Testsuite and Harness for Browser Based Editing

Dave Raggett
W3C/ERCIM
2004, route des Lucioles
Sophia Antipolis - France
dsr@w3.org

ABSTRACT

The Web is still awkward when it comes to online editing. It is time to fix that, and make it easier for developers to create smarter browser based editing tools. This short paper presents work on a test framework for a cross browser open source library for browser based editing. The aim is to encourage a proliferation of different kinds of browser based editing for a wide range of purposes. The library steps around the considerable variations in the details of browser support for designMode and contentEditable, which have acted as a brake on realizing the full potential for browser based editing.

Keywords: testing; browser-based editing, collaborative editing

1. INTRODUCTION

Tim Berners-Lee envisioned the Web as something people could both read and write. The first step towards browser based editing came with the introduction of HTML forms and the TEXTAREA element in the early nineties. This later formed the basis for wikis, together with conventions for marking up headings, lists and links etc. The next step came with the introduction of XMLHTTP and designMode in Internet Explorer version 5. This allows you to use an IFRAME element for the document to be edited and to save the document back to the server via HTTP. The execCommand method allows developers to provide a user interface for basic editing operations.

Other browser vendors copied Internet Explorer, but did so in different ways [1] that make life hard for developers and have held back the full potential of browser based editing. This paper describes work on a cross browser open source library for use in editors. To avoid the interoperability problems in browsers, the library intercepts keystrokes and manages its own undo/redo history. There are a surprising lot of edge cases even for handling Enter, Backspace and Delete. As a result, a test framework has been developed to automate the application of a test suite, as a basis for incremental development and testing on a variety of browsers.

The aim is to encourage a proliferation of browser based editors for different purposes, e.g.:

- Basic rich text documents
- HTML based slide presentations such as Slidy [2]
- Online meeting management tools
- Next generation wikis and blogs
- Tools for reviewing and annotating documents

2. TEST SUITES

The framework needs to specify for each test:

- The action to be tested, which is either a key such as Enter, Backspace or Delete, or an action that would be triggered via a button or menu item, e.g. indent, outdent or insert heading.
- Secondary keys such as Shift, Control and Alt.
- The document before the action is applied.
- The editing cursor position before the action is applied.
- The document after the action has been applied.
- The editing cursor position after the action has been applied.

The test harness loads the test suite and applies each test, one by one, reporting any errors found. The test suite syntax was chosen for ease of maintenance. Tests are separated by blank lines. Each test starts with the name of the action preceded by any applicable secondary keys. The second line gives the document markup before the action is applied, and the third line gives the document markup expected after the action has been applied. Line breaks in the document markup are represented as \n. The editing cursor position is denoted by the vertical slash '|'. Here is a short example:

```
shift enter
<p>|</p>
<p><br>|<br></p>

shift enter
<p><em>ab|</em></p>
<p><em>ab<br>|<br></em></p>

shift enter
<p>ab|c</p>
<p>ab<br>|c</p>

backspace
<p>a|</p>
<p>|<br></p>
```


WWW 2014 Companion, April 7–14, 2014, Seoul, Korea.
ACM 978-1-4503-2038-2/13/05.

The test harness is a web page that uses XHR to load the test suite and parse it into an array of test objects using string split operations. Each test has to be applied three times: first for the edit action, second for the undo action and third for the redo action. Each test document is loaded into a test DIV element by setting the innerHTML property following string operations to restore the line breaks and to replace the vertical slash denoting the editing position by `<span id="caret"></span>`.

The span element is located using its id, and removed from the DOM, after having identified the target node in the DOM. If the caret is in the middle of a text node, it is necessary to merge the adjacent text nodes that were split as a result of the span element. The caret can now be set using a combination of document selection and range objects. One wrinkle is that you must not use range.deleteContents since on some browsers this has a side effect of modifying the DOM in ways that would mess up the test. A similar process is used to insert the vertical slash when preparing to match the document markup after the action with the expected markup as given in the test suite.

A further complication is dealing with older versions of Internet Explorer that don't support the W3C standard for document selections and ranges. This involves the insertion and removal of a short random substring as createRange doesn't work for text nodes.

Collapsed blocks and unrendered BR elements

Browsers collapse lines unless they contain a BR or visible content excluding whitespace characters. Browsers derived from KHTML (e.g. Safari and Chrome) take this a step further by preventing scripts from positioning the editing caret in a collapsed line. For example, attempting to position the caret in an empty P element results in the caret being placed before the P. A work around is to detect a difference in the requested and achieved caret position and to then insert a BR after the requested position and try again.

A better idea is to detect collapsed lines and insert a BR as needed. The BR is needed for empty block level elements, and when you want to insert the caret after a BR that is either at the end of its parent's content or is only followed by whitespace. The corrollary is that when backspacing over a BR, the BR at the end of the line should be removed if the resulting line contains visible content other than whitespace. The same applies when forward deleting a BR. Inserting required BR's should be done when loading a document, and after cut and paste operations. The other cases can be integrated into the code for Shift-Enter, Backspace and Delete.

3. UNDO/REDO

This is handled by a separate library module. The history is an array of transactions, where each transaction is a sequence of operations. Each operation is an object with undo and redo methods. The library includes a suite of operations on the document object model. These are invoked by the editing library's code for each edit action. The operations are in turn responsible for maintaining the undo/redo history. Examples include: move node, copy node, remove node, insert before, append child, merge text nodes, split text node, insert character, delete character, and set attribute. The undo library also provides methods for managing transactions, and for invoking undo and redo, and testing whether they are currently applicable.

4. DOCUMENT MANAGEMENT

The hypertext transfer protocol (HTTP [3]) provides a set of methods that can be used together with XHR for editing documents on the web server:

- GET – to get a document for editing
- PUT – to save a new or updated document
- DELETE – to remove a document from the server

The GET method should be used with the If-Modified-Since header to avoid loading a stale version of the document from a local cache. All three methods require server side configuration to manage access control, to avoid attempts to put excessively large documents, and to ensure that the saved documents are HTML and not something completely different, e.g. a binary file.

Web Distributed Authoring and Versioning (WebDAV [4]) provides an extension of HTTP for a broader set of file operations, e.g. making a copy of a file, moving a file from one location to another, managing directories, and lock and unlock operations on files.

When using HTTP GET and PUT there is a risk that someone else has modified the document whilst you were editing it, and a consequential risk of losing work. WebDav's lock and unlock mechanism is a partial solution by alerting you to the fact that someone else is already editing the document you want to edit. Another complementary solution is to use a document management solution to identify and merge changes. This needs to have some knowledge of document structure to avoid the risk of producing invalid markup. A document management solution can also offer the ability to review the sequence of document versions, and even to branch and merge versions as required.

5. LIVE EDITING

Live editing is where several people can be simultaneously viewing and editing the same document, and to see the changes others are making in pretty much real time. The most popular example of this approach is Google Docs [5]. Unfortunately, Google Docs requires you to save your documents on their servers and to limit yourself to the editing tools they provide. One example of an open source alternative is Etherpad [6]. Others are listed in wikipedia [7].

The author of this paper is seeking to support live editing in a clean cross browser open source library and would like to see this applied to the World Wide Web Consortium website, e.g. for taking minutes in teleconferences, and for collaborative editing of draft specifications. The aim is a modular library that can be applied for different kinds of editors.

The starting point is the core set of editing operations as mentioned above for the undo library module. These can be serialized as messages indicating proposed changes to a specific version of the document. My experiments used JSON messages over a Web Sockets connection, but an alternative is to use some form of HTTP polling, e.g. with long lived connections and streamed messages.

One approach is to get the server to review proposed changes and to merge them into sequence of accepted changes.

Unfortunately, it seems that server based real-time revision control is subject to a US patent. This prompted me to work on client-based revision control as a work around, that also has the happy benefit of minimizing the load on the server, and hence allowing for more people to use the server for collaborative editing at the same time.

The approach involves several browsers editing the document at the same time in an editing session. One of the browsers is elected as the senior editor and the others considered to be junior editors. The server maintains two message queues, one for proposed changes from junior editors, and the other for accepted changes from the senior editor. If the senior editor fails to respond in a reasonable time, the server arranges a new election to reassign the role.

The JSON format for changes gives the version number (as controlled by the senior editor). Each change is a sequence of operations, just as for the undo/redo history. Document nodes are identified using tumbler notation e.g. "1.4.2" where the successive numbers indicate the child node, with 1 for the first child, 2 for the second, and so forth.

The merging action takes the sequence of proposed changes since a specified version of the document and maps them to the changes that would be needed to the current version of the document. Some changes may not survive, e.g. when the senior editor deletes a parent node, and a junior editor updates one of its children. This is a case where having a live chat session integrated into the editor can be really beneficial by allowing the person whose work was just eradicated to ask the other editors to undo their changes.

Special treatment is given to changes to text nodes to allow for the situation where two or more people are editing the same paragraph at the same time. This is based upon matching substrings.

Junior editors need to be able to revert their document to the version given by the senior editor before applying the changes approved by the senior editor. At the same time, the changes the junior editor has made since then need to be transformed to apply to the new version of the document. A further complication is the need to revise the local undo/redo history so that users can undo/redo their local changes.

When a new election takes place, it does so with respect to the latest version of the document as approved by the former senior editor. Each editor needs to synchronize its uncommitted changes and its undo/redo history to match. The election should try to pick the most active editor.

6. MISSING EVENTS

For keystrokes, browsers widely support the keydown, keypress and keyup events. These were never formally standardized at W3C, and vary across browsers in respect to key codes and auto repeat. However, the situation is better with respect to keys such as Enter, Backspace and Delete, which are critical to cross browser editor interoperability.

There are events for cut, copy and paste, as well as for drag and drop operations. These are less widely supported, and as such require work arounds on older browsers. There are no events for undo/redo actions. This is only a problem if these actions are invoked using the browsers native UI. Providing prominent undo/redo buttons in the editor's UI will encourage users to invoke these actions in a way that the editor can deal with. Ideally, undo and redo events would be standardized, but it is likely to take time to convince everyone of the benefits.

7. EDITOR ARCHITECTURE

A common approach is to have a web page for the editor and to load the document to be edited into an IFRAME element in combination with the designMode property. One challenge is that documents may use scripts and style sheets that cause problems for the editing user interface, e.g. through use of CSS positioning and hidden elements. It may be better to impose an editing style sheet that overrides the document's own styling. This can be done by the parent page using a script to append a style element to the document's HEAD. The editor's user interface and styling can be chosen to match the kind of document to be edited. A major distinction is between structural editing and rich text editing. There are limits to the applicability of what you see is what you get (wysiwyg) editing, and sometimes it is worth using a different presentation for documents for the purposes of editing them.

8. EDITING PRACTICES

Editors need to be designed to work in an obvious fashion. This can break down if people apply changes to a style sheet and to individual paragraphs, as it can become unclear what is causing the appearence to be what it is. Similar difficulties are often found in word processors with numbering, especially for nested numbers for lists and sections. This is where simple approaches for rich text editing can come into conflict with the demands of structured documents.

Editing tools need to provide a rich range of document templates and styles, and to discourage users from making ad hoc changes that could later come back to bite them. A related idea is for the editing tool to provide context sensitive actions that make it easier for users to manage the desired document structure.

9. CONCLUSION

This paper reports work on a test framework for a cross browser open source library for browser based editors along with work on live editing support. The aim is to encourage the proliferation of a new generation of browser based editors for a wide range of different purposes. This will in turn encourage discussion on future standards for native browser support, and replacements for today's flawed execCommand method.

10. REFERENCES

[1] Mark Pilgrim's March 2009 blog on contentEditable. `http://blog.whatwg.org/the-road-to-html-5-contenteditable`.

[2] Slidy – an HTML based slide presentation solution. `http://www.w3.org/Talks/Tools/Slidy2`.

[3] Hypertext transfer protocol (HTTP 1.1). RFC2616, June 1999. `http://www.ietf.org/rfc/rfc2616.txt`.

[4] Web Distributed Authoring and Versioning (WebDAV). `http://en.wikipedia.org/wiki/WebDAV`.

[5] Google Docs. `http://docs.google.com/`.

[6] Etherpad. `http://en.wikipedia.org/wiki/Etherpad`.

[7] Collaborative Real-Time Editors. `http://en.wikipedia.org/wiki/Collaborative_real-time_editor`.