

A WYSIWYG Framework

by

Johannes-Lukas Bombach

Submitted to the Fachbereich Informatik, Kommunikation und
Wirtschaft

in partial fulfillment of the requirements for the degree of

Master of Science

at the

HOCHSCHULE FÜR TECHNIK UND WIRTSCHAFT BERLIN

August 2015

Author
Johannes-Lukas Bombach
August 26, 2015

Certified by
Prof. Dr. Debora Weber-Wulff
Associate Professor
Thesis Supervisor

A WYSIWYG Framework

by

Johannes-Lukas Bombach

Submitted to the Fachbereich Informatik, Kommunikation und Wirtschaft
on August 26, 2015, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

Browsers do not offer native elements that allow for rich-text editing. There are third-party libraries that emulate these elements by utilizing the `contenteditable`-attribute. However, the API enabled by `contenteditable` is limited and unstable. Bugs and unwanted behavior can only be worked around and not fixed. The library "Type" demonstrates that rich-text editing can be achieved without requiring the `contenteditable` attribute, thus solving many problems contemporary third-party rich-text editor libraries have.

Thesis Supervisor: Prof. Dr. Debora Weber-Wulff
Title: Associate Professor

Acknowledgments

I would like to extend my thanks to my supervisor Prof. Dr. Debora Weber-Wulff for giving me the opportunity to work on a topic I have been passionate about for years.

I would like to thank Marijn Haverbeke for his work on CodeMirror, from which I could learn a lot.

I would like to thank my father for supporting me. Always.

Contents

1	Introduction	13
1.1	Motivation	13
1.2	Terminology	15
1.3	Structure	15
2	Text editing in desktop environments	17
2.1	Basics of plain-text editing	17
2.2	Basics of rich-text editing	17
2.3	Libraries for desktop environments	17
3	Text editing in browser environments	19
3.1	Plain-text inputs	19
3.2	Rich-text editing	19
3.2.1	HTML Editing APIs	20
3.2.2	Usage of HTML Editing APIs	23
3.2.3	Advantages of HTML Editing APIs	23
3.2.4	Disadvantages of HTML Editing APIs	23
3.2.5	DOM manipulation without Editing APIs	23
3.2.6	Rich-text editing without Editing APIs	24
4	Implementation	25
4.1	JavaScript library development	25
4.2	Ich habe verwendet	25

4.3	Coding conventions	26
4.4	Coding Klassen	26
4.5	Programmstruktur	26
4.6	Type	27
4.7	Caret	27
4.8	Range	27
4.9	Selection	27
4.10	Selection Overlay	27
4.11	Input	27
4.12	Formatting	27
4.13	Change Listener	28
4.14	Contents	28
4.15	Development	28
4.16	Dom Utilities	28
4.17	Dom Walker	28
4.18	Environment	28
4.19	Core Api	28
4.20	Event Api	28
4.21	Events	29
4.22	Input Pipeline	29
4.23	Plugin Api	29
4.24	Settings	29
4.25	Text Walker	29
4.26	Utilities	30
A	Tables	31
B	Figures	33

List of Figures

List of Tables

2.1	Rich-text components in desktop environments	18
3.1	Editing API attributes	20
A.1	HTML Editing API	31

Chapter 1

Introduction

1.1 Motivation

Rich-text editors are commonly used by many on a daily basis. Often, this happens knowingly, for instance in an office suite, when users wilfully format text. But often, rich-text editors are being used without notice. For instance when writing e-mails, entering a URL inserts a link automatically in many popular e-mail-applications. Also, many applications, like note-taking apps, offer rich-text capabilities that go unnoticed. Many users do not know the difference between rich-text and plain-text writing. Rich-text editing has become a de-facto standard, that to many users is *just there*. Even many developers do not realise that formatting text is a feature that needs special implementation, much more complex than plain-text editing.

While there are APIs for creating rich-text input controls in many desktop programming environments, web-browsers do not offer native rich-text inputs. However, third-party JavaScript libraries fill the gap and enable developers to include rich-text editors in web-based projects.

The libraries available still have downsides. Most importantly, only a few of them work. As a web-developer, the best choices are either to use CKEditor or TinyMCE. Most other editors are prone to bugs and unwanted behaviour. Piotrek Koszuliński, core developer of CKEditor comments this on StackOverflow as follows:

*Don't write wysiwyg editor[sic] - use one that exists. It's going to consume all your time and still your editor will be buggy. We and guys from other... two main editors (guess why only three exist) are working on this for years and we still have full bugs lists ;).*¹

A lot of the bugs CKEditor and other editors are facing are due to the fact that they rely on so-called "HTML Editing APIs" that have been implemented in browsers for years, but only been standardized with HTML5. Still, to this present day, the implementations are prone to numerous bugs and behave inconsistently across different browsers. And even though these APIs are the de-facto standard for implementing rich-text editing, with their introduction in Internet Explorer 5.5, it has never been stated they have been created to be used as such.

It's a fact, that especially on older browsers, rich-text editors have to cope with bugs and inconsistencies, that can only be worked around, but not fixed, as they are native to the browser. On the upside, these APIs offer a high-level API to call so-called "commands" to format the current text-selection.

However, calling commands will only manipulate the document's DOM tree, in order to format the text. This can also be achieved without using editing APIs, effectively avoiding unfixable bugs and enabling a consistent behaviour across all browsers.

Furthermore CKEditor, TinyMCE and most other libraries are shipped as user interface components. While being customizable, they tend to be invasive to web-projects.

This thesis demonstrates a way to enable rich-text editing in the browser without requiring HTML Editing APIs, provided as a GUI-less software library. This enables web-developers to implement rich-text editors specific to the requirements of their web-projects.

¹<http://stackoverflow.com/questions/11240602/paste-as-plain-text-contenteditable-div-textarea-word-excel/1129008211290082>, last checked on 07/13/2015

1.2 Terminology

rich-text, WYSIWYG, word-processing, WYSIWYM

1.3 Structure

The first part of this thesis explains rich-text editing on desktop PCs. The second part explains how rich-text editors are currently being implemented in a browser-environment and the major technical differences to the desktop. Part three will cover the downsides and the problems that arise with the current techniques used. Part four will explain how rich-text editing can be implemented on the web bypassing these problems. Part five dives into the possibilities of web-based rich-text editing in particular when using the techniques explained in this thesis.

Chapter 2

Text editing in desktop environments

2.1 Basics of plain-text editing

caret selection input

2.2 Basics of rich-text editing

document tree formatting algorithms

2.3 Libraries for desktop environments

It is no longer needed to implement basic rich-text editing components from the ground up. Rich-text editing has become a standard and most modern Frameworks, system APIs or GUI libraries come with built-in capabilities. Table 2.1 lists rich-text text components for popular languages and frameworks.

Environment	Component
Java (Swing)	JTextPane / JEditorPane
MFC	CRichEditCtrl
Windows Forms / .NET	RichTextBox
Cocoa	NSTextView
Python	Tkinter Text
Qt	QTextDocument

Table 2.1: Rich-text components in desktop environments

Chapter 3

Text editing in browser environments

3.1 Plain-text inputs

Text input components for browsers have been introduced with the specification of HTML 2.0¹. The components proposed include inputs for single line (written as `<input type="text" />`) and multiline texts (written as `<textarea></textarea>`). These inputs allow writing plain-text only.

3.2 Rich-text editing

Major browsers, i.e. any browser with a market share above 0.5%², do not offer native input fields that allow rich-text editing. Neither the W3C's HTML5 and HTML5.1 specifications nor the WHATWG HTML specification recommend such elements. However, by being able to display HTML, browsers effectively are rich-text viewers. By the early 2000s, the first JavaScript libraries emerged, that allowed users to interactively change (parts of) the HTML of a website, to make rich-text editing in the browser possible. The techniques used will be discussed in section 3.2.1 to section 3.2.2.

¹<https://tools.ietf.org/html/rfc1866>, last checked on 07/15/2015

²<http://gs.statcounter.com/all-browser-ww-monthly-201406-201506-bar>, last checked on 07/15/2015

Attribute	Type	Can be set to	Possible values
<code>designMode</code>	IDL attribute	Document	"on", "off"
<code>contentEditable</code>	IDL attribute	Specific <code>HTMLElements</code>	boolean, "true", "false", "inherit"
<code>contenteditable</code>	content attribute	Specific <code>HTMLElements</code>	empty string, "true", "false"

Table 3.1: Editing API attributes

3.2.1 HTML Editing APIs

In July 2000, with the release of Internet Explorer 5.5, Microsoft introduced the IDL attributes `contentEditable` and `designMode` along with the content attribute `contenteditable`³⁴. These attributes were not part of the W3C’s HTML 4.01 specifications⁵ or the ISO/IEC 15445:2000⁶, the defining standards of that time. Table 3.1 lists these attributes and possible values.

By setting `contenteditable` or `contentEditable` to "true" or `designMode` to "on", Internet Explorer 5.5 switches the affected elements and their children to an editing mode. The `designMode` can only be applied to the entire document and the `contentEditable` and `contenteditable` attributes can be applied to specific HTML elements as described on Microsoft’s Developer Network (MSDN) online documentation⁷. These elements include "divs", "paragraphs" and the document’s "body" element. In editing mode

1. Users can interactively click on and type inside texts
2. An API is enabled that can be accessed via JScript and JavaScript

In editing mode, setting the caret by clicking on elements, accepting keyboard input and modifying text nodes is handled entirely by the browser. No further scripting is necessary.

The API enabled can be called globally on the `document` object, but will only execute when the user’s selection or caret is focussed inside an element in editing

³[https://msdn.microsoft.com/en-us/library/ms533720\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/ms533720(v=vs.85).aspx), last checked on 07/10/2015

⁴[https://msdn.microsoft.com/en-us/library/ms537837\(VS.85\).aspx](https://msdn.microsoft.com/en-us/library/ms537837(VS.85).aspx), last checked on 07/10/2015

⁵<http://www.w3.org/TR/html401/>, last checked on 07/14/2015

⁶http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=27688, last checked on 07/14/2015

⁷[https://msdn.microsoft.com/en-us/library/ms537837\(VS.85\).aspx](https://msdn.microsoft.com/en-us/library/ms537837(VS.85).aspx), last checked on 07/10/2015

mode. Table A.1 lists the full HTML editing API. To format text, the method `document.execCommand` can be used. Calling

```
1 document.execCommand('italic', false, null);
```

Listing 3.1: Emphasizing text using the HTML editing API

will wrap the currently selected text inside an element in editing mode with `<i>` tags. The method accepts three parameters. The first parameter is the "Command Identifier" to determine which command to execute. For instance, this can be `italic` to italicize the current selection or `createLink` to create a link with the currently selected text as label.

```
1 document.execCommand('createLink', false, 'http://google.de/');
```

Listing 3.2: Creating a link using the HTML editing API

The *third* parameter will be passed on to the internal command given as first parameter. In the case of a `createLink` command, the third parameter is the URL to be used for the created link. The *second* parameter determines if executing a command should display a user interface specific to the command. If the `createLink` command with this set to `true` and without a third parameter, the user will be prompted to enter a URL.

With the release of Internet Explorer 5.5 and the introduction of editing capabilities, Microsoft released a sparse documentation⁸ describing only the availability and the before-mentioned element restrictions of these attributes.

According to Mark Pilgrim, author of the "Dive into" book series and contributor to the the Web Hypertext Application Technology Working Group (WHATWG), Microsoft did not state a specific purpose for its editing API, but, its first use-case has been rich-text editing⁹.

In March 2003, the Mozilla Foundation introduced an implementation of Microsoft's `designMode`, named `Midas`, for their release of Mozilla 1.3. Mozilla names

⁸[https://msdn.microsoft.com/en-us/library/ms537837\(VS.85\).aspx](https://msdn.microsoft.com/en-us/library/ms537837(VS.85).aspx), last checked on 07/10/2015

⁹<https://blog.whatwg.org/the-road-to-html-5-contenteditable>, last checked on 07/10/2015

this "rich-text editing support" on the Mozilla Developer Network (MDN)¹⁰. In June 2008, Mozilla added support for contentEditable IDL and contenteditable content attributes with Firefox 3.

