

A WYSIWYG FRAMEWORK

MASTER'S THESIS

LUKAS BOMBACH

538587

26TH AUGUST 2015

SUPERVISORS:

PROF. DR. DEBORA WEBER-WULFF

PROF. DR. BARBARA KLEINEN

HTW BERLIN

INTERNATIONAL MEDIA AND COMPUTING (MASTER)

© 2015 Lukas Bombach

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0
International License.

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 API's history, it can be argued that its design has never been revisited only to ensure compatibility to current browsers. This thesis explains the API's downsides and demonstrates that rich-text editing can be achieved without requiring the `contenteditable`-attribute with library "Type", thus solving many problems of contemporary third-party rich-text editors.

Acknowledgements

I would like to extend my thanks to my two supervisors, Prof. Dr. Debora Weber-Wulff and Prof. Dr. Barbara Kleinen, 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

Contents	4
1 Introduction	7
1.1 Motivation	7
1.2 Terminology	9
1.3 Structure	9
I Theory	11
2 Principles of rich-text editing	12
2.1 Overview	12
2.2 Formatting	12
2.3 Behavior	12
3 Text editing in desktop environments	13
3.1 Basics of plain-text editing	13
3.2 Basics of rich-text editing	13
3.3 Libraries for desktop environments	13
4 Text editing in browser environments	15
4.1 Overview	15
4.2 Plain-text editing	15
4.3 Rich-text editing	16
4.4 HTML Editing APIs	16
4.5 Usage of HTML Editing APIs for rich-text editors	18

<i>CONTENTS</i>	5
II Discussion	20
5 Overview	21
6 History HTML editing APIs	22
6.1 Browser support	22
6.2 Emergence of HTML editing JavaScript libraries	23
6.3 Standardization of HTML Editing APIs	24
7 Advantages and disadvantages	26
7.1 Discussion	26
7.2 Advantages of HTML Editing APIs	27
7.3 Disadvantages of HTML Editing APIs	28
7.4 Treating HTML editing API related issues	32
8 Rich-text editing without editing APIs	36
8.1 Alternatives to HTML editing APIs	36
8.2 Rich-text without HTML editing APIs in practice	40
8.3 Advantages of rich-text editing without editing APIs	41
8.4 Disadvantages of rich-text editing without editing APIs	43
III Concept	46
9 Approaches for enabling rich-text editing	47
9.1 Overview	47
9.2 Native inputs, images and third-party plugins	47
9.3 Enabling editing mode without using its API	47
9.4 Native text input imitation	48
9.5 Approaches for imitating native components	49
9.6 Implementation	51
10 Software design	52
10.1 Implementation as pure library	52
10.2 API	53
10.3 Distribution	56

<i>CONTENTS</i>	6
11 Architecture	57
11.1 Model-view-controller	57
11.2 Modular and object-oriented programming	58
IV Implementation	60
12 Implementation	61
12.1 Overview XX	61
12.2 Technology	61
12.3 Base class	63
12.4 Api	66
12.5 Input flow	67
12.6 Input reading	68
12.7 Input Pipeline	72
12.8 Pasting	75
12.9 Caret	77
12.10 Selection	78
12.11 Contents	79
12.12 Undo Manager	83
12.13 Events	84
12.14 Utility classes	87
12.15 Cache	90
12.16 Real-time collaboration with Etherpad	91
12.17 Extending	95
V Evaluation	98
12.18 Mobile Support	99
12.19 Development / Meta	99
12.20 Outlook	99
List of Figures	100
Listings	101

<i>CONTENTS</i>	6
Bibliography	102
VIAppendix	104

Chapter 1

Introduction

1.1 Motivation

Detlev Rich text ist sauwichtig (geschichte, lulturgut). So was wie google docs ist im moment NICHT MÖGLICH. Wegen contenteditable. Ich ändere das.

AUSSERDEM KEIN EDITOR SONDER EINE LIBRARY ganz tolles feature und wichtiger Punkt

hier auch irgendwo sagen dass ich ne library schreibe ****und sie type nenne****

Rich-text editors are commonly used by many on a daily basis. Often, this happens knowingly, for instance in an office suite, when users willfully format text. But often, rich-text editors are being used without notice. For instance when writing e-mails, entering a URL inserts a link automatically in many popular e-mail-applications. Also, many applications, like note-taking apps, offer rich-text capabilities that go unnoticed. Many users do not know the difference between rich-text and plain-text writing. Rich-text editing has become a de-facto standard, that to many users is *just there*. Even many developers do not realize 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 behavior. 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 ;).[sop,]"

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

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

However, calling commands will only manipulate the document's DOM tree, in order to format the text. This can also be achieved without using editing APIs, effectively avoiding unfixable bugs and enabling a consistent behavior 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 is rarely written about in books and papers and no one 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. Überhaupt, das questioning sollte im kern stehen. SCHREIBEN DASS AUCH ANDERE WIE MEDIUM DARÜBER RAGEN, SOWIESO MEHR REFERENZEN FINDEN

detlev Warum textverarbeitung im Web? - 1 applikation auf allen systemen anstatt 1000 programme und versionen, weniger wartbar aufwändiger - weltweit verfügbar - gebackuppt durch hosting und cloud - und wirtschaftlich zentral verwetbar. - probleme der privatsphäre sind nicht teile dieser arbeit.

1.2 Terminology

In the web-development world, the term *WYSIWYG* editor is commonly used. *WYSIWYG* is an abbreviation for **What You See Is What You Get** and describes a text editor's capability to display formatted text as it is being edited. This stands out to plain-text editors that can neither display nor edit formattings. The term rich-text editor has often been used for this feature and is more precise. For this reason, the term *rich-text editor* and *rich-text editing* will be used in this thesis.

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.

Part I

Theory

Chapter 2

Principles of rich-text editing

2.1 Overview

Rich text editors have been developed from HISTORY to NOW. They made their way from typewriters to desktop apps to the web. Nowadays even office suits can be entirely web-based. Google made it popular, even the dominant Microsoft Office is offering browser based rich-text editing.

2.2 Formatting

2.3 Behavior

Chapter 3

Text editing in desktop environments

3.1 Basics of plain-text editing

caret selection input

3.2 Basics of rich-text editing

document tree formatting algorithms

3.3 Libraries for desktop environments

It is no longer needed to implement basic rich-text editing components from the ground up. Rich-text editing has become a standard and most modern Frameworks, system APIs or GUI libraries come with built-in capabilities. Table 3.1 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 3.1: Rich-text components in desktop environments

Chapter 4

Text editing in browser environments

4.1 Overview

Developing a text editor in a browser environment differs from implementing a text editor in a desktop environment. Any implementation is based on the features of the browsers. Development is generally restricted to the components and APIs offered by the HTML5 standard as well as experimental features that are usually implemented in a subset of browsers¹. The boundaries of these restrictions can be overcome. It is common practice to combine native elements and APIs in ways they have not been designed for to enable features that are not offered. These techniques are often referred to as "hacks" and, despite this terminology, are generally not regarded as a bad practice.

This chapter will discuss the basics of plain-text and rich-text editing in browsers as well as the APIs and techniques that are natively offered.

4.2 Plain-text editing

Text input components for browsers have been introduced with the specification of HTML 2.0². The components proposed include inputs for sin-

¹Whether a specific component or API can be used or not, is determined by the intended audience and browser market share.

²<https://tools.ietf.org/html/rfc1866>, 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 4.1: Editing API attributes

gle line (written as `<input type="text" />`) and multiline texts (written as `<textarea></textarea>`). These inputs allow writing plain-text only.

4.3 Rich-text editing

Major browsers, i.e. any browser with a market share above 0.5%^[ag,], 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) a website to enable rich-text editing in the browser. The techniques used will be discussed in section 4.4 through section 4.5.

4.4 HTML Editing APIs

In July 2000, with the release of Internet Explorer 5.5, Microsoft introduced the IDL attributes³ `contentEditable` and `designMode` along with the content attribute `contenteditable`^{4,5}. These attributes were neither part of the W3C's HTML 4.01 specifications⁶ nor the ISO/IEC 15445:2000⁷, the defining standards of that time. Table 4.1 lists these attributes and possible values.

1 `<div contenteditable="true">`

³IDL attributes can only be set to DOM objects via JavaScript, whereas content attributes can be set to tags in the HTML source code. See <http://www.w3.org/TR/WebIDL/>, last checked on 08/17/2015

⁴[https://msdn.microsoft.com/en-us/library/ms533720\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/ms533720(v=vs.85).aspx), last checked on 07/10/2015

⁵[https://msdn.microsoft.com/en-us/library/ms537837\(VS.85\).aspx](https://msdn.microsoft.com/en-us/library/ms537837(VS.85).aspx), last checked on 07/10/2015

⁶<http://www.w3.org/TR/html401/>, last checked on 07/14/2015

⁷http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=27688, last checked on 07/14/2015

```
2   This text can be edited by the user.  
3 </div>
```

Listing 4.1: 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`-attribute 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 documentation⁸. These elements include "divs", "paragraphs" and the document's "body" element amongst others. Other than that, there is no difference in these attributes. In editing mode

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

When an element is switched to editing mode, the browser handles setting the caret if a user clicks inside the text, accepting keyboard input and modifying text nodes entirely by itself. No further scripting is necessary.

The API enabled by the editing mode must be called globally on the `document` object, but will only execute when the user's selection or caret is contained within an element in editing mode. Table ?? lists the full HTML editing API. To format text, the method `document.execCommand` must be used.

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

Listing 4.2: Emphasizing text using the HTML editing API

Listing 4.2 demonstrates an example call of the "italic" command. Calling this at any time on the `document` object, the browser will wrap the currently selected text (if inside an element in editing mode) with `<i>` tags. The method accepts three parameters.

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

Order	Parameter	Description
1	cmdID	The name of the command that will be executed
2	showUI	Determines if the browser will display a dialog if needed
2	value	A parameter that can be passed to the command invoked with the cmdID

Table 4.2: execCommand parameters

The first parameter is the "Command Identifier", which determines which command to execute. 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://example.com
  /');
```

Listing 4.3: Creating a link using the HTML editing API

The *third* parameter will be passed on to the internal command⁹ as a parameter. In the case of a `createLink` command, the third parameter is the URL to be used for the link to create. The *second* parameter determines if executing a command should display a user interface specific to the command. Using the `createLink` command with the second parameter set to `true` while not passing a third parameter, the user will be prompted with a system dialog to enter a URL. Most commands (command identifiers) `execCommand` accepts trigger text formatting. This includes commands to format text as bold, underlined, struck-through or as a headline. A full list of possible command identifiers can be found on MSDN¹⁰. Apart from executing commands, the API enabled by the editing mode includes the functions `queryCommandEnabled`, `queryCommandIndeterm`, `queryCommandState`, `queryCommandSupported` and `queryCommandValