Title

author
`email@email.com`

June 5, 2014

The aim of the project was to conceive and implement a rich style editor as an addition to the Axel library. The editor had to provide editable areas embeddable into any HTML5 webpage[1], giving the user the ability to modify their content, to format them with at least the basic styles offered by most editors, and to paste content copied from other sources. The editor had to ensure cross-browser compatibility inasmuch as was possible[2]. An optional extension was to let the user customize the plugin's behaviour, allowing for user-defined styles and several input/output formats.

Axel (REFQQQ) is a Javascript-based, client-side library allowing to edit XML documents directly on a webpage. It provides a number of plugins to make various elements user-editable (text areas, selects, links, images, videos, etc.). It also makes it possible to load content defined as XML-templates and to save the same content, even after editing, in the same format. The purpose of the rich-content plugin is thus to enrich Axel's collection with a new kind of editable object.

1 Implementation

The code is organized around three main files: `richContent.js` that contains the bulk of the code, `richcontent.css`, that defines a number of CSS classes (those can be tweaked to change the appearance of the menu and the functions of the buttons), and `richcontentparams.js`, that specifies non-default parameters. Note that the latter file is not *stricto sensu* necessary ; it is only a convenience to modify the editor for specific purposes.

[1]The plugin is intended to work on HTML5-compliant browsers. Its main features ('contentEditable' attribute, draggable elements) would be meaningless otherwise. This is why we have not made a particular effort to test it on less advanced implementations.

[2]The ability for non-trivial Javascript code to work on different browsers is often a thorny issue, due to the lack of complete standardization among the various browsers. The code was mostly developed on Firefox, and tested on Microsoft's Internet Explorer, Safari, and Chrome. The main functions should work on all of them, although the specific effects might differ slightly depending on the browsers' implementation.

The editor relies on the possibility to designate some elements as 'contenteditable', a new feature introduced in HTML 5 (REFQQQ). Furnishing an element (such as a div or a span) with the attribute `contenteditable=true` allows a user to directly type in, delete or paste new content. Whenever the edition process starts, the plugin will thus first set the root of the element (the 'handle') as `contenteditable=true`. The attribute is set back to false when the editor is closed or another field gets the focus.

The main point of the rich-content plugin is however not so much to let some elements be editable, but to enable rich-style editing, i.e. to be able to impose various formats on (parts of) the content. The edition process is carried through thanks to a menu that pops up when the user clicks on the editable content, and thus makes it editable. The menu offers a number of buttons corresponding to the available styles. The default options we settled on are 'bold', 'italics', 'underlined' and 'striked', but, as we will see below, other formats can be defined by the user. A 'clear' button is also provided to rid the selected text of any style. A textarea and two further buttons let the user add and remove hyperlinks.

Styles are of course combinable as much as the browser-specific rendering makes it possible: a given piece of text can be made at the same time bold and underlined, striked and italics[3]. Selecting some text already formatted and formatting it again will however remove the corresponding style from the selection. Hitting the 'bold' button after selecting a bit of text in bold will thus clear it of its bold style (but any other style will be conserved).

Although standard HTML defines some formatting tags (`<strong>`, `<i>`, etc.) that could be used as markers of style, we decided to rely exclusively on CSS classes, for the sake of simplicity and regularity[4]. The use of CSS also allows for a better separation between structure and presentation, and makes the editor less browser-dependent.

The menu is endowed with the `draggable=true` attribute, also a feature of HTML5 (REFQQQ), which makes it possible to move it around on the window, at least in standard-compliant browsers. Note that the menu will keep its position even when it is hidden. It should reappear later a the position it occupied when it was last closed.

The editor expects the initial data to assume a strict structure, which will also be imposed on the saved content. Three main patterns are possible, depending on the encoding of the link nodes and the choice of the tags: HTML-like, Fragment-like (the default one) or semantic.

- In the HTML structure, the root node of the data (corresponding to "`<xt:use types="richContent" ... >` in the templates) is parent

[3]Some browsers however do not display the striked and underline styles together.

[4]We can thus compose the HTML content of the handle exclusively with a sequence of `<spans>` and `<a>` nodes. The handle structure remains akin to QQQQ and turns out to be easier to manipulate.

to a list of nodes of two kinds: standard nodes (typically `<span>`) and link nodes (typically `<a>` nodes), each of which contains only a text node. The links bear additionally a target attribute (typically '`href=<value>`'). A 'class' attribute is used to specify the style of any node.

- In the Fragment-type structure the data are composed of a list of `<Fragment>` nodes containing a single text node, similar to the HTML structure. The links however should be parents to two nodes, a `<LinkText>` node that will contain the visible text of the link, and a `<LinkRef>` node, whose inner text should be the target. The Fragment- and LinkText-nodes can optionally support a 'RichStyle' attribute that will be reinterpreted as a CSS class when the plugin creates the corresponding DOM elements. The visible rendering of a RichStyle attribute should hence be defined in some accessible CSS class of the same name.

- The semantic structure works with data made of a list of elements, the tags of which are defined by the user. Each of them will be used by the plugin to name the class attribute of the corresponding DOM elements[5]. Additionally two "neuter" tags are defined, `<Text>` for the standard unstyled text bits, and `<Link>` for the links. The latter should possess two children, a `<LinkRef>` to define the target, and a semantically tagged node to hold the text of the link (or a standard `<Text>` node, if no semantics is to be imposed on the text).

The default values for the tags and attributes in all three structures can be redefined in the `richcontentparams.js` file. The latter, normally stored in the `axel/bundles/richcontent/` folder, will override the default parameters applied while creating the editing menu, and while extracting and dumping the data. The relevant definitions are expected to appear, in JSON-like format, as a 'richContent' object inside an 'axelParams' object attached to the window:

```
window.axelParams['richContent'] = {
    formatsAndCSS: <array of button names and styles>,
    dataStructure: <structures of data>
}
```

In the default setting, the menu displays buttons for the default styles (bold, italic, underline, striked) as well a link making/unmaking area. A new set of buttons can be defined, with other names and other effects, in the `formatsAndCSS` array of `richcontentparams.js`. Each item in the

[5]Only the tags with a name matching an existing CSS class will be retained in the process.

array should be an object with two fields, a 'name', that will be displayed on the button, and a 'style' that designates the CSS class that will be applied on the selection when hitting the button. Note that the appearance of the buttons and the menu is defined in `axel/bundles/richcontent.css` and can be by redefining the appropriate classes.

```
    formatsAndCSS : [
        {name : 'Bold', style : 'bold'},
        {name : 'Italics', style : 'italics'},
        {name : 'Underline', style : 'underline'},
        {name : 'Strike', style : 'line-through'}
    ]
```

The 'dataStructure' field serves to override the names of the fields in some or all of the three structures allowed for the data (if some structure is not mentioned, the default setting will be used). This is especially useful to customize the tags of the data dumped in the saving process.

```
  dataStructure : {
    html : {
      link : {tag : 'a', ref : 'href', style : 'class'},
      standard : {tag : 'span', style : 'class'}
    },
    fragments : {
      link : {tag : 'Link',
              ref : {tag : 'LinkRef'},
              text : {tag : 'LinkText', style : 'RichStyle'}},
      standard : {tag : 'Fragment', style : 'RichStyle'}},
    semantic : {
      link : {tag : 'Link',
              ref : {tag : 'LinkRef'},
              text : {standard : 'LinkText'}},
      standard : {tag : 'Text'}}
    }
```

Any editable field defined on the page will become the current editable instance, responding to the menu, whenever the user clicks on its span elements. Clicking on the links however will trigger a pop-up window[6] to let the user choose to either open the target URL, or start the edition process. If several editable instances are present on the page, only the instance currently focused on will be registered with the menu. Clicking on another instance will just reregister the menu on the latter.

[6]We reuse the device in `popup.js`.

The editor is configured to permit only CSS classes defined in the array `formatsAndCSS` to occur inside the handle, either in the default version hard-coded inside `richContent.js`, or in the user-defined version.

The RichStyle attributes supported by the `<Fragment>` items and the tags defining the nodes in the semantic structure will become the class attributes born by the corresponding `<span>` and `<a>` children of the handle. As is customary for CSS classes, when several of them are present, they will appear as a sequence of strings separated by spaces. The corresponding semantic tags and RichStyle attributes will however handle them as strings separated by '_'. Care should thus be taken to avoid underscores when choosing class identifiers.

The default version of the editor creates the handle as a `<span>`. Since only `<span>`- and `<a>`- nodes will appear as its children, whatever newline character present in their text content will be ignored. It is possible to force the editor to display line breaks and spaces either by requiring the editor to create a `<pre>` node for the handle, `<xt:use ... handle="pre">`', or by specifying the "multilines" parameter as `<xt:use ... param="multilines=normal">`'[7]. In the first case, any line break already present in the data will appear as such, but the user will not be allowed to add further ones. In the second case, the user will be able to introduce carriage returns by hitting Shift+Enter[8].

2 Technical notes

The main editing operations are achieved by the functions

- `makeLink(linkArea, link)`
- `clearRange()`
- `enrich(style)`
- `interceptPaste(node, event)`

that rely on the auxiliary functions

- `recreateTree(root, mainTag, allTagged, inherit, style, link, clear)`
- `cleanTree(target, current)`

[7]Note that in general we have taken care to make the parameters case-insensitive, so that the browser should respond similarly to `"multilines=normal"` and `"multilines=Normal"`, etc.

[8]This is done through a call to the function `this.keyboard.enableRC()`.

The first three follow a similar structure. They first obtain a range object (REFQQQ) covering the current selection inside the editable field; they extract its content, and inject it into a new node, created on the spot. The node is then inserted onto the page instead of the selection. This has the effect of disturbing the relatively flat structure of the handle (the handle is a 3-level tree with `<span>`- and `<a>`-children nodes, each with a text node as only child). The `recreateTree()` function is then called to reshuffle the handle, while applying any transformation necessary to achieve the desired effect. This involves adding or removing classes, creating elements with different tag names, adding the 'href' attributes where needed. Finally, the `cleanTree()` function is called to suppress redundant or empty nodes created in the process[9].

`interceptPaste()` works similarly except that it only deals with simple unformatted text. Avoiding the paste of HTML-formatted text is an intentional decision, made as much for the sake of simplicity, and because it would be difficult to translate consistently text extracted from random pages into the formats allowed by the editor. One advantage of the decision is to make it unnecessary to call `recreateTree()` after the pasted text is injected, as we only need to make sure the parent to the newly inserted node (that contains the pasted text) contains a single text node. This is done by a call to the `innerText(node)` function, that extracts the text content from its argument, discarding any formatting at the same time.

`interceptPaste()` has to cancel the default effect of the paste operation. Unfortunately, browsers tend to be inconsistent here, with some responding to `event.preventDefault()` (Firefox, for instance) while others (at least some versions of IE) will work properly only with the instruction `event.returnValue = false;`. Recovering the content of the clipboard also requires different constructs (such as calling `window.clipboardData.getData(‘‘Text")` for Microsoft IE, and `event.clipboardData.getData(‘‘Text")` for other browsers)[10].

The content of the data is extracted thanks to the three functions

- `extractHTMLContentXT()`
- `extractFragmentContentXT()`
- `extractSemanticContentXT()`

[9]The extraction of the range object and its reinjection create a number of empty nodes, while the edition and reedition of the same bits of text easily create sequences of nodes with identical formats. It is thus useful to simplify the content by removing the first and merging the second.

[10]The resulting output should be a cleaned, tag-free, version of the copied content; unfortunately, no matter how much care is taken to account for every situation, some residual — and hardly predictable — corner cases seem to be handled improperly by some implementations.

one for each of the possible structures. Despite their differences, they all work similarly to create a DOM tree suitable to fill the editable area. Note that we had to overload `xtdom.extractDefaultContentXT()` to let the editor work with formatted text, instead of the simple text released by the original function. The reverse operation — creating an XML tree from the data before saving it — is achieved by the three corresponding functions `logToHTML()`, `logToSemantic()`, `logToFragments()`.

3 Conclusion