CSCD01 Deliverable 3

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

Table of Contents

Table of Contents 1

Promising Features 3

Security: Ignores protection from Copy 3

Namespace classes and ids 4

Selected Feature — Namespace classes and ids 5

The Feature We Didn’t Select —
Security: Ignores protection from Copy 5

Implementation Plan 6

Acceptance test suite 8

Test cases: Namespace classes and ids 8

Test Case 1: Control (Disabled, No Namespace Conflicts) 8

Test Case 2: Enabled, No Namespace Conflicts 9

Test Case 3: Enabled, Namespace Conflicts Exist 9

Revised overall architecture of project 10

Promising Features

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 is a concern for some enterprise users where documents have to respect the permissions, or else they have to use other solutions to view PDF documents.

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.

This bug was opened and discussed 8 years ago, and had a "fix" implemented. However the fix was later backed out due to the lack of approval from the firefox product team, and there hasn't been any follow up with the fix. The most recent comment from one of the contributors is that, to fix this issue properly, it needs to be set as a preference that can be enabled and disabled in firefox/PDF.js.

In order to implement it properly, there needs to be changes in the core layer where it reads the permission of the PDF document, and exposes that in the display layer. There are similar implementations already with reading information of the document, so this should not be too difficult. There also needs to be a preference option in the viewer to respect the permissions as per request from the contributors. The viewer will need to read whether copy protection is enabled in PDF.js and check if the document is copy protected. If both are true, then the viewer can achieve copy protection via "-moz-user-select: none" to disallow users to select and copy text on the text layer.

Namespace classes and ids

Issue: 6706

HTML and CSS Classes and ID names used by PDF.js are too generic and when PDF.js is used within another website, these generic class and ID names may conflict with the existing ones on the site. For example #LoadingBar is an extremely generic name and many websites may use it as part of their own application. Because of this you may run into CSS class name conflicts.

Unfortunately, a fix is not as simple as prefixing all of the class and IDs with “pdfjs” or something similar. Any extension of PDF.js that uses query selectors to specifically select classes or IDs on the page would no longer function if ‘pdfjs’ were prefixed to the front of every class and ID.

The solution would have to be to make this feature able to be toggled on and off. This would prevent PDF.js from breaking in both situations. A proposed idea is a boolean value that when enabled, loads the system with prefixes on every class and ID and when disabled loads the system normally.

This bug was opened 5 years ago and no progress has yet to be made on it. Some ideas were proposed by the one who opened the issue and suggested things like, a duplicate style sheet with the modified names, popups asking whether to enable or disable this function, etc.

Selected Feature — Namespace classes and ids

We chose to implement this feature for three reasons.

The first is that the discussion thread for the issue in Github gives a clear overall outline of how to go about the implementation. The solution suggested is one we understand, and it is endorsed by the project maintainers themselves, so this gives us a strong starting point.

The second reason is that the implementation itself requires knowledge of Javascript's language features and how it interacts with DOM elements. Since all members of our team have taken CSCC09 in the past, and most of us have used Javascript extensively, we have a good grasp on how to use Javascript functions and closures to make the necessary edits to pdf.js' frontend data flow.

The third and final reason is that this feature seems like one that would bring significant value to users of pdf.js. One of the project maintainers said that it is something worth working on, and other Github users have commented on the issue thread throughout the years saying that they are still interested in a solution. Even besides that, the core issue -- that pdf.js' CSS namespace can conflict with the namespace of web pages that embed pdf.js -- is one that obviously impacts every single developer who embeds pdf.js in their websites.

The Feature We Didn't Select — Security: Ignores protection from Copy

We decided not to implement this feature for two reasons.

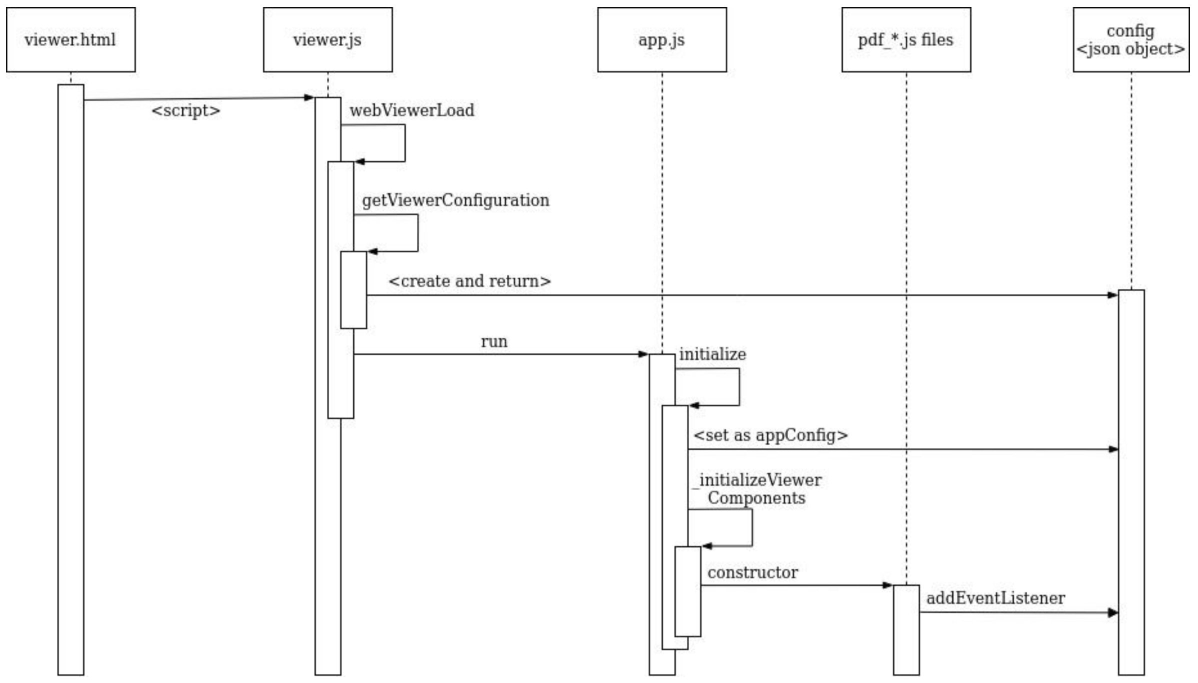
The first is that the actual implementation of the fix would be too straightforward for this deliverable. As detailed in the preceding sections, all that would need to be changed in pdf.js is a boolean toggle to apply a CSS tag to the outputted HTML text. The most labour-intensive parts will be finding the exact location in the code to modify and writing documentation on how to enable or disable this feature. The actual code implementation is very minimal, so this feature does not really qualify as the kind of significant feature required for this deliverable.

The second reason is that this feature does not provide significant value to users of pdf.js. This was brought up in a Bugzilla thread related to the feature request. The specific comment that we want to highlight is this one. In sum, adding DRM like copy protection to pdf.js would conflict with Mozilla's free and open philosophy. Additionally, adding copy protection to pdf.js or even Firefox as a whole would be pointless, because a user could just open the PDF file in another PDF viewer that ignores copy protection.

Implementation Plan

As suggested in the discussion thread for Issue #6706, the first step will be to duplicate both **viewer.html** and **viewer.css** as **viewer-prefixed.html** and **viewer-prefixed.css**, respectively. These two files will be edited so that all CSS classes have the ‘pdfjs’ prefix added to them. These two files can either be created manually, or (as also suggested in the discussion thread) they can be automatically generated via a Gulp build script.

After that, the Javascript files will also need to be updated to reference the updated CSS namespace identifiers. The current flow for pdf.js’ web viewer setup is illustrated in the UML sequence diagram below:



The key component for Issue #6706 is the **config** object that is created by **getViewerConfiguration** in **viewer.js**. This object uses Javascript’s getElementById method to obtain references to all of the HTML DOM elements that make up the viewer’s UI in **viewer.html**.
The config object is then sent to the **initialize** method inside **app.js**, where it is stored as app.js’ appConfig property. Then, as the initialize method instantiates all other viewer helper classes within the pdf_* files, it passes appConfig to those classes’ constructors.
The helper classes then use appConfig to directly interact with the relevant DOM elements. For example, PDFFindBar (in pdf_find_bar.js) attaches event listeners to appConfig.findField, appConfig.highlightAll, and so on.

From this, we can see that the necessary changes will need to be made to the config object. Specifically, we have to modify its creation so that when the relevant flag is enabled, each statement inside of the object like this:

    document.getElementById(“<elementId>”)

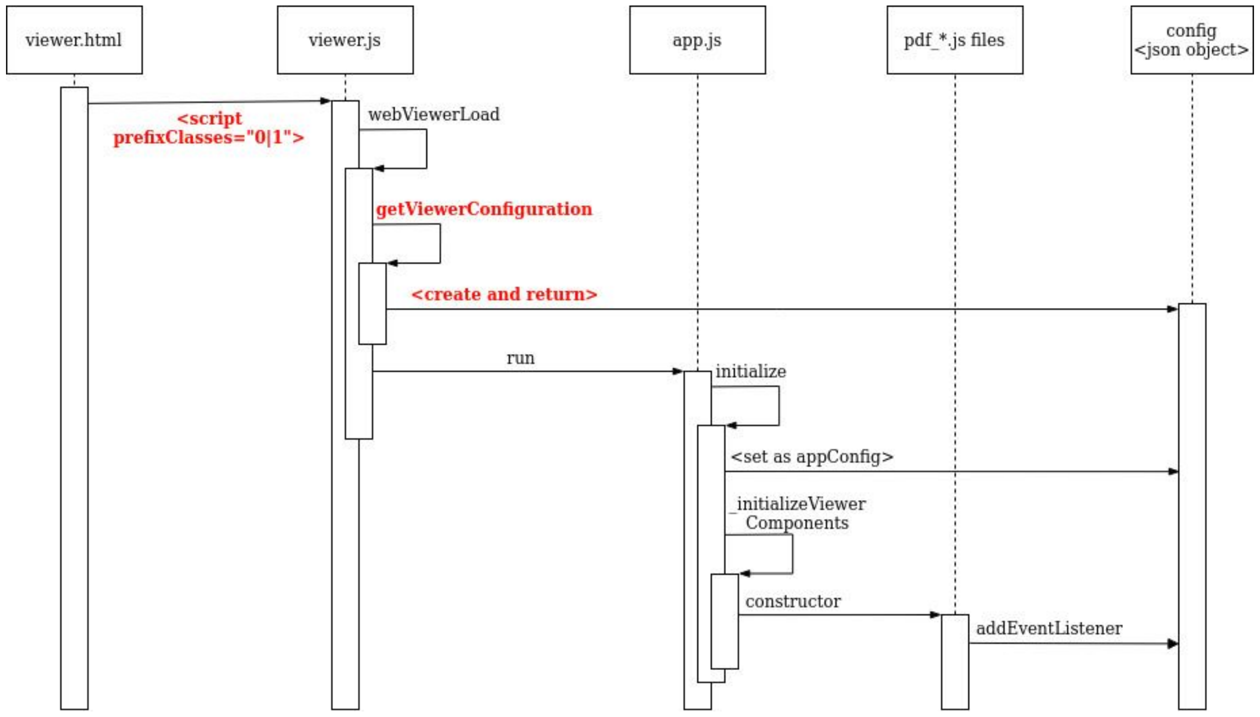
becomes

    document.getElementById(“**pdfjs**<elementId>”)

Since **getViewerConfiguration** is the method that creates and returns the config object, these modifications will take place inside of it. Based on the value of a flag variable, it will either return the original config object, or the prefixed config object that we described above.

However, we have to make additional modifications to define and transport that flag variable as well. We want this flag variable to be something that can be easily seen and modified by a user looking to embed the pdf.js viewer into their website.
According to the pdf.js wiki page on embedding, the user copies the entire web folder into their project’s folder, then includes **viewer.html** directly into their web pages.
As such, the best option would be to edit the **<script>** tag in **viewer.html** that loads **viewer.js** so that it takes a ‘prefixClasses’ attribute. Then, in viewer.js, to get the ‘prefixClasses’ attribute from the <script> tag, we can use this solution: https://stackoverflow.com/a/21253793.

The updated flow is illustrated below, with modified components highlighted in red.



Acceptance test suite

Since Issue #6706 deals with styling and the appearance of the whole pdf.js viewer, we decided 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.

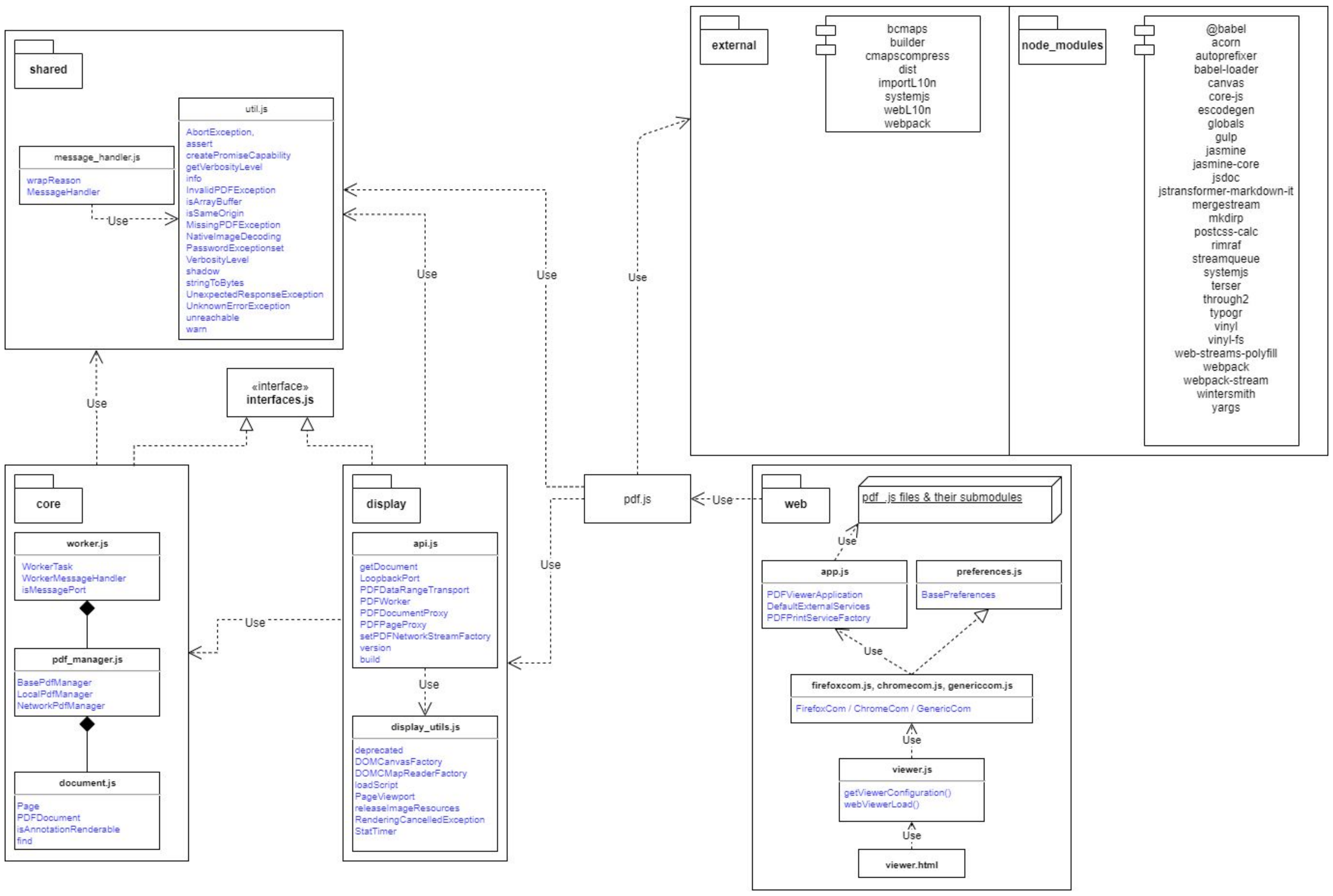
Test Case 2: Enabled, No Namespace Conflicts

