

CSCD01 Project Deliverable 4

Issue 1

Name: Feature request: Displaying a list of search results

Link: <https://github.com/mozilla/pdf.js/issues/7605>

Documentation:

Originally when searching for a word or phrase in a pdf when using pdf.js you would only be able to search the words one at a time, and would have to press the next/previous buttons to navigate through the different results like the traditional search functions found in browsers or in adobe acrobat reader.

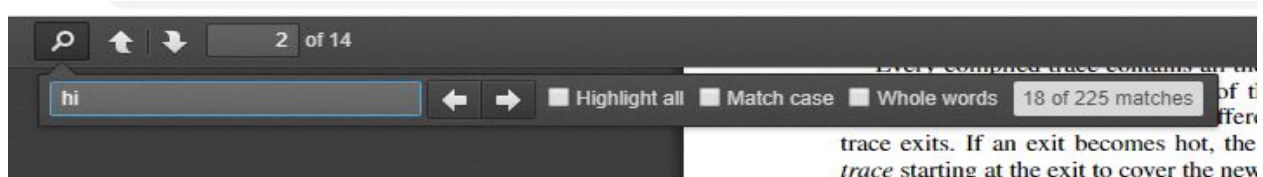


Fig. 1 The view of the search before fix was implemented

To start we had to find out how pdf.js actually stores the search results, and it is all done in the **pdf_findbar_controller.js** file, where it stores the index of the word in the current page, and then scrolls to that part of the page.

The next step was to add the area for displaying the search results which was done by adding an html element in the **viewer.html** document along with the rest of the findbar elements.

The final step was to style the “display all” area, which was accomplished by altering the **viewer.html** and **viewer.css** files.

The changes were very different than what we planned them to be, only relying on extensive changes to the three aforementioned files instead of the five previously thought in Deliverable 3. The functions that were added and which files they were changed in can be seen in the UML below:

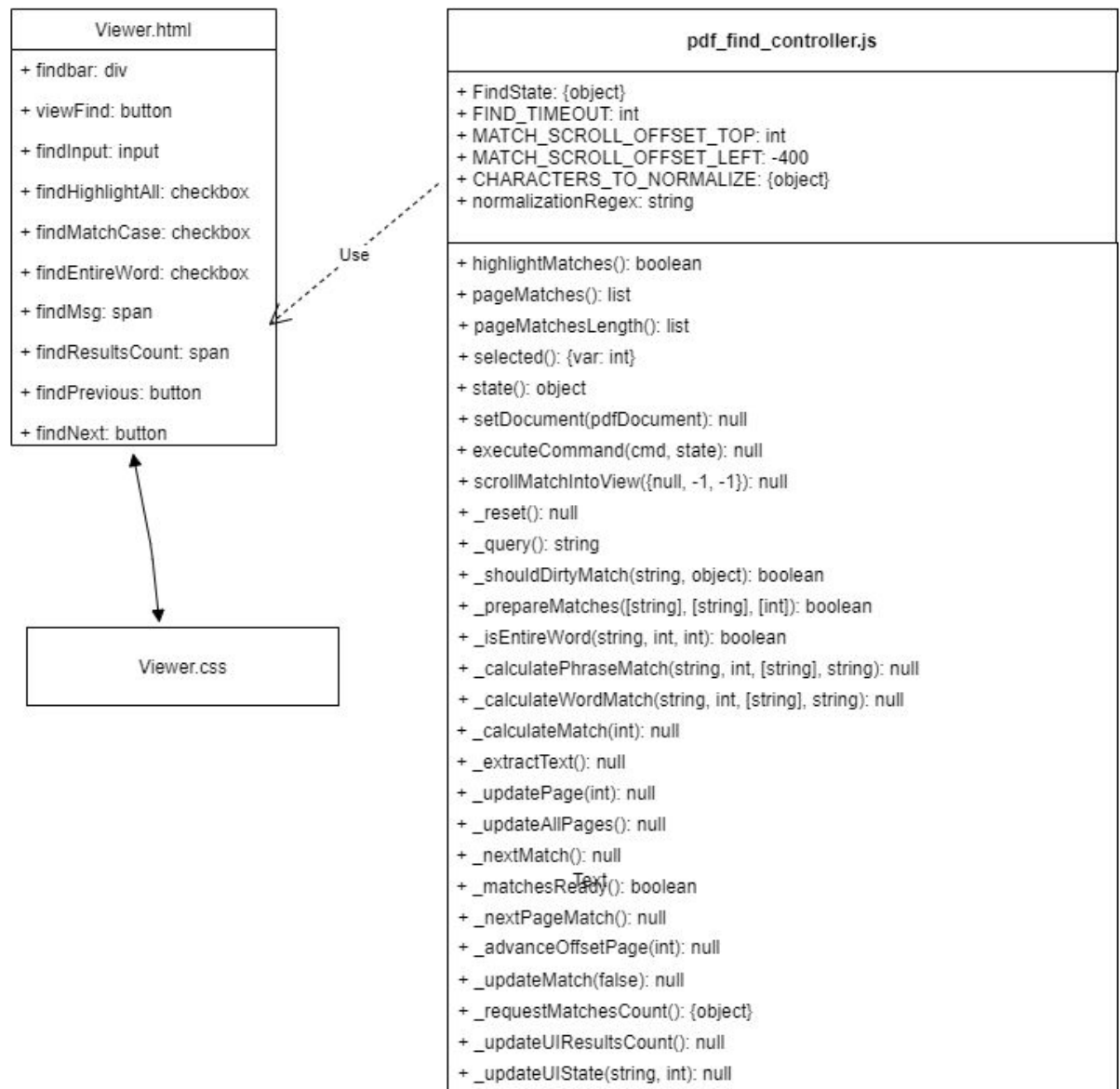


Fig. 2 UML for changes to display list of search results

With the changes this is how the search field now looks:

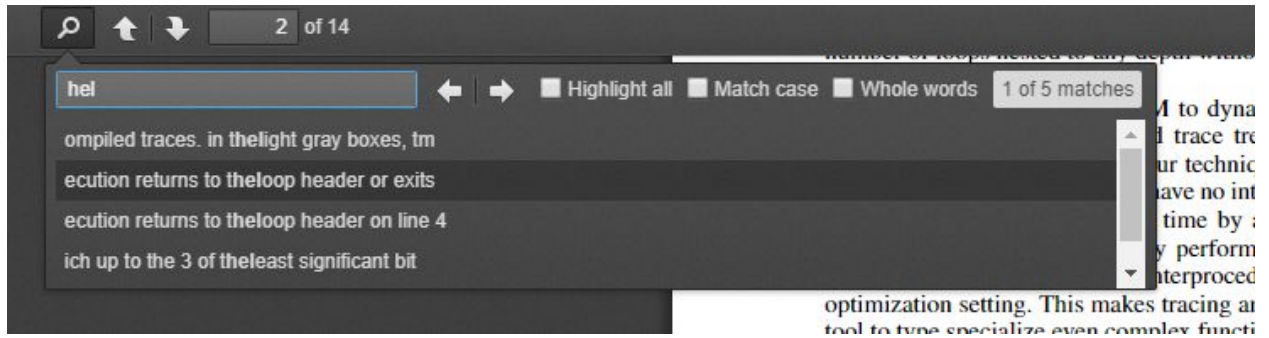


Fig. 3 Search bar with a “display all results” section

Acceptance Tests

Test 1:

1. Open pdf.js and view the default pdf file available.
2. Use CTRL+F to open up the “Find in document” menu.
3. Enter the word “provide” in the search bar and hit the “Enter” key.
4. There should be a series of search results with context displayed in the UI (five results)
5. Click the “next” button to navigate to the next result
6. Click the “previous” button twice to navigate to the result before the one originally found
7. Click the 4th result.

Test succeeds if the user is brought down to the 4th search result in the general document, and the next and previous buttons work as expected from before the fix.

Test 2:

1. Open pdf.js and view the default pdf file available.
2. Use CTRL+F to open up the “Find in document” menu.
3. Enter the word “provide” in the search bar and hit the “Enter” key.
4. There should be a series of search results with context displayed in the UI (five results).
5. Click the 3rd result.
6. Use the next key to navigate to the 4th result in the document.

Test succeeds if the user is brought down to the 4th search result in the general document, and using the find all display does not mess up the original navigation.

Acceptance tests did not need to be modified too much from the previous deliverable as they were comprehensive enough to test what the customer (in this case the issue creator) wanted to see. Simply had to add an extra section in each test to make sure extra attention was being paid to making sure nothing broke due to regression issues, by checking that the default functionality still worked as expected.

Issue 2

Name: No spaces between words when copying text

Link: <https://github.com/mozilla/pdf.js/issues/10640>

Documentation:

One issue with PDF.js at the time of writing this was that when highlighting text across line breaks or formatting changes (italics for example), when copying the highlighted text, there would be no spaces where the user would reasonably expect them to be. In addition, when words are hyphenated across lines and these words are highlighted and copied, the hyphen within the word is also copied, instead of copying as an unhyphenated word.

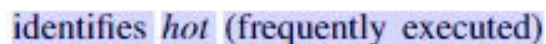


Fig. 4 Example of text that would be copied incorrectly, will copy as “identifieshot(frequently executed)” with no spaces

PDF.js creates a transparent text layer above the text rendered to the canvas so the user can interact with the text and highlight it. The text layer creates HTML spans and encapsulates the text content of the document into these spans, creating new spans generally for each line and to change text formatting spaces between lines and text formatting were missing because there were no trailing space characters within these spans, nor were there leading whitespace characters, so when highlighting across spans, there was whitespace between the start and end of different spans. Words

hyphenated across lines also simply left a hyphen at the end of one span and performed no processing on the span to remove the hyphen from the text layer.

The fix adds a function “_processStrEnding()” to **src/display/text_layer.js** that processes the ending of the text content of each span to add trailing whitespace and remove trailing hyphens when appropriate. This function is called in the “appendText()” function in the same file, and pre-processes text that would have originally just been placed directly into the span

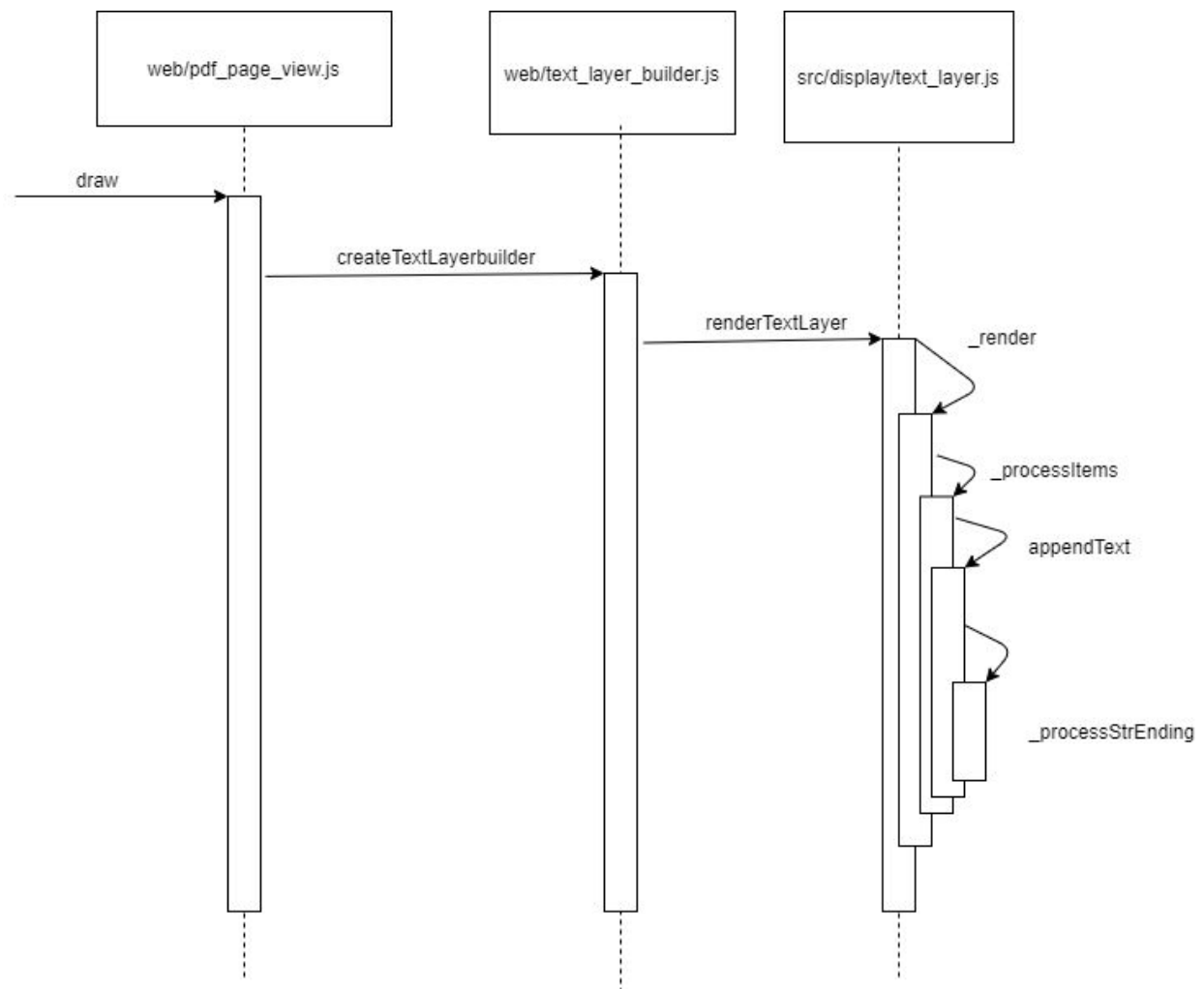


Fig. 5 Sequence diagram of the call stack when `_processStrEnding()` is called

Acceptance Tests:

Test 1:

1. Open pdf.js and view the default pdf file
2. Highlight and copy the text as shown below

Abstract

Dynamic languages such as JavaScript are more difficult to compile than statically typed ones. Since no concrete type information is available, traditional compilers need to emit generic code that can handle all possible type combinations at runtime. We present an alternative compilation technique for dynamically-typed languages that identifies frequently executed loop traces at run-time and then generates machine code on the fly that is specialized for the actual dynamic types occurring on each path through the loop. Our method provides cheap inter-procedural type specialization, and an elegant and efficient way of incrementally compiling lazily discovered alternative paths through nested loops. We have implemented a dynamic compiler for JavaScript based on our technique and we have measured speedups of 10x and more for certain benchmark programs.

Categories and Subject Descriptors D.3.4 [Programming Languages]: Processors — Incremental compilers, code generation.

General Terms Design, Experimentation, Measurement, Performance.

Keywords JavaScript, just-in-time compilation, trace trees.

1. Introduction

Dynamic languages such as JavaScript, Python, and Ruby are popular

and is used for the application logic of browser-based productivity applications such as Google Mail, Google Docs and Zimbra Collaboration Suite. In this domain, in order to provide a fluid user experience and enable a new generation of applications, virtual machines must provide a low startup time and high performance.

Compilers for statically typed languages rely on type information to generate efficient machine code. In a dynamically typed programming language such as JavaScript, the types of expressions may vary at runtime. This means that the compiler can no longer easily transform operations into machine instructions that operate on one specific type. Without exact type information, the compiler must emit slower generalized machine code that can deal with all potential type combinations. While compile-time static type inference might be able to gather type information to generate optimized machine code, traditional static analysis is very expensive and hence not well suited for the highly interactive environment of a web browser.

We present a trace-based compilation technique for dynamic languages that reconciles speed of compilation with excellent performance of the generated machine code. Our system uses a mixed-mode execution approach: the system starts running JavaScript in a fast-starting bytecode interpreter. As the program runs, the system identifies *hot* (frequently executed) bytecode sequences, records them, and compiles them to fast native code. We call such a sequence of instructions a *trace*.

3. Paste into a text document

Test succeeds if it pastes

“identifies *hot* (frequently executed)”.

Note, the spaces surrounding the word “hot”

Test fails if it pastes anything else, originally would have pasted

“identifies*hot*(frequently executed)”

With no spaces surrounding the word “hot”

Test 2:

1. Open pdf.js and view the default pdf file
2. Highlight and copy the text as shown below

Abstract

Dynamic languages such as JavaScript are more difficult to compile than statically typed ones. Since no concrete type information is available, traditional compilers need to emit generic code that can handle all possible type combinations at runtime. We present an alternative compilation technique for dynamically-typed languages that identifies frequently executed loop traces at run-time and then

3. Paste into a text document

Test succeeds if it pastes

“We present an alternative compilation”,

The test fails if it pastes anything else, originally would have pasted as

“We present an al-ternative compilation”,

With erroneous hyphen in the word ‘alternative’.

Test 3:

1. Open pdf.js and view the default pdf file
2. Highlight and copy the text as shown below

Abstract

Dynamic languages such as JavaScript are more difficult to compile than statically typed ones. Since no concrete type information is available, traditional compilers need to emit generic code that can handle all possible type combinations at runtime. We present an al-

3. Paste into a text document

Test succeeds if it pastes

“Information is available”

The test fails if it pastes anything else, originally would have pasted as

“Informationis available”

With no space between the words “information” and “is”.

Unit Test Suite

We did not make unit tests for this part of deliverable as they were not suitable for the fixes that we were implementing. For the first issue, our change was mostly on the front

end, and there was no functionality that could be tested without having to mock large portions of the code, as pdf.js was not built for displaying all search results, but rather paginating through results on the fly. Because of this, we did not have the resources to effectively write unit tests. For the second issue, the fix was done in the text layer, and this layer is hard to make unit tests for without, again, mocking large amounts of the codebase. There was also nothing to base the unit tests off of, as the original project doesn't have any unit tests for the text layer.

Software Development Process

Kanbanized Agile was used to moderate the development process of deliverable 4 much in the same way it was for deliverable 2. Over the course of development, various cards were added to the Kanban for different parts of the development process. During the first section of the process, finding possible issues on the pdf.js GitHub, a new card was created and added to the "Available" column of the Kanban for each issue the team decided to document. Each team member would assign one or two of these cards to themselves and document the issue, moving the card from "Available" to "Working" to "Pending Review" until the documentation of the issue was deemed complete by the team and moved to the "Done" column. At the same time, a card was created and assigned to the entire team to decide which two issues would be implemented. After deciding on these two issues, two new cards were added to the board for these issues, and team members chose which issue they wanted to work on, with a limit of two members to each issue. Cards were also created for creating the documentation of the tests done for each issue implemented. Over the course of the implementation of the issue fixes, all team members worked on the documentation of these fixes. All members were assigned a card for documenting implementation in this file here, which, once all fixes were complete and fully documented, was reviewed by the team and moved to the "done" column.