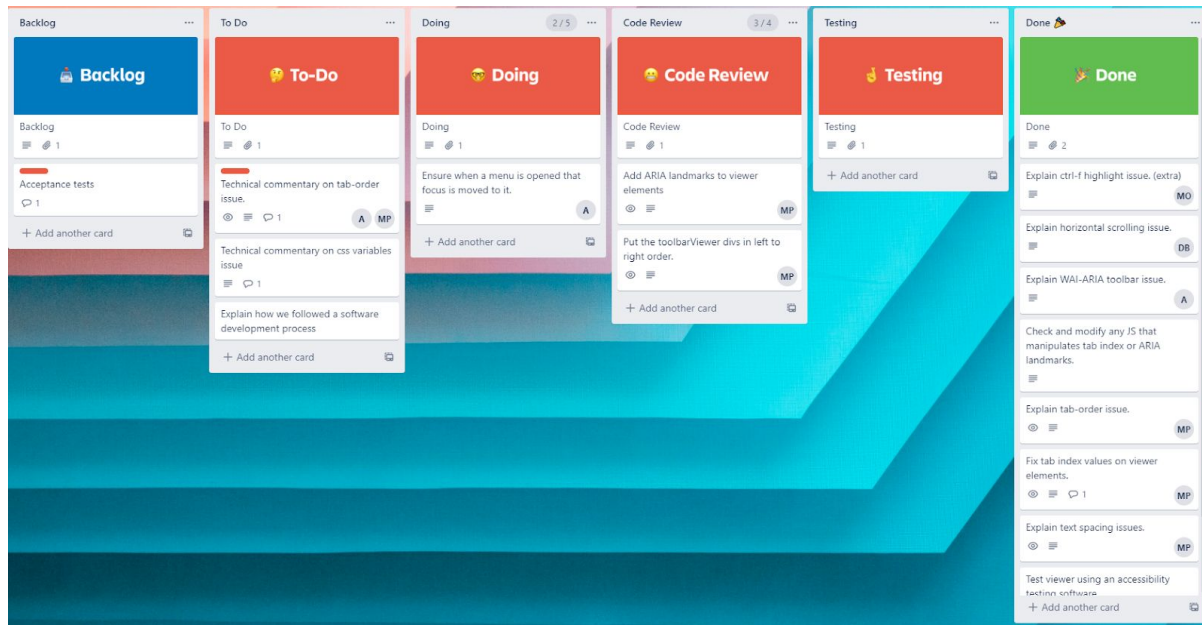
Additionally, ARIA landmark roles were added to two elements in the HTML document. The 'toolbar', which is the top bar in the viewer, was given 'role="navigation"'. The 'viewerContainer', which contains the display for the PDF page, was given 'role="main"'. This allows the screen reader user to quickly skip to the navigation bar or the main document.

While some javascript files (such as pdf\_thumbnail\_viewer.js) did modify ARIA labels in the html file, ARIA landmarks were not included and tabindex attributes were only handled in the HTML. Due to this, no other part of the JS was affected by the HTML changes.

# Software Development Process: Kanban







We used Trello for our Kanban board, so we could communicate the tasks that were being worked on. In making our tasks, we had a meeting where we discussed the basic tasks that needed to be worked on and put them on the Kanban board backlog. From there, we broke the tasks up into smaller tasks to make it easier to communicate exactly what needs to be done. Each task has a basic title explaining it, and a description of what needs to be done as well as any relevant links pertaining to the task. Additionally, we put the blocked label on certain tasks that require others to be completed first. Such as in the image below where the technical commentary was blocked until the issue was fixed. The comment underneath the card links to the card that is blocking it.

Technical commentary on tab-order issue.

in list [To Do](#)

MEMBERS

A MP +

LABELS

Blocked +

Description

Edit

A technical commentary on how our changes affect the code of the project.  
To be completed after tab-order issue fixed: [mozilla/pdf.js: Issue #9557](#)

Activity

Hide Details

MP

Write a comment...

MP

Michelle Pasquill joined this card

an hour ago

MP

Michelle Pasquill 2 hours ago

Blocked by [Ensure when a menu is opened that focus is moved to it.](#)


👤

[Edit](#) - [Add Link as Attachment](#) - [Delete](#)

The Kanban process was useful for us because it gave us the flexibility to add or modify tasks during the development cycle. The work in progress limits helped us to make sure that we were not focusing on more than one task at a time. By limiting the tasks to only one at a time per person, it was easier to see what tasks are actually being worked on. The visualization of the board was really useful in seeing where problems in the workflow exist. We were able to see the number of tasks that still need to be completed which helped us be more efficient. If any other issue arose and someone decided a new task needed to be added, the descriptions on the cards were a good source to make communication more efficient. Rather than needing to explain the issue to whoever decides to work on the task, the team member can just read the description on the card, and can see who created it if they have any further questions. For instance the following task has a description and a link to an article on how to fix this, so that it is easier to see what needs to be done by whoever picks up this task.



## Fix tab index values on viewer elements.

in list [Done](#)  

### MEMBERS

MP



### Description

Edit

Currently HTML elements have tabindex values greater than zero, this is bad for anyone who uses screen reader software.

<https://dequeuniversity.com/rules/axe/3.2/tabindex>