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

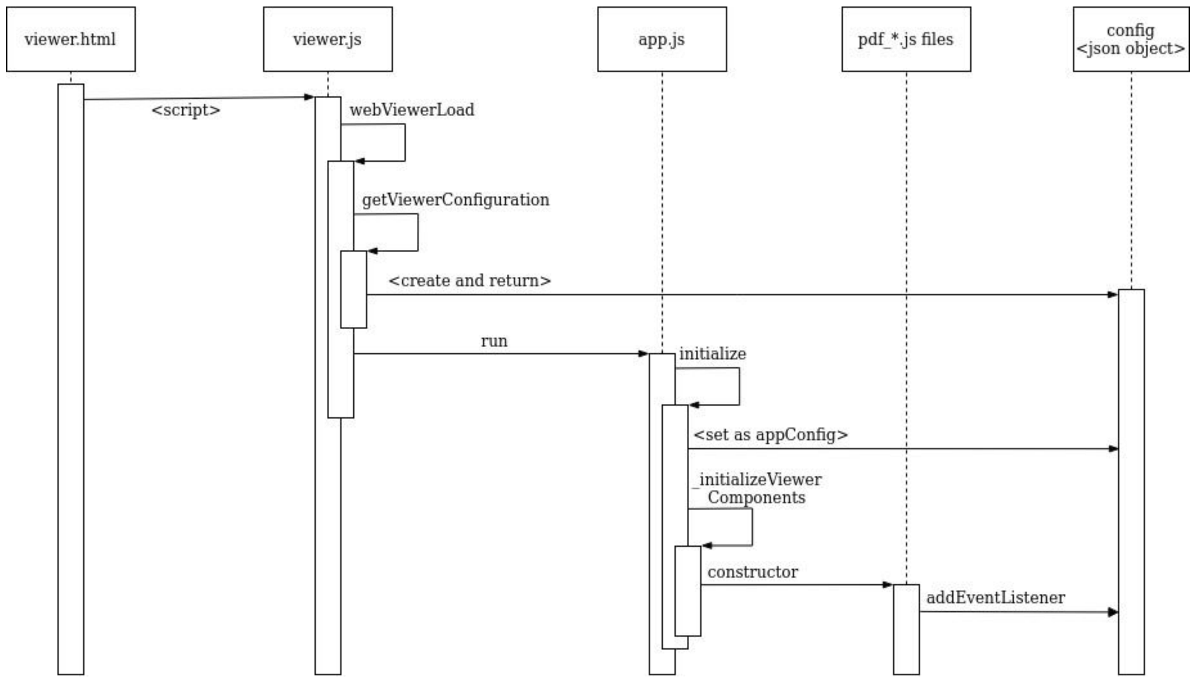
1. In test/6706/3, open index.js in an editor.
2. Find the location where ‘PDFJS.usePrefixedSelectors’ is set.
3. Set PDFJS.usePrefixedSelectors = 1.
4. Launch test/6706/3/index.html in another browser tab or window separate from the one used for Test Case 1.
5. Wait for pdf.js to finish loading.
6.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

Revised overall architecture of project



As can be seen above, our understanding of the overall architecure of pdf.js has not changed. This is because the files relevant to Issue #6706 are already included in our diagram, and our investigation did not show any mistakes that needed to be corrected. We attribute this to the strength of the architectural implementation in pdf.js' web layer, in that the files all adhere to the layer's high-level architectural ideas and don't deviate or introduce unnecessary complexity.

Where our understanding has improved, however, is in the data flow between components in the web layer. Previously, we only had a intuitive understanding of how the DOM elements were transported between viewer.html, viewer.js, app.js, and the rest of the web layer files.
After working on this deliverable, we are now able to accurately map the flow of the config object that contains references all of the DOM elements. Our findings are summarized in the UML sequence diagram that we produced, and which is shown on the next page for emphasis.


viewer.html
viewer.js
app.js
pdf_*.js files
config <json object>
<script>
webViewerLoad
getViewerConfiguration
<create and return>
run
initialize
<set as appConfig>
_initializeViewer Components
constructor
addEventListener