# 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 very limited and unstable. Bugs and unwanted behavior make it hard to use and can only be worked around, not fixed. By reviewing the APIs history, it can be argued that its design has never been revisited only to ensure compatibility to current browsers. This thesis explains the APIs downsides and demonstrates that rich-text editing can be achieved without requiring the `contenteditable`-attribute with library "Type", thus solving many problems that can be found contemporary third-party rich-text editors.

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** **15**

1.1 Motivation . . . . . 15

1.2 Terminilogy . . . . . 17

1.3 Structure . . . . . 17

**2 Text editing in desktop environments** **19**

2.1 Basics of plain-text editing . . . . . 19

2.2 Basics of rich-text editing . . . . . 19

2.3 Libraries for desktop environments . . . . . 19

**3 Text editing in browser environments** **21**

3.1 Plain-text inputs . . . . . 22

3.2 Rich-text editing . . . . . 22

3.3 HTML Editing APIs . . . . . 22

3.4 Discussion of HTML Editing APIs . . . . . 24

3.5 Browser support . . . . . 24

3.6 Emergence of HTML editing JavaScript libraries . . . . . 26

3.7 Standardization of HTML Editing APIs . . . . . 27

3.8 Usage of HTML Editing APIs for rich-text editors . . . . . 28

3.9 HTML Editing APIs are questionable . . . . . 29

3.10 Advantages of HTML Editing APIs . . . . . 30

3.11 Disadvantages of HTML Editing APIs . . . . . 31

3.12 Treating HTML editing API related issues . . . . . 35

3.13 Rich-text editing without editing APIs . . . . . 35
3.14 Rich-text without HTML editing APIs in practice . . . . . 38
3.15 Advantages of rich-text editing without editing APIs . . . . . 39
3.16 Disadvantages of rich-text editing without Editing APIs . . . . . 41
3.17 Comparison of these approaches . . . . . 43

**4 Concept** **45**

4.1 Approaches for enabling rich-text editing . . . . . 45
4.2 Mimicking native components for rich-text editing . . . . . 49
4.3 Wrapping it up . . . . . 50
4.4 Conformity with HTML Editing APIs . . . . . 50
4.5 Leave as much implementation to the browser as possible . . . . . 50
4.6 Markup . . . . . 50
4.7 MVC . . . . . 50
4.8 Events . . . . . 51
4.9 Stability and performance . . . . . 51
4.10 Disadvantages of offering user interface components . . . . . 52
4.11 iFrame . . . . . 53
4.12 API Design . . . . . 53

**5 Implementation** **55**

5.1 JavaScript library development . . . . . 55
5.2 Ich habe verwendet . . . . . 55
5.3 Coding conventions . . . . . 56
5.4 Class pattern . . . . . 56
5.5 Programmstruktur . . . . . 56
5.6 Type . . . . . 57
5.7 Range . . . . . 57
5.8 Input . . . . . 57
5.9 Caret . . . . . 59
5.10 Selection . . . . . 60

5.11 Selection Overlay . . . . . 60
5.12 Formatting . . . . . 60
5.13 Change Listener . . . . . 60
5.14 Contents . . . . . 60
5.15 Development . . . . . 60
5.16 Dom Utilities . . . . . 60
5.17 Dom Walker . . . . . 61
5.18 Environment . . . . . 61
5.19 Core Api . . . . . 61
5.20 Event Api . . . . . 61
5.21 Events . . . . . 61
5.22 Input Pipeline . . . . . 61
5.23 Plugin Api . . . . . 62
5.24 Settings . . . . . 62
5.25 Text Walker . . . . . 62
5.26 Utilities . . . . . 63
5.27 Extensions . . . . . 63

**6 Evaluation** **65**

6.1 Mobile Support . . . . . 65
6.2 Development / Meta . . . . . 65

**A Tables** **67**

**B Figures** **69**

# List of Figures

4-1 Rendering of highlighted source code in Ace and CodeMirror . . . . . 48

# List of Tables

2.1 Rich-text components in desktop environments . . . . . 20

3.1 Editing API attributes . . . . . 23

A.1 HTML Editing API . . . . . 67

A.2 `execCommand` command implementations with DOM methods . . . . 68

A.3 Rich-text editors using HTML editing APIs . . . . . 68

A.4 Rich-text editors not using HTML editing APIs . . . . . 68

# 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.

*Rich-text editing on the web is a particularly overlooked topic. Most libraries use contenteditable without questioning its benefits. The literature on this topic is thin. It*

[1] http://stackoverflow.com/questions/11240602/paste-as-plain-text-contenteditable-div-textarea-word-excel/11290082112900082, last checked on 07/13/2015

*is rarely written about in books and papers and noone really examines alternative ways for implementation. ACE and CodeMirror show techniques how to do it. However looking at its history, it seems very questionable. People who implement editors using it often rant about its disadvantages. <- der letzte Satz sollte einer der kernpunkte der introduction sein. ÃIJberahupt, das questioning sollte im kern stehen.*

## 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

Developing a text editor in a browser environment differs from implementing a text editor in a desktop einvironment. Any implementation is based on the axioms of the browsers. Development is generally restricted to the components and APIs offered by the HTML5 standard and experimental features that are usually implemented in a subset of browsers[1]. However, the boundaries of these restrictions can be pushed. It is common practice to combine native elements and APIs[2] in ways they have not been designed for to enable features not natively offered. These techniques are often reffered to as ”hacks” and, despite their terminolagy, are generally not regarded as a bad practice.

This chapter will describe the basics of text and rich-text editing in browsers as well as the APIs and techniques used. The origins and the history of rich-text editing pose the question if the paradigms rich-text editing is based on have been thoroughly reviewed and if alternative ways for an implementation, possibly using hacks, could be considered.

[1] If a specific component or API can be used is determined by the intended audience and browser market share.

[2] Native to the browser, not the operating system.

## 3.1 Plain-text inputs

Text input components for browsers have been introduced with the specification of HTML 2.0[3]. 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%[4], 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 3.3 through section 3.8.

## 3.3 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`[5][6]. These attributes were not part of the W3C's HTML 4.01 specifications[7] or the ISO/IEC 15445:2000[8], the defining standards of that time. Table 3.1 lists these attributes and possible values.

[3]https://tools.ietf.org/html/rfc1866, last checked on 07/15/2015

[4]http://gs.statcounter.com/all-browser-ww-monthly-201406-201506-bar, last checked on 07/15/2015

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

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

[7]http://www.w3.org/TR/html401/, last checked on 07/14/2015

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

| 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

```
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[9]. 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

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

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

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[10].

## 3.4 Discussion of HTML Editing APIs

HTML editing APIs have been poorly designed. Sections V to W discuss the origins and the reasons why HTML editing APIs have been standardized and are supported by all major browser. It can be seen, that they did not came to be because of their good design. In sections X and Y will discuss the design of these APIs.

## 3.5 Browser support

With the release of Internet Explorer 5.5 and the introduction of editing capabilities, Microsoft released a short documentation[11], containing the attributes' possible values and element restrictions along with two code examples. Although a clear purpose has

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

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

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[12].

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)[13]. 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[14][15]). This includes the available command identifiers [16][17] as well as the markup generated by invoking commands[18].

In June 2006, Opera Software releases Opera 9[19], providing full support for contentEditable and designMode[20], followed by Apple in March 2008[21] providing full support Safari 3.1[22]. MDN lists full support in Google Chrome since version 4[23], released in January 2010[24].

Starting in November 2004, WHATWG members have started actively discussing

[12]https://blog.whatwg.org/the-road-to-html-5-contenteditable, last checked on 07/10/2015
[13]https://developer.mozilla.org/en/docs/Rich-Text_Editing_in_Mozilla, last checked on 07/10/2015
[14]https://msdn.microsoft.com/en-us/library/hh772123(v=vs.85).aspx, last checked on 07/10/2015
[15]https://developer.mozilla.org/en-US/docs/Midas, last checked on 07/10/2015
[16]https://developer.mozilla.org/en-US/docs/Midas, last checked on 07/10/2015
[17]https://msdn.microsoft.com/en-us/library/ms533049(v=vs.85).aspx, last checked on 07/10/2015
[18]https://developer.mozilla.org/en/docs/Rich-Text_Editing_in_MozillaInternet_Explorer_Differences, last checked on 07/10/2015
[19]http://www.opera.com/docs/changelogs/windows/, last checked on 07/10/2015
[20]http://www.opera.com/docs/changelogs/windows/900/, last checked on 07/10/2015
[21]https://www.apple.com/pr/library/2008/03/18Apple-Releases-Safari-3-1.html, last checked on 07/10/2015
[22]http://caniuse.com/feat=contenteditable, last checked on 07/10/2015
[23]https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Content_Editable, last checked on 07/10/2015
[24]http://googlechromereleases.blogspot.de/2010/01/stable-channel-update_25.html, last checked on 07/10/2015

to incorporate these editing APIs in the HTML5 standard. Through reverse engineering, the WHATWG developed a specification based on Microsoft's implementation[25] 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[26] and WHATWG's HTML Standard[27].

## 3.6 Emergence of HTML editing JavaScript libraries

Around the year 2003[28] 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"[29] 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[30].

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[31]. However, it should be noted, that some projects only use other projects' editors and some projects are stubs. Popular choices on GitHub include "MediumEditor", "wysihtml", "Summernote" and others.

[25] https://blog.whatwg.org/the-road-to-html-5-contenteditable, last checked on 07/15/2015
[26] http://www.w3.org/TR/html5/editing.html, last checked on 07/15/2015
[27] https://html.spec.whatwg.org/multipage/interaction.htmlediting-2, last checked on 07/15/2015
[28] compare *Meine Tabelle aller Editoren*
[29] Now distributed as "CKEditor"
[30] https://www.drupal.org/project/project_module, last checked on 07/16/2015
[31] https://github.com/search?o=desc&q=wysiwyg&s=stars&type=Repositories&utf8=%E2%9C%93, last checked on 07/16/2015

## 3.7 Standardization of HTML Editing APIs

It makes sense to use HTML editing APIs for rich-text editing. Microsoft's demos published with the release of these API 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 for browsers to offer a native 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[32], but none with rich-text capabilites. The WHATWG discussed various ways to specify rich-text editing for the upcoming HTML5 standard, including dedicated input fields. The issues that have been faced with that idea are

1. Finding a way to tell the browser which language the rich-text input should generate. E.g. should it output (the then popular) "bb" code, (X)HTML, Textile or something else?

2. How can browser support for a rich-text input be achieved?

Ian Hickson, editor of WHATWG and author of the HTML5 specification adresses these main issues in a message from November 2004[33]. He states

> *Realistically, I just can't see something of this scoped[sic] [the ability to specify a input 'language' for a text-area and possibly to specify a subset of language elements allowed] getting implemented and shipped in the default install of browsers.*

and agrees with Ryan Johnson, who states

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

[33] https://lists.w3.org/Archives/Public/public-whatwg-archive/2004Nov/0014.html, last checked on 07/16/2015

*Anyway, I think that it might be quite a jump for manufacturers. I also see that a standard language would need to be decided upon just to describe the structure of the programming languages. Is it worth the time to come up with suggestions and examples of a programming language definition markup, or is my head in the clouds?*

Ian Hickson finally concludes

*Having considered all the suggestions, the only thing I could really see as being realistic would be to do something similar to (and ideally compatible with) IE's "contentEditable" and "designMode" attributes.*

Mark Pilgrim lists this as milestone of the decision to integrate Microsoft's HTML editing APIs in the HTML5 standard.[34].

## 3.8 Usage of HTML Editing APIs for rich-text editors

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.

[34]https://blog.whatwg.org/the-road-to-html-5-contenteditable, last checked on 07/16/2015

## 3.9 HTML Editing APIs are questionable

Understanding the history of the HTML editing APIs, the reasons for their wide browser support and their final standardization are questionable. It can be doubted if they fit their purpose secifically well. In fact, all major browsers mimicked the API as implemented in Internet Explorer 5.5, even though there was no specification on it. It had to be reverse-engineered. The reasons for this have not been publically discussed. A reason may have been to be able to compete with the other browsers. Both, Microsoft's original implementation as well as Mozilla's adoption have been released in the main years of the so-called "browser wars". However Mozilla adopted Microsoft's API applying practically no change to it. It can be argued that this has been part of the browser wars. At this time, it was essential for any browser to be able to be compatible with as many websites as possible. Many websites were only optimized for a specific browser. To gain market share, it was essential to support methods that other browsers already offered and that have been used by the web developers. Being able to display websites just as good as their competitor may have been a key factor for Mozilla's decision to implement Microsoft's HTML editing APIs and not alter them in any way. Creating another standard would have been a disadvantage over the then stronger Internet Explorer.

As described in section ABC, other now popular browsers, i.e. Chrome, Safari and Opera, implemented these APIs only years later, when JavaScript libraries based on them have already been popular and widely used, which can be seen as a reason for these decisions. As described in section YZ, it has clearly been stated (see section 3.7), that the reason for standardizing these APIs for rich-text editing has been to ensure browser support.

The API itself stems from the time when the usage of the web was different from today, its future was still unknown and web applications like Google Docs have not even been thought of. It should be discussed if this API really is the answer to all problem and if it still fits (or ever fit) modern requirements for content management systems or web application. The advantantages, disadvantages and practical issues

will be discussed in sections x y z.

## 3.10 Advantages of HTML Editing APIs

HTML Editing APIs have some notable advantages which will be discussed in this section.

**Browser support** A fair reason for using HTML editing APIs is its wide browser support. caniuse.com lists browser support for 92.78% of all used web browsers[35]. I.e. 92.78% of all people using the web use browsers that have full support for HTML editing APIs.

**High-level API** HTML editing APIs offer high-level commands for formatting text. It requires little setup to implement basic rich-text editing. The browser takes care of generating the required markup.

**HTML output** HTML editing APIs modify and generate HTML. In the context of web development, user input in this format is likely to be useful for further processing.

**No need for language definitions** The WHATWG discussed dedicated rich-text inputs, for instance as an extension of the `textarea` component. Offering a native input for general rich-text input brings up the question which use-cases this input conforms. For a forum software, it might be useful to generate ”BB” code, while for other purposes other languages might be needed. Offering HTML editing APIs offers a semantically distinct solution, while still enabling a way to implement rich-text editing.

**Possbile third-party solutions for other languages** While HTML editing APIs can be used to generate HTML only, third-party libraries can build on top of that

[35]http://caniuse.com/search=contenteditable, last checked on 07/17/2015

by implementing editors that write ”BB” code (for instance) and use HTML only for displaying it as rich-text.

## 3.11 Disadvantages of HTML Editing APIs

**No specification on the generated output** The specifications on the HTML editing APIs do not state what markup should be generated by specific commands. There are vast differences in the implememtations of all major browsers. Calling the `italic` command, this is the output of Internet Explorer, Firefox and Chrome:

```
1 <i>Lorem ipsum</i>
```

Listing 3.4: Markup of italic command in Internet Explorer

```
1 <span style="font-style: italic;">Lorem ipsum</span>
```

Listing 3.5: Markup of italic command in Firefox

```
1 <em>Lorem ipsum</em>
```

Listing 3.6: Markup of italic command in Chrome

This is a *major* problem for web development, because it makes processing input very difficult. Given the number of possible edge cases, it is very hard to normalize the input. Apart from that Internet Explorer's output is semantically incorrect for most use cases[36] while Firefox's output is breaking sementics entirely and considered a bad style regarding the principle of the separation of concerns[37]. Different browsers will not only generate differnent markup when executing commands. When a user enters a line break (by pressing enter), Firefox will insert a `<br>` tag, Chrome and Safari will insert a `<div>` tag and Internet Explorer will insert a `<p>` tag.

[36] https://developer.mozilla.org/en-US/docs/Web/HTML/Element/i#Notes, last checked on 07/17/2015

[37] https://en.wikipedia.org/wiki/Separation_of_concerns#HTML.2C_CSS.2C_JavaScript, last checked on 07/17/2015

**Flawed API** The original and mostly unaltered API is limited and not very effective. MDN lists 44 commands available for their `execCommand` implementation[38]. While other browsers do not match these commands exactly, their command lists are mostly similar. 17 of those commands format the text (for instance to italicize or make text bold) by wrapping the current selection with tags like `<em>` or `<strong>`. The only difference between any of these commands is which tag will be used. At the same time there is no command to wrap the selected text in an arbitrary tag, for instance to apply a custom class to it (`<span class="highlight">Lorem ipsum</span>`). All 17 commands could be summarized by a single command that allows to pass custom tags or markup and wraps the selected text with it. This would not only simplify the API, but would also give it enormously more possibilities. The same goes for inserting elements. 7 commands insert different kinds of HTML elements, this could be simplified and extended by allowing to insert any kind of (valid) markup or elements with a single command.

Both alternatives would also give developers more control of what to insert. As previously described, browsers handle formatting differently. Allowing to format with specific HTML would generate consistent markup (in the scope of a website) and allow developers generate the markup fitting their needs.

**Not extendable** Google points out that implementing an editor using HTML editing APIs comes with the restriction that such an editor can only offer the least common denominator of functions supported by all browsers. They argue, if one browser does not support a specific feature or its implementation is buggy, it cannot be supported by the editor[39]. This is mostly true, although it is to be noted, that editors like CKEditor show, that some bugs can be worked around as well as some functionality be added through JavaScript. These workaround still have limitations and not everything can be fixed. In particular there can be cases where the editing mode is not able to handle content inserted or altered by workarounds.

[38] https://developer.mozilla.org/en-US/docs/Web/API/document/execCommand#Commands, last checked on 07/17/2015

[39] http://googledrive.blogspot.fr/2010/05/whats-different-about-new-google-docs.html, last checked on 07/18/2015

**Clipboard** When dealing with user input, usually some sort of filtering is required. It is possibly harmful to accept any kind of input. This must be checked on the server side since attackers can send any data, regardless of the front end a system offers. However, in a cleanly designed system, the designated front end should not accept and send "bad" data to the back end. This applies to harmful content as well as to content that is simply *unwanted*. For example, for asthetical reasons, a comment form can be designed to allow bold and italic font formatting, but not headlines or colored text.

Implementing a rich-text editor with HTML editing APIs, unwanted formatting can be prevented by simply not offering input controls for these formattings (assuming no malicious behavior by the user). However any content, containing any formatting, can be pasted into elements in editing mode from the clipboard. There is no option to define filtering of some sort.

Recent browser versions allow listening to paste events. Chrome, Safari, Firefox and Opera grant full read access to the clipboard contents from paste events, which can then be stopped and the contents processed as desired. Internet Explorer allows access to plain-text and URL contents only. Android Browser, Chrome for Android and IOS Safari allow reading the clipboard contents on paste events as well. Other browsers and some older versions of desktop and mobile browsers do not support clipboard access or listening to paste events. 82.78% of internet users support listening to and reading from clipboard events[40].

When dealing with the clipboard, especially older browsers show an unexpected behavior. Older WebKit-based browsers insert so-called "Apple style spans"[41] on copy and paste commands. "Apple style spans" are pieces of markup that have no visible effect, but clutter up the underlying contents of an editor. When pasting formatted text from Microsoft Word, Internet Explorer inserts underlying XML, that Word uses to control its document flow into the editor.

[40] http://caniuse.com/#feat=clipboard, last checked on 07/18/2015
[41] https://www.webkit.org/blog/1737/apple-style-span-is-gone/, last checked on 07/18/2015

**Bugs** HTML editing APIs are prone to numerous bugs. Especially older browser versions are problematic. Piotrek Koszuliński states:

> *"First of all... Don't try to make your own WYSIWYG editor if you're thinking about commercial use. [...] I've seen recently some really cool looking new editors, but they really doesn't[sic] work. Really. And that's not because their developers suck - it's because browsers suck."*[42]

Mozilla lists 1060 active issues related to its "Editor" component[43]. Google lists 420 active issues related to "Cr-Blink-Editing"[44]. The WebKit project lists 641 active issues related to "HTML Editing"[45]. Microsoft and Opera Software allow public access to their bug trackers. Some rich-text editors like CKEditor have been developed for over 10 years and still need to fix bugs related to the editing API[46][47]. Some bugs have caused big websites to block particular browsers entirely[48].

Given the argument that editing APIs provide easy to use and high-level methods to format text, in practice, the number of bugs and work-arounds required, renders a "easy and quick" implementation impossible. Also, browser bugs cannot be fixed by web developers. At best they can be worked around, enforcing particular software design on developers and possibly spawning more bugs.

---

[42] http://stackoverflow.com/questions/10162540/contenteditable-div-vs-iframe-in-making-a-rich-text-wysiwyg-editor/11479435#11479435, last checked on 07/18/2015

[43] https://bugzilla.mozilla.org/buglist.cgi?bug_status=__open__&component=Editor&product=Core&query_forma last checked on 07/18/2015

[44] https://code.google.com/p/chromium/issues/list?q=label:Cr-Blink-Editing, last checked on 07/18/2015

[45] https://bugs.webkit.org/buglist.cgi?query_format=advanced&bug_status=UNCONFIRMED&bug_status=NEW& last checked on 07/18/2015

[46] http://dev.ckeditor.com/report/2, last checked on 07/18/2015

[47] http://stackoverflow.com/questions/11240602/paste-as-plain-text-contenteditable-div-textarea-word-excel/11290082112900082, last checked on 07/18/2015

[48] https://medium.com/medium-eng/the-bug-that-blocked-the-browser-e28b64a3c0cc, last checked on 07/18/2015. Medium however has contacted Microsoft and lead them to fix this bug.

## 3.12 Treating HTML editing API related issues

The issues arising with HTML editing APIs cannot be fixed. Many libraries find workarounds to treat them. CKEditor, TinyMCE, that framework. Google Docs finds another way and does not use HTML editing APIs.

huge editor libraries, developed for 10 years trying to fix stuff libraries, not editors targeting inconstiencies they all can never know what's gonna happen medium

**Clipboard** CKEditor "fixes" paste problems by implementing a custom fake context menu and opening a modal with some instruction that the user should press ctrl+v. This is a UX nightmare. Other editors like retractor (oder so) sanitize any change to the editors contents for the case of input by paste events.

## 3.13 Rich-text editing without editing APIs

HTML editing APIs are the recommended way for implementing a web-based rich-text editor. There is no native text input that can display formatted text. The only way to natively display rich-text on a website is through the Document Object Model (DOM). Editors based on HTML editing APIs utilize the DOM to display their rich-text contents too. Only the editing (of the DOM), commonly phrased "DOM manipulation", is implemented with HTML editing APIs.

**Manipulation via the DOM APIs** Manipulating the DOM has been possible since the first implementations of JavaScript and JScript. It has been standardised in 1998 with the W3C's "Document Object Model (DOM) Level 1 Specification" as part of the "Document Object Model (Core) Level 1"[49].

The DOM and its API is the recommended[50] way to change a website's contents and—apart from HTML editing APIs—the only possibility *natively* implemented in any major browser. Popular libraries like jQuery, React or AngularJS are based on

[49] http://www.w3.org/TR/REC-DOM-Level-1/level-one-core.html, last checked on 07/10/2015
[50] recommended by the W3C and WHATWG

it. The API has been developed for 17 years and proven to be stable across browsers. Any DOM manipulation that can be achieved with HTML editing APIs can also be achieved using this API.

**Algorithm 1** Simplified text formatting pseudocode

```
1: Split text nodes at beginning and end of selection
2: Create a new tag before the beginning text node of the selection
3: Loop over all nodes inside selection
4: Move each node inside new tag
```

Algorithm 1 demonstrates a simplified procedure to wrap a text selection in a tag. To implement the `bold` commmand of `execCommand`, this procedure can be implemented using the `strong` tag. The text selection can be read with the browser's selection API[51][52].

**Algorithm 2** Simplified element insertion pseudocode

```
1: If{Selection is not collapsed}{
2: Split text nodes at beginning and end of selection
3: Loop over all nodes inside selection
4: Remove each node
5: Collapse selection}
6: Insert new tag at beginning of selection
```

Algorithm 2 demostrates a simplified procedure to insert a new tag and possibly overwriting the current text selection and thereby mimicking `execCommand`'s command based on insertion.

**Algorithm 3** Simplified text removal pseudocode

```
1: If{Selection is not collapsed}{
2: Loop over all nodes inside selection
3: Remove each node
4: Collapse selection}
5: Else{
6: Remove one character left of the beginning of the selection}
```

Algorithm 3 demonstrates a procedure to mimick the `delete` command of `execCommand`.

[51] https://developer.mozilla.org/en-US/docs/Web/API/Selection, last checked on 07/18/2015

[52] Internet Explorer prior version 9 uses a non-standard API https://msdn.microsoft.com/en-us/library/ms535869(v=VS.85).aspx, last checked on 07/18/2015

**Algorithm 4** Simplified element unwrapping pseudocode

```
1: Loop over all nodes inside element
2: Move each node before element
3: Remove element
```

Algorithm 4 demonstrates a procedure to unwrap an element, mimicking the `removeFormat` and `unlink` commands of `execCommand`. With formatting and removing text as well as inserting and unwrapping elements, we can find equivalents for all commands of the HTML editing APIs related to manipulating rich-text. Aside from this, editing APIs offer commands for undo/redo and clipboard capabilites as well as a command for selecting the entire contents of the editable elements. The latter can easily be implemented with the designated selection API, whereas chapter ch:implementation demonstrates an implementation for undo, redo and clipboard capabilities. Table A.2 lists all commands available for `execCommand` and possible equivilents with DOM Level 1 methods. However, these are just simplified examples. A specific implementation accounting for all commands and all edge-cases will be discussed in chapters ch:concept and ch:implementation.

**Third-party alternatives** The only alternative way to display and edit rich-text inside a browser is through third-party plugins like Adobe Flash or Microsoft Silverlight. Flash and Silverlight lack mobile adoptions and have been subject to critique since the introduction of smartphones and HTML5. Other third-party plugins are even less well adopted. This makes Flash, Silverlight and other third-party browser-plugins a worse choice as compared to displaying and manipulating rich-text though the DOM.

## 3.14 Rich-text without HTML editing APIs in practice

Google completely rewrote their document editor in 2010 abandoning HTML editing APIs entirely. In a blog post[53], they stated some of the reasons discussed in section XYZ. They state, using the editing mode, if a browser has a bug in a particular function, Google won't be able to fix it. In the end, they could only implement "least common denominator of features". Furthermore, abanding HTML editing APIs, enables features not possible before, for example tab stops for layouting, they state.

With their implementation, Google demonstrates it is possible to implement a fully featured rich-text editor using only JavaScript and not HTML editing APIs. However, fetching input and modifying text will not suffice to implement a text editor or even a simple text field. There is many more things, that need to be considered which will be discussed in chapter 4.

Google's document however is proprietary software and its implementation has not been documented publically. Most rich-text editors still rely on HTML editing APIs. The editor "Firepad"[54] is an exception. It is based on "CodeMirror", a web-based code editor, and extends it with rich-text formatting. The major disadvantage of Firepad, is based on its origin as a code editor. It generates "messy" (unsemantic) markup with lots of control tags. It has a sparse API that is not designed for rich-text editing. It has no public methods to format the text. It is to be noted that Google's document editor generates lots of control tags too, but it is only used within Google's portfolio of office app and it may not be necessary that it creates *well-formatted*, semantic markup. A full list of rich-text editors using and not using HTML editing APIs can be found in tables A.3 and A.4.

[53] http://googledrive.blogspot.fr/2010/05/whats-different-about-new-google-docs.html, last checked on 07/18/2015

[54] http://www.firepad.io/, last checked 07/23/2015

## 3.15 Advantages of rich-text editing without editing APIs

With a pure JavaScript implementation, many of the problems that editing APIs have, can be solved.

**Generated output and flawed API** The generated markup can be chosen with the implementation of the editor. Furthermore, an editor can be implemented to allow developers using the editor to *choose* the output. Section AX describes the inconsistent output across various browsers as well as the problems of the API design of `execCommand`. Both issues can be adressed by offering a method to wrap the current selection in arbitrary markup. jQuery's `htmlString` implementation[55] demonstrates a simple and stable way to define markup in a string and pass it as an argument to JavaScript methods. A sample call could read as

```
1 // Mimicking document.execCommand('italic', false, null);
2 editor.format('<em />');
3
4 // Added functionality
5 editor.format('<span class="highlight" />');
```

Listing 3.7: Example calls to format text

With an implementation like this, developers using the editor can choose which markup should be generated for italicising text. The markup will be consistent in the scope of their project unless chosen otherwise. Since the DOM manipulation is implemented in JavaScript and not by high-level browser methods, this will also ensure the same output on all systems and solve cross-browser issues. The second example function call in listing 5.1 demonstrates that custom formatting, fitting the needs of a specific project, can be achieved with the same API.

As described in section AB, many components native to text editing have to be

[55] http://api.jquery.com/Types/#htmlString, last checked on 07/19/2015

implemented in JavaScript. This requires some effort but also enables full control and direct over it. Ultimately, these components can be exposed in an API to other developers, enabling options for developing editors, not offered by HTML editing APIs. An example API will be discussed in sectionXImplementationn.

**Not extendable** When implementing and editor in pure JavaScripts, the limitations aufgedrÃijck by the editing mode, do not apply. Anything that can be implemented in a browser environment can also be implemented as part of a (rich-)text editor. The Google docuement editor demonstrates rich functionality incliding layouting tools or floating images. Both of which are features that are hardly possible in an editing mode enabled environment[56]. SectionABC discusses some use cases exploring the possibilites of rich-text editing implemented this way.

**Clipboard** In a pure JavaScript environment, clipboard functionality seems to be harder to implement than with the use of editing mode. Apart from filtering the input, pasting is natively available—via keyboard shortcuts as well as the context menu. However, as demonstrated in section IMPLEMENTATION, it is possible to enable native pasting—via keyboard and context menu—even without editing mode. Furthermore, it is possible to filter the pasted contents before inserting them in the editor.

**Bugs** No software can be guaranteed to be bug free. However, refraining from using HTML editing APIs will render developing an editor independent from all of these APIs' bugs. Going a step further, the implementation can be aimed to minimize interaction with browser APIs, especially unstable ones. DOM manipulation APIs have been standardized for more than 15 years and tend to be well-proven and stable. This will minimize the number of "unfixable" bugs and ultimately free development from being dependent on browser development, update cycles and user adoption. Bugs can be fixed quicker and rolled out to websites. This means that, other than

[56]http://googledrive.blogspot.fr/2010/05/whats-different-about-new-google-docs.html, last checked on 07/19/2015

with HTML editing APIs, bugs that occur are part of the library can be fixed and not only worked around. Furthermore, with minimizing browser interaction, bugs probably occur indeopendently of the browser used, which makes finding and fixing bugs easier.

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

**Formatting** Editing APIs' formatting methods take away a crucial part of rich-text editing. Especially on the web, where a text may have many sources, formatting must account for many edge cases. Nick Santos, author of Medium's rich-text editor states in regeards of their editor implementation:

> *"Our editor should be a good citizen in [the ecosystem of rich-text editors]. That means we ought to produce HTML that's easy to read and understand. And on the flip side, we need to be aware that our editor has to deal with pasted content that can't possibly be created in our editor."*[57]

An editor implemented *without* HTML editing APIs does not only need to account for content (HTML) that will be pasted into the editor[58] (in fact, content should be sanitized before it gets insterted in the editor, see sections A and b, paragraphs "clipboard"), but also for content that will be loaded on instantiation. It cannot be assumed that the content that the editor will be loaded with (for example integrated in a CMS), is *well-formatted* markup or even valid markup. "Well-formatted" means, the markup of a text is *simple* in the sense that it expresses semantics with as few tags as possible (and it conforms the standards of the W3C). The same visual representation of a text, can have many different—and valid—underlying DOM forms. Nick Santos gives the exaple of the following text[59]:

[57] http://stackoverflow.com/questions/11240602/paste-as-plain-text-contenteditable-div-textarea-word-excel/11290082112900082, last checked on 07/13/2015

[58] Medium uses HTML editing APIs

[59] http://stackoverflow.com/questions/11240602/paste-as-plain-text-contenteditable-div-textarea-word-excel/11290082112900082, last checked on 07/13/2015

The hobbit was a very well-to-do hobbit, and his name was ***Baggins***.

The word "Baggins" can be written in any of the following forms:

```
1 <strong><em>Baggins</em></strong>
2 <em><strong>Baggins</strong></em>
3 <em><strong>Bagg</strong><strong>ins</strong></em>
4 <em><strong>Bagg</strong></em><strong><em>ins</em></strong>
```

Listing 3.8: Different DOM representations of an equally formatted text

A rich-text editor must be able to edit any of these representations (and more). Furthermore, the same edit operation, performed on any of these representations must provide the same *expected* behavior, i.e. generate the same visual representation and produce predictive markup. Above that, being a "good citizen" it should produce simple and semantically appropriate HTML even in cases when the given markup is not. It may even improve the markup it affects.

**Mimicking native functionality** As described in section XY, rich-text editing consists of many components like the caret or the ability to paste text via a context menu. These basic components require complex implementations. HTML editing APIs offer these components natively without further scripting.

**Possible performance disadvantages** Modifying the text on a website means manipulating the DOM. DOM operations can be costly in terms of performance as they can trigger a browser reflow[60]. The same goes for mimicking some elements like the caret. To display a caret a DOM element like a `div` is needed and it needs to be moved by changing its style attribute. While it should be a goal to keep browser interactions to a minimum, there is no way to avoid DOM interaction with any visual text change.

**File size** While bandwidth capacities have vastly improved, there may still be situations where a JavaScript libraries' file sizes matter. This may be for mobile appli-

[60] https://developers.google.com/speed/articles/reflow, last checked on 07/19/2015

cations or for parts of the world with less developed connections. When not using HTML editing APIs a lot of code must be written and transmitted just to enable basic text editing, which would not be needed otherwise.

## 3.17 Comparison of these approaches

HTML editing APIs vs other method, pros and cons, conclusion. Also HTML editing APIs will get better, still the API sucks, fewer features, also still dependent on browser development in terms of possible bugs in the future and further features. Edge casese can be tackled and solutions rolled out immediately. Cross browser inconsistencies are less of a problem and can also be tackled as required.

# Chapter 4

# Concept

## 4.1 Approaches for enabling rich-text editing

This section will discuss the options to implement rich-text editing without relying (entirely) on HTML editing APIs and discuss the advantages and disadvantages of each method.

**Native input elements** Native text inputs are hard-wired to plain-text editing. No major browser offers an API for formatting. There is also no option to write HTML to an input and have it display it as rich-text. `input` fields and `textarea` elements will simply display the HTML as source code. Rich-text can only be implemented as an editable part of the website.

**Image elements** In February 2015, Flipboard Inc. demonstrated an unprecedented technique to achieve fluid full-screen animations with 60 frames per second on their mobile website[1]. Instead of using the DOM to display their contents, the entire website was rendered to a `canvas` element. When a user swiped over the website the canvas element was rerendered, essentially imitating the browser's rendering engine. `canvas` elements allow rendering rich-text too. A rich-text editor can be implemented using this technique. This however has two major downsides. On the one hand it

[1] http://engineering.flipboard.com/2015/02/mobile-web/, last checked on 07/24/2015

would require implementing a text-rendering engine. The `canvas` API is only capable of displaying unlayouted text with specifically set line breaks. On the other hand, making the editor accessible to other developers would be much more complex since the text only exists in an internal representaion inside the editor and would not be exposed as DOM component on the website.

An approach related to rendering the text on a `canvas` element is to render the text inside a Scalable Vector Graphic (SVG). In contrast to `canvas` elements, SVGs contain DOM nodes that can be accesed from the outside. However this has no benefit over using HTML DOM nodes with the downside that SVG too has no native implementation for controlling the text flow.

**Enabling editing mode without using its API** One way to enable editing but avoiding many bugs and browser inonstistencies, is to enable the editing mode on an element, but avoid using `execCommand` to format the text. The latter could be implemented using the DOM core APIs. This would provide the user with all basic editing functions, i.e. a caret, text input, mouse interaction and clipboard capabilites. All of this would be taken care of by the browser.

This sem-radical approach would solve the problem of buggy and inconsistent `execCommand` implementations but not the problems that arise with different browser behavior on the user's text input—for instance when entering a line break. If the markup is customly generated with JavaScript, the input may break elements or simply get stuck. This was one of the reasons why Google decided to abandon editing APIs entirely[2]. It could be the source to many bugs, have the development of the library be dependent on browser development and ultimatively restrict the editors capabilites.

**Native text input imitation** The only other option to allow the user to change the text on a website is by manually fetching the user's input and manipulating the DOM with JavaScript. However, this does not suffice to provide the experience

[2]http://googledrive.blogspot.fr/2010/05/whats-different-about-new-google-docs.html, last checked on 07/21/2015

of a text input. The following components, common to text editing, must also be accounted for:

**Caret** The most obvious part is probably the text caret. Even if a user types on his or her keyboard, a caret must be seen on the screen to know where the input will be inserted. The caret also needs to be responsive to the user's interaction. In particular, the user must be able to click anywhere in the editable text and use the arrow keys to move it (possibly using modifier keys, which's behaviour even depends on the operating system used).

**Selection** Just like the caret, the user must be able to draw a text selection using his or her mouse and change the selection using shift and the arrow keys. Most systems allow double clicks to select words and sometimes tripple clicks to select entire paragraphs. Other systems, for example OS X, allow holding the option key to draw are rectengular text selection, independent of line breaks.

**Context menu** The context menu is different in text inputs from other elements on a website. Most importantly, it offers an option to paste text, that is only available in native text inputs or elements in editing mode.

**Keyboard shortcuts** Text inputs usually allow keyboard shortcuts to format the text and to perform clipboard operations. Formatting the text is possible through DOM manipulation, pasting text however natively only works on text inputs or elements in editing mode.

**Undo / Redo** Undo and redo are common functions of text processing and it may be frustrating to users if they were missing.

**Behaviour** Rich-text editors (usually) share a certain behaviour on user input. When writing a bulleted list, pressing the enter key usually creates another bulleting point instead of inserting a new line. Pressing enter inside a heading will insert a

new line. However pressing enter when the caret is located at the end of a heading commonly creates a new text paragraph just after that heading. Other rules that need to be considered will be discussed in section [implementation].

**Imitating native components** These components are natively available for text inputs across all browsers. Switching an element to editing mode enables these components too. That means a user can click in a text to place a caret and move it with the keyboard's arrow keys. He or she can copy and paste text. The browser offers a native context menu that allows pasting on input elements as well as on element in editing mode. All major browsers implement a behaviour for the user' input that is common for rich-text editing (as described above).

When not using editing APIs, all of this must be implemented with JavaScript. This requires a lot of trickery and many components must be imitated to make it *seem* there is an input field, where there is none. The user must be convinced he or she is using a native input and must not notice he or she is not.

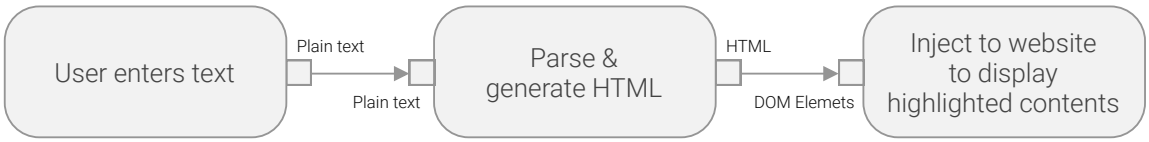


Figure 4-1: Rendering of highlighted source code in Ace and CodeMirror

Web-based code editors like "Ace" and "CodeMirror" demonstrate that this is possible. They display syntax-highlighted source code editable by the user. The user seemingly writes inside the highlighted text and is also presented with a caret as well as the above mentioned components. In reality, the content inside the editor that a user sees is a regular part of the DOM—non-editable text, colored and formatted using HTML and CSS. When the user enters text, the input will be read with JavaScript. Based on the input Ace and CodeMirror generate HTML and add it to the editors contents, to show a properly syntax-highlighted representation (see figure 4-1). A `div` element that is styled to look like a caret is shown and moved with the user's keyboard and mouse input. The user's text input will be inserted at the according text offset. Amongst others, Ace and CodeMirror use other elements like `div`s to display a

text selection and a `textarea` to fetch keyboard inputs, to recreate the behavior and capabilites of a native text input. Google's document editor uses similar techniques too.

Using tricks and *faking* elements or behavior is common in web front end development. This applies to JavaScript as well as to CSS. For instance, long before CSS3 has been developed, techniques (often called "hacks") have been discussed on how to implement rounded corners without actual browser support. Only years later, this has become a standard. This not only enables features long before the creators of browsers implement them, this *feedback* by the community of web developers also influences future standards. Encorporating feedback is a core philisophy of the WHATWG, the creators of HTML5.

## 4.2 Mimicking native components for rich-text editing

The techniques used in Ace, CodeMirror and Google's document editor for imitating a native text input are similar. For the caret, each of the editors create a `div` element, style it to make it look like a caret[3] and move it when the user clicks in the text or uses his or her arrow keys. Each editor uses `div` elements to display a text selection, styled to imitate the user's operating system's native selections.

The concept to use HTML elements for all visible components that are required for text editing is inevitable. Anything that can be seen on a website must exist in the DOM. The only deviation to this approach would be to resort a `canvas` element, as discussed above. This apporach will be used for the implementation of this editor.

[3] inlcuding a blinking animation

## 4.3 Wrapping it up

## 4.4 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

## 4.5 Leave as much implementation to the browser as possible

"Easy to make it fast – The browser (not the app) handles the most computationally intensive task: text layout. Since layout is a core component of browser functionality, you can trust that layout performance has already been heavily optimized." http://googledrive.blogspot.fr/2010/05/whats-different-about-new-google-docs.html

## 4.6 Markup

Our editor should be a good citizen in this ecosystem. That means we ought to produce HTML thatâĂŹs easy to read and understand. And on the flip side, we need to be aware that our editor has to deal with pasted content that canâĂŹt possibly be created in our editor. https://medium.com/medium-eng/why-contenteditable-is-terrible-122d8a40e480

Also FirePad and Google generate unusable markup. we're gonna generate semantic markup, simple and clean.

## 4.7 MVC

Document model -> no UrsprÃijnglich MVC mÃďÃ§ig mit document im kern geplant

## 4.8 Events

Dazu entscheiden dass es ein Type-Internes Eventsystem gibt, dessen Events nur in Type existieren. Damit polluten die nicht den globalen (name)space mit events. Nur bestimmte events werden global (nativ) getriggert, und das sind genau die events, die nach aussenhin von Belang sind und abgegriffen werden koennen sollen (=API design)

## 4.9 Stability and performance

**Cache** For traversing the text, for example when the caret moves, the text will need to be measured. All measurements can be stored to a cache to only perform the same measurement operations once. On the other hand, the library should be a "good citizen" on a website, which means it should be as unobtrusive and leave the developers as much freedom as possible. The library will essentially modify parts of the DOM that act as the editor's contents. It should be agnostic to the editor's contents at all times to give other developers the freedom to change the contents in any way needed without breaking the editor. A cache must account for external changes. The DOM3 Events specification[4] offers `MutationObserver`s to check for DOM changes. This feature is not supported by Internet Explorer version 10 or less[5]. Internet Explorer 9 and 10 offer an implementation for `MutationEvents`[6]. The W3C states that "The MutationEvent interface [...] has not yet been completely and interoperably implemented across user agents. In addition, there have been critiques that the interface, as designed, introduces a performance and implementation challenge."[7]. Apart from that, the benefits of a cache may not signifcantly increase the library's performance. The actions that can be supported by a cache, most importantly moving the caret in the text, are not very complex and do not noticably

[4]http://www.w3.org/TR/DOM-Level-3-Events/, last checked on 07/21/2015
[5]http://caniuse.com/#search=mutation, last checked on 07/21/2015
[6]http://help.dottoro.com/ljfvvdnm.php#additionalEvents , last checked on 07/21/2015
[7]http://www.w3.org/TR/DOM-Level-3-Events/#legacy-mutationevent-events, last checked on 07/21/2015

afflict the CPU. Implementing an editor that is stateless in regards of its contents can also improve stability.

**Minimze interaction with the DOM** DOM operations are slow and should be avoided. DOM operations can be ”collected” and only executed when necessary etc pp like React.

**Minimze interaction with unstable APIs** Some APIs like the `Range` or `Selection` are prone to numerous bugs. To maximize stability, these APIs should be avoided when possible unless doing so has any downsides (like lower performance).

## 4.10 Disadvantages of offering user interface components

Most rich-text editors are implemented and distributed as user interface components. That means instead of only providing a library that offers methods to format the selected text and leaving the implementation of the user interface to the respective developer, most libraries are shipped as input fields with a default editor interface that is, at best, customizable.

This can be unfitting for many situations. The user interface of an editor highly depends on the software it will be integrated in. Within the software the interface may even vary depending on the specific purpose within it. For instance, a content management system may require an edtitor with a menubar offering many controls while a comment form on a blog requires only very little controls. Medium.com uses an interface that only shows controls when the user selects text and has no menubar at all. Assuming there are many implementations of editors, it seems, choosing between editors is often just a choice of the desired ui. Customizing a ui can be just as complex as writing a ui from scratch. The latter affords to add HTML elememts and call JavaScript methods. In a worst case styling the elements can be just as much effort. That is why the library ”Type” will be implemented and offered as a software

library, rather than an editor and a user interface component.

It can be much more fitting for developers to include a library that handles all text-input and -formatting operations while only providing a powerful API, leaving the ui to the developer. The API must be *well-designed*. That means it must be simple, effective and fit the developers' needs. The methods it offers should be simple in the sense that they conceal possibly complex tasks with understandable high-level concepts. They should be effective and fit the developers' needs in the sense that the API should be designed so that any requirements should be matched with as little effort as possible. The API should create a workflow for developers that allows them to do what they want to do and is as easy to use as possible. jQuery is an example of encorporating an API conforming these principles.

## 4.11 iFrame

iFrame hat vor und nachteile. mit meiner library kann es jeder so machen, wie er/sie es will, weil es nur ne library ist.

## 4.12 API Design

Allgemeine ÃIJberlegungen wie man ein Framework gestaltet (siehe "Freiheit fÃijr Programmierer", gute Arbeitsweise) UrsprÃijnglich API und Codestruktur an window.execCommand orientiert, das ist aber doof. Bessere (weil prÃďzisere) API mit mehr MÃűglichkeiten als der contentEditable scheiss fÃijr Programmierer und fÃijr 2 anwendungsfÃďlle: einen editor bauen type mit plugins erweitern fÃijr beides gibt es die passenden Funktionen, das eine einfach und schlau, das andere prÃďzise Deswegen werden auch alle Module, alle Klassen exponiert aber es gibt eine eigene API nach jQuery konzept Type kommt mit bestimmten Kernmodulen die fuer Textbearbeitung essenziell sind. Aber an dieser Stelle laesst sich Type um weitere Module anhand einer KOnvention (erweitern des Type Objekts) erweitern. UrsprÃijnglich API und Codestruktur an window.execCommand orientiert, das ist aber doof.

# Chapter 5

# Implementation

## 5.1 JavaScript library development

No IDEs, tools, not even conventions. Es gibt frameworks die eine architektur fÃijr webseiten und web-applikationen vorgeben, aber nichts dergleichen fÃijr bibliotheken.

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

ES5 oder ES6?

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

Automatisierte Client side Tests mit PhantomJs und Mocha/Chai

## 5.3 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.

## 5.4 Class pattern

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

Eine alternative wÃďre das Modul-pattern, das ist aber schlechter zu lesen und bietet die oben genannten vorteile nur teilweise.

## 5.5 Programmstruktur

Modulbasiert? Erweiterbarkeit 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 erlÃď'ren

## 5.6 Type

Die Type

## 5.7 Range

range

## 5.8 Input

There are many devices (hardware and virtual) a user can interact with native inputs:

1. Keyboard (as hardware and as virtual keyboard)
2. Mouse
3. Touch
4. Game controller (on browsers imlpemented in game consoles)
5. Remote control (on Smart TV einvironments)

When simulating a native input, in a best-case scenario, all these input methods should be accounted for. Fetching input needs to account for two scenarios: The user

clicks / touches / or selects the input in any way and does so at any position inside the input. If the user touches / clicks / etc in the middle of the text, the caret should move to that position and typing must be enabled. In environments without hardware keyboards, the libarary must ensure that a virutal keyboards, possible native to the system, show up. Once the input is selected, text input must be fetched and written to the editor. There are various options to fetch user input, which will be discussed in the following paragraphs:

**Events** One way to fetch user input is by listening to events. Text input can be read through `KeyboardEvent`s. Keyboard events will be triggered for virtual keyboards just as for hardware keyboards. When the user presses a key, the event can be stopped and the according characters can be inserted at the position of the caret. As a downside, listeners for keyboard events cannot be bound to a specific element that is not a native text input, that means keyboard events must be listened to on the `document` level. This not only has (minor) performance downsides but als requires more logic to decide whether a keyboard input should be processed and ultimatively stopped or ignored and allowed to bubble to other event listeners of a website. Furthermore, there can be edge cases, where even though a keyboard event should write contents to the editor, the event itself is supposed to trigger other methods that are not part of the editor. Keyboard events are supported by all major browsers across all devices.

To support clicking or touching inside the editor's contents `MouseEvent`s and `TouchEvent`s can be used. Mouse events are supported on all major desktop browsers and all mobile browsers support touch events. Both event types support reading the coordinates indicating where the click or touch has been performed.

Although some smart TVs offer keyboards, mice, pointers similar to Nintendo's Wii remote, input via smartphone apps and many others, button-based remote controls are offered with almost any smart TV and remain a edge case for interacting with a text editor. In such an enviroment, users commonly switch between elements by selecting focusable elements with a directional pad. Using only events would not

account for this case since there would be no focusable element representing the editor. Recent browsers on Samsung's and LG's smart TVs are based on WebKit[1] while Sony's TVs use Opera. Before 2012 Samsung's browser was based on Gecko. All of these browsers and browser engines supprt keyboard events to fetch input.

**Clipboard** Another problem with relying entirely on events is the lack of native clipboard capabilities. Unless a native text input (including elememts with enabled editing mode) is focused, shortcut keys for pasting will not trigger a paste event and the mouse's context menu will not offer an option for pasting. Recent versions of Chrome, Opera and Android Browser[2] allow triggering arbitrary paste events thereby read the cliboard contents. With this, shortcuts could be enabled with JavaScript and instead of the native context menu, a customly build context menu using HTML could be shown that allows the user to paste, but this only works on elements in editing mode and only in these three browsers.

**Hidden native input fields** Apart from rich-text editors there are also third-party JavaScript libraries that allow embedding code editors with syntax highlighting in a website. The "Ajax.org Cloud9 Editor (Ace)" and "CodeMirror" editor are amongst the most well-known choices. Both editors keep an internal representation of the plain-text (i.e. non-highlighted) contents of the editor, parse it and display a highlighted version using HTML in a designated `div`. Both editors use a hidden `textarea` in which the user enters his or her input. The input will be read from the `textarea` and

## 5.9 Caret

caret

### BiDi support

**IME** http://marijnhaverbeke.nl/blog/browser-input-reading.html https://en.wikipedia.org/wiki/Inp

[1] http://www.samsungdforum.com/Devtools/Sdkdownload, last checked on 07/22/2015
[2] http://caniuse.com/#feat=clipboard, last checked on 07/22/2015

## 5.10 Selection

Fake damit copy & past nativ funktioniert Obwohl eine selection mehrere Ranges haben kann, wird das von keinem Browser unterstÃijtzt http://stackoverflow.com/questions/4777860/hi select-multiple-divs-with-ranges-w-contenteditable/4780571#4780571 Habe versucht mit die Darstellung Ãijber getClientRects zu emulierten, aber das ist buggy siehe https://github.com/edg2s/rangefix/blob/master/rangefix.js

## 5.11 Selection Overlay

overlay

## 5.12 Formatting

formatting

## 5.13 Change Listener

change

## 5.14 Contents

contents

## 5.15 Development

developmnent

## 5.16 Dom Utilities

dom util

## 5.17 Dom Walker

dom walker. Used SO OFTEN in the project. Goal is to have a really simple api new DomWalker('text') by having lots of pre-defined magick in it.

## 5.18 Environment

env

## 5.19 Core Api

Api als eigenes Modul, separation of concerns, gute saubere code base und eigene saubere code base fÃijr die API. Laesst freiheiten in der Softwarearchitektur - lost coupling, API ein eigenes aenderbares gekapseltes modul:

## 5.20 Event Api

ev api

## 5.21 Events

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

## 5.22 Input Pipeline

pipeline idea rules for behaviours (lists, headlines, enter, spaces...) UrsprÃijnglich so gedacht: Was gehÃűrt zu einem editor X, Y und Behaviour. FÃijr Behaviour aber das input pipeline system gefunden, welches das ganze schÃűn abbildet/lÃűst. Noch mal skimmen und sehen ob da was dabei ist, besonders "Detour": http://marijnhaverbeke.nl/blog/browser-input-reading.html

**Pasting** Kontrolle durch die Implementierung mit einem versteckten Eingabefeld. Erst mal gibt es ein reliable Paste event, was es in keinem Editor basierend auf contentEditable gibt. 2. Hat man volle Kontrolle Ãijber den Inhalt, was gepastet wird und was nicht. Und das ganze mit einer einfachen API. Pasting sollte internes event ausloesen.

**Caret** caret

**Command** command

**Headlines** head lines

**Line Breaks** line breaks

**Lists** lists

**Remove** remove

**Spaces** spaces

## 5.23 Plugin Api

plugin api

## 5.24 Settings

settings

## 5.25 Text Walker

text walker

## 5.26 Utilities

util

## 5.27 Extensions

**Plugin API** ...and plugin cache

```
1  // Adding contructors
2  Type.fromEtherpad = function() { /* Implementation */ };
3
4  // Events to load plugins on instantiation
5  Type.on('ready', function(type) { /* Implementation */ });
6
7  // Lazy loading using plugin cache
8  Type.fn.myPluginMethod = function (tag, params) {
9    var cmd = this.plugin('cmd', /* Plugin instantiation */ );
10   /* Execute myPluginMethod */
11 };
12
13 // Static function calls just like jQuery
14 Type.fn.cmd = function (tag, params) { /* Implementation */ };
```

Listing 5.1: Example calls to format text

Erstellung von Erweiterungen und Plugins soll so einfach und frei wie mÃűglich sein. Es ist die Verantwortung der Programmierer diese Freiheit so zu nutzen, dass sie trotzdem sauber arbeiten. jQuery zeigt, dass Freiheit funktioneren kann und nicht zu schlechtem nutzen fÃijhrt. Schlechte Plugins sterben durch die Dynamik des "Marktes" aus. Die Vorteile ueberwiegen: Beim CKEditor ist es aufwÃd'ndig (viel Code um sich in den Editor zu integrieren) und schwierig (Lernkurve) Erweiterungen zu schreiben. Der CKEditor ist aber auch ein Produkt und kein Framework. Ein Framework sollte MÃűglichkeiten geben und einfach (und painless) sein. jQuery

hat fÃijr sich Konventionen erzeugt und MÃűglichkeiten gegen, die Programmierer effektiv arbeiten lassen kÃűnnen. Und jQuery zeigt: Es funktioniert.

**Etherpad** Collaboration with etherpad ist wirklich einfach zu machen mit meinem coolen editor. Mein Editor ist ganz ganz toll dass das so einfach geht, ja! Markdown wohl eher als Ausblick.

# Chapter 6

# Evaluation

## 6.1 Mobile Support

## 6.2 Development / Meta

Crockford style is a bad idea. I will change it to Standard or Airbnb https://github.com/airbnb/javascrip

# 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

| Command Identifier | Document Object Model (Core) Level 1 equivalent |
|---|---|
| bold | a |
| italic | b |

Table A.2: `execCommand` command implementations with DOM methods

| Attribute | Type | Can be set to | Possible values |
|---|---|---|---|
| designMode | IDL attribute | Document | "on", "off" |
| contenteditable | content attribute | Specific HTMLElements | empty string, "true", "false" |

Table A.3: Rich-text editors using HTML editing APIs

| Attribute | Type | Can be set to | Possible values |
|---|---|---|---|
| designMode | IDL attribute | Document | "on", "off" |
| contenteditable | content attribute | Specific HTMLElements | empty string, "true", "false" |

Table A.4: Rich-text editors not using HTML editing APIs

# Appendix B

# Figures

# Bibliography