Mozilla's editing API resembles the API implemented for Internet Explorer, however, there are still differences (compare ¹¹¹²). Most notably, Microsoft and Mozilla differ in the commands provided to pass to document.execCommand¹³¹⁴ and the markup generated by invoking commands¹⁵. In fact, Mozilla only provides commands dedicated to text editing while Microsoft offers a way to access lower-level browser components (like the browser's cache) using execCommand. This may show, that even though rich-text editing was its first use case and Mozilla implemented it naming it that, this editing API was not originally intended to be used as such.

In March 2008, Apple released Safari 3.1¹⁶ including full support for contentEditable and designMode¹⁷, followed by Opera Software in June 2006¹⁸ providing full support in Opera 9¹⁹. MDN lists full support in Google Chrome since version 4²⁰, released in January 2010²¹.

Around the year 2003[*MeineTabelle*] the first JavaScript libraries emerged that made use of Microsoft's and Mozilla's editing mode to offer rich-text editing in the browser. Usually these libraries were released as user interface components (text fields) with inherent rich-text functionality and were only partly customizable.

¹⁰https://developer.mozilla.org/en/docs/Rich-Text_Editing_in_Mozilla, last checked on 07/10/2015

¹¹[https://msdn.microsoft.com/en-us/library/hh772123\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/hh772123(v=vs.85).aspx), last checked on 07/10/2015

¹²<https://developer.mozilla.org/en-US/docs/Midas>, last checked on 07/10/2015

¹³<https://developer.mozilla.org/en-US/docs/Midas>, last checked on 07/10/2015

¹⁴[https://msdn.microsoft.com/en-us/library/ms533049\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/ms533049(v=vs.85).aspx), last checked on 07/10/2015

¹⁵https://developer.mozilla.org/en/docs/Rich-Text_Editing_in_MozillaInternet_Explorer_Differences, last checked on 07/10/2015

¹⁶<https://www.apple.com/pr/library/2008/03/18Apple-Releases-Safari-3-1.html>, last checked on 07/10/2015

¹⁷<http://caniuse.com/feat=contenteditable>, last checked on 07/10/2015

¹⁸<http://www.opera.com/docs/changelogs/windows/>, last checked on 07/10/2015

¹⁹<http://www.opera.com/docs/changelogs/windows/900/>, last checked on 07/10/2015

²⁰https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Content_Editable, last checked on 07/10/2015

²¹http://googlechromereleases.blogspot.de/2010/01/stable-channel-update_25.html, last checked on 07/10/2015

In May 2003 and March 2004 versions 1.0 of "FCKEditor" and "TinyMCE" have been released as open source projects. These projects are still being maintained and remain among the most popular choices for incorporating rich-text editing in web-based projects. // *Technik, wie diese Editoren funktionieren erklären.*

Seeing the history of editing APIs, it is understandable how this has become the standard for rich-text editing. However, with its introduction in Internet Explorer 5.5, it has not been stated that the `designMode` and `contentEditable` attributes have been intended to enable rich-text editing. Sections X and Y will discuss the advantages and disadvantages of these APIs.

3.2.2 Usage of HTML Editing APIs

How Js Libraries work. Maybe how only a few work. StackOverflow quote, buglists, Medium post, other stuff to find.

3.2.3 Advantages of HTML Editing APIs

Higlevel API Wenig Aufwand discussion of WHATWG

3.2.4 Disadvantages of HTML Editing APIs

bugs very limited api. limited wo a few commands and a few paramters No specifications on what markup to generate (mozilla != ie, mdn has links for that)

3.2.5 DOM manipulation without Editing APIs

In October 1998 the World Wide Web Consortium (W3C) published the "Document Object Model (DOM) Level 1 Specification". This specification includes an API on how to alter DOM nodes and the document's tree²². It provided a standardized way for changing a website's contents. With the implementations of Netscape's JavaScript and Microsoft's JScript this API has been made accessible to web developers.

²²<http://www.w3.org/TR/REC-DOM-Level-1/level-one-core.html>, last checked on 07/10/2015

3.2.6 Rich-text editing without Editing APIs

Chapter 4

Implementation

4.1 JavaScript library development

No IDEs, tools, not even conventions.

Not for building: Big JS libraries all do it differently. Top 3 client side JavaScript repositories (stars) on github <https://github.com/search?l=JavaScript&q=stars%3A%3E1&s=stars&ty>
Angular.js: Grunt d3 Makefile, also ein custom build script welche node packages aufruft jQuery custom scripts, mit grunt und regex und so

Not for Architecture Angular custom module system with own conventions d3 mit nested objects (assoc arrays) und funktionen jQuery mit .fn ACE mit Klassen, daraus habe ich gelernt

4.2 Ich habe verwendet

Gulp requireJs AMDClean Uglify

JSLint - Douglas Crockford coding dogmatas / conventions JSCS - JavaScript style guide checker

Livereload PhantomJs Mocha Chai

Durch Require und AMDClean schÃ¶ne arbeitsweise (am ende Ãĳber bord geworfen) und kleine DateigrÃ¶Ãe, wenig overhead.

Automatisierte Client side Tests mit PhantomJs und Mocha/Chai

4.3 Coding conventions

Habe mich größtenteils an Crockfordstyle orientiert, aber die Klassen anders geschrieben. Habe den Stil von ACE editor verwendet, denn der ist gut lesbar. Lesbarkeit war mir wichtiger als Crockford style. Für private Eigenschaften und Methoden habe ich die prefix convention verwendet. https://developer.mozilla.org/en-US/Add-ons/SDK/Guides/Contributor_s_Guide/Private_Properties Sie bewirkt keine echte accessibility restriction, aber es ist eine allgemein anerkannten convention und ist auch viel besser lesbar.

4.4 Coding Klassen

Ich habe mich für Klassen entschieden. Das hat folgende Vorteile:

- * Klassen sind ein bewährtes Konzept um Code zu kapseln, logisch zu strukturieren und lesbar zu verwalten
- * Durch prototypische Vererbung existiert die funktionalität von Klassen nur 1x im Browser 7 RAM
- * Zudem gibt es Instanzvariablen, die für jede Type Instanz extra existieren und so mehrere Instanzen erlauben
- * Die instanzvariablen sind meistens nur Pointer auf Instanzen anderer Klassen
- * Das ganze ist dadurch sehr schlank

4.5 Programmstruktur

Es gibt ein Basisobjekt, das ist die Type "Klasse". Darin werden dann die anderen Klassen geschrieben "Type.Caret", "Type.Selection", "Type.Range", ... Das hat den Vorteil dass das ganze ge-name-spaced ist, so dass ich keine Konflikte mit Systemnamen habe (Range) (und auch nicht mit anderen Bibliotheken) Effektiv gibt es eine (flache) Baumstruktur und so mit Ordnung. Für bestimmte Klassen, "Type.Event.Input", "Type.Input.Filter.X" geht es tiefer. Der zweite Grund ist, dass ich somit alle Klassen die ich geschrieben habe für Entwickler sichtbar bereit stelle und nicht implizit und versteckt für irgend nen Quatsch.

Ursprünglich ein MVC konzept geplant mit einem Document Model und verschiedenen Renderern, aber aber den haufen geworfen.

Ich werde jetzt die einzelnen Module erstellen

4.6 Type

Die Type

4.7 Caret

caret

4.8 Range

range

4.9 Selection

selection

4.10 Selection Overlay

overlay

4.11 Input

input

4.12 Formatting

formatting

4.13 Change Listener

change

4.14 Contents

contents

4.15 Development

developmment

4.16 Dom Utilities

dom util

4.17 Dom Walker

dom walker

4.18 Environment

env

4.19 Core Api

core api

4.20 Event Api

ev api

4.21 Events

Input input event only event required so far

4.22 Input Pipeline

pipeline ideas

Caret caret

Command command

Headlines head lines

Line Breaks line breaks

Lists lists

Remove remove

Spaces spaces

4.23 Plugin Api

plugin api

4.24 Settings

settings

4.25 Text Walker

text walker

4.26 Utilities

util

Appendix A

Tables

Method	Description
execCommand	Executes a command.
queryCommandEnabled	Returns whether or not a given command can currently be executed.
queryCommandIndeterm	Returns whether or not a given command is in the indeterminate state.
queryCommandState	Returns the current state of a given command.
queryCommandSupported	Returns whether or not a given command is supported by the current document's range.
queryCommandValue	Returns the value for the given command.

Table A.1: HTML Editing API

Appendix B

Figures

Bibliography