# 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

# List of Figures

# List of Tables

# 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 capabilites 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 noumerous bugs and behave inconsitently 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 consistend 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/11290082112900082, last checked on 07/13/2015

## 1.2 Terminilogy

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 capabilites. 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 2.0[1]. 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 enable rich-text editing in the browser. The techniques used will be discussed in section 3.2.1 through section 3.2.4.

[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 |
|---|---|---|---|
| designMode | IDL attribute | Document | "on", "off" |
| contentEditable | IDL attribute | Specific HTMLElements | boolean, "true", "false", "inherit" |
| contenteditable | content attribute | Specific HTMLElements | 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`[3][4]. These attributes were not part of the W3C's HTML 4.01 specifications[5] or the ISO/IEC 15445:2000[6], the defining standards of that time. Table 3.1 lists these attributes and possible values.

```
1 <div contenteditable="true">
2   This text can be edited by the user.
3 </div>
```

Listing 3.1: An element set to editing mode

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 documentation[7]. These elements include "divs", "paragraphs" and the docuement's "body" element amongst others. 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

[3]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, last checked on 07/10/2015
[5]http://www.w3.org/TR/html401/, last checked on 07/14/2015
[6]http://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, last checked on 07/10/2015

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 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 3.2: 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 paramter is the "Command Identifier", that determines 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 3.3: 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 link to create. The *second* paramter determines if executing a command should display a user interface specific to the command. For instance, using the `createLink` command with the second parameter set to `true` and not passing a third parameter, the user will be prompted with a system dialog to enter a URL. A full list of possible command identifiers can be found on MSDN[8].

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

### 3.2.2 Development and standardization of HTML Editing APIs

With the release of Internet Explorer 5.5 and the introduction of editing capabilities, Microsoft released a short documentation[9], containing the attributes' possible values and element restrictions along with two code examples. Although a clear purpose has not been stated, the code examples demonstrated how to implement rich-text input fields with it. Mark Pilgrim, author of the "Dive into" book series and contributor to the the Web Hypertext Application Technology Working Group (WHATWG), also states that the API's first use case has been for rich-text editing[10].

In March 2003, the Mozilla Foundation introduced an implementation of Microsoft's designMode, named Midas, for their release of Mozilla 1.3. Mozilla already named this "rich-text editing support" on the Mozilla Developer Network (MDN)[11]. In June 2008, Mozilla added support for contentEditable IDL and contenteditable content attributes with Firefox 3.

Mozilla's editing API mostly resembles the API implemented for Internet Explorer, however, to this present day, there are still differences (compare[12][13]). This includes the available command identifiers [14][15] as well as the markup generated by invoking commands[16].

In March 2008, Apple released Safari 3.1[17] including full support for contentEditable and designMode[18], followed by Opera Software in June 2006[19] providing full

[9]https://msdn.microsoft.com/en-us/library/ms537837(VS.85).aspx, last checked on 07/10/2015
[10]https://blog.whatwg.org/the-road-to-html-5-contenteditable, last checked on 07/10/2015
[11]https://developer.mozilla.org/en/docs/Rich-Text_Editing_in_Mozilla, last checked on 07/10/2015
[12]https://msdn.microsoft.com/en-us/library/hh772123(v=vs.85).aspx, last checked on 07/10/2015
[13]https://developer.mozilla.org/en-US/docs/Midas, last checked on 07/10/2015
[14]https://developer.mozilla.org/en-US/docs/Midas, last checked on 07/10/2015
[15]https://msdn.microsoft.com/en-us/library/ms533049(v=vs.85).aspx, last checked on 07/10/2015
[16]https://developer.mozilla.org/en/docs/Rich-Text_Editing_in_MozillaInternet_Explorer_Differences, last checked on 07/10/2015
[17]https://www.apple.com/pr/library/2008/03/18Apple-Releases-Safari-3-1.html, last checked on 07/10/2015
[18]http://caniuse.com/feat=contenteditable, last checked on 07/10/2015
[19]http://www.opera.com/docs/changelogs/windows/, last checked on 07/10/2015

support in Opera 9[20]. MDN lists full support in Google Chrome since version 4[21], released in January 2010[22].

Starting in November 2004, WHATWG members have started actively discussing to incorporate these editing APIs in the HTML5 standard. Through reverse engineering, the WHATWG developed a specification based on Microsoft's implementation[23] and finally decided to include it in HTML5. With W3C's coorporation and the split in 2011, similar editing APIs based on this work are now included in W3C's HTML5 Standard[24] and WHATWG's HTML Standard[25].

### 3.2.3 Emergence of HTML editing JavaScript libraries

Around the year 2003[26] 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.

In May 2003 and March 2004 versions 1.0 of "FCKEditor"[27] and "TinyMCE" have been released as open source projects. These projects are still being maintained and remain among the most used rich-text editors. TinyMCE is the default editor for Wordpress and CKEditor is listed as the most popular rich-text editor for Drupal[28].

Since the introduction of Microsoft's HTML editing APIs, a large number of rich-text editors have been implemented. While many have been abandoned, GitHub lists about 600 JavaScript projects related to rich-text editing[29]. However, it should be noted, that some projects only use other projects' editors and some projects are stubs.

[20] http://www.opera.com/docs/changelogs/windows/900/, last checked on 07/10/2015

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

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

[23] https://blog.whatwg.org/the-road-to-html-5-contenteditable, last checked on 07/15/2015

[24] http://www.w3.org/TR/html5/editing.html, last checked on 07/15/2015

[25] https://html.spec.whatwg.org/multipage/interaction.htmlediting-2, last checked on 07/15/2015

[26] compare *Meine Tabelle aller Editoren*

[27] Now distributed as "CKEditor"

[28] https://www.drupal.org/project/project_module, last checked on 07/16/2015

[29] https://github.com/search?o=desc&q=wysiwyg&s=stars&type=Repositories&utf8=%E2%9C%93, last checked on 07/16/2015

Popular choices on GitHub include ”MediumEditor”, ”wysihtml”, ”Summernote” and others.

### 3.2.4 Usage of HTML Editing APIs

Most rich-text editors use HTLM editing APIs as their basis. CKEditor and TinyMCE dynamically create an `iframe` on instantiation and set its `body` to editing mode using `contenteditable`. Doing so, users can type inside the `iframe` so it effectively acts as text input field. Both libraries wrap the `iframe` in a user interface with buttons to format the `iframe`'s contents. When a user clicks on a button in the interface `document.execCommand` will be called on the `iframe`'s `document` and the selected text will be formatted. While using an `iframe` is still in practice, many newer editors use a `div` instead. The advantages and disadvatanges of this technique will be discussed in Sections XY.

It makes sense to use HTML editing APIs for rich-text editing. Microsoft's demos published with the first implementation of these APIs suggest to do so. Mozilla picked up on it and called their implementation ”rich-text editing API”. Other browsers followed with APIs based on Microsoft's idea of editable elements. However, it would have been imaginable to implement an actual user interface component, a dedicated rich-text input field. The HTML editing APIs have only been standardized with HTML5, which itself introduces 13 new types of input fields[30], but none with rich-text capabilites.

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

Seeing how editing APIs have come to existance, as an undocumented API by microsoft, taken over by mozilla, bossted by JS libraries and then adopted by other browsers, it can be understood how editing APIs came to be. However, by its introduction by microsoft, it has not been stated that this has been its original purpose. And even if Mozilla picked up on it, it is not clear that this API is in fact the best

[30] https://developer.mozilla.org/en/docs/Web/HTML/Element/Input, last checked on 07/16/2015

way to implement rich-text editing. WHATWG has discussed this API, arguing if it is the best way to provide rich-text functionality.

** WICHTIG DEN BOGEN ZU SPANNEN WARUM ICH ÃIJBER DIE GESCHICHTE SCHREIBE** ERKLÃĎREN, DASS ALLE DIE API VON MICROSOFT ÃIJBERNOMMEN HABEN; ANSTATT RICH-TEXT ELEMENTE EINZUFÃIJHREN. HTML5 HAT KOMPONENTEN FÃIJR TIME UND DATE; WARUM NICHT RICH TEXT- WHATWG HAT DAS DISCUSSED. WAS WAREN DIE GRÃIJNDE?. BEI DER BESCHREIBUNG DARAUF ACHTEN DASS ICH DEN GRUND GEBE; WARUM ICH DIE ENTWICKLUNG SO LANGE ERKLÃĎRE. VIELLEICHT AUCH SAGEN, DASS TROTZ DASS HTML5 ERST SEIT X RELEASED WURDE, ANHAND DER GESCHICHTE GESEHEN WERDEN KANN; DASS ES SCHON LANGE BROWSER SUPPORT GIBT.

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` end `contentEditable` attributes have been intended to enable rich-text editing. Sections X and Y will discuss the advantages and disadvantages of these APIs.

### 3.2.5 Advantages of HTML Editing APIs

Higlevel API Wenig Aufwand discussion of WHATWG

### 3.2.6 Disadvantages of HTML Editing APIs

bugs it seems to me the api as provided is a bad idea. Programmers can handle control, and why not make this control specific instead of executing bulkish commands. 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)

**pasting** is a problem: shitty markup can be pasted, paste event not implemented in all browser, some offer a modal, some clean up after change. I can solve this problem.

### 3.2.7 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 tree[31]. 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.

### 3.2.8 Advantages of rich-text editing without Editing APIs

Only this can guarantee the same behaviour and the same output across all browsers. It puts the contenteditable implementation in the hand of JavaScript developers. We no longer have to wait for browsers to fix issues *and conform each other* and thus can be faster, at least possibly, than browsers are. Also we can deploy updates immediately to all users and do not have to worry they use old browsers even if updates exist.

### 3.2.9 Disadvantages of rich-text editing without Editing APIs

Apparently implementing editing is prone to a lot of bugs issues **and edge-casese**. It can be assumed that it is difficult to do so. / Discuss the issues discussed by WHATWG here. It's difficult to do this.

Also this can be seen as "yet another" implementation of contenteditable, equal to browser implementations. Just one more editor for developers to take care of. However, this isn't quite correct, cos developers usually can choose a single editor that is used for the entire project, so they only have to take care of a single editor.

### 3.2.10 Disadvantages of offering user interface components

Warum es besser ist eine library zu shippen und nicht einen editor als ui component

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

# Chapter 4

# Implementation

## 4.1 Conformity with HTML Editing APIs

Wie sehr passt meine library zu dein HTML Editing APIs Was definieren die? Was muss ich conformen, welche Freiheiten habe ich? https://dvcs.w3.org/hg/editing/raw-file/tip/editing.html

## 4.2 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

## 4.3 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 schÃűne arbeitsweise (am ende Ãijber bord geworfen) und kleine DateigrÃűÃ§e, wenig overhead.

Automatisierte Client side Tests mit PhantomJs und Mocha/Chai

## 4.4 Coding conventions

Habe mich grÃűÃ§tenteils 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. FÃijr private Eigenschaften und Methoden habe ich die prefix convention verwedendet. 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.

## 4.5 Coding Klassen

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

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

## 4.6 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. FÃijr 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 fÃijr Entwickler sichtbar bereit stelle und nicht implizit und versteckt Ãijber irgend nen Quatsch.

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

Ich werde jetzt die einzelnen Module erlÃďren

## 4.7 Type

Die Type

## 4.8 Caret

caret

## 4.9 Range

range

## 4.10 Selection

selection

## 4.11 Selection Overlay

overlay

## 4.12 Input

input

## 4.13 Formatting

formatting

## 4.14 Change Listener

change

## 4.15 Contents

contents

## 4.16 Development

developmnent

## 4.17 Dom Utilities

dom util

## 4.18 Dom Walker

dom walker

## 4.19 Environment

env

## 4.20 Core Api

core api

## 4.21 Event Api

ev api

## 4.22 Events

**Input** input event only event required so far

## 4.23 Input Pipeline

pipeline ideas

**Caret** caret

**Command** command

**Headlines** head lines

**Line Breaks** line breaks

**Lists** lists

**Remove** remove

**Spaces** spaces

## 4.24 Plugin Api

plugin api

## 4.25 Settings

settings

## 4.26 Text Walker

text walker

## 4.27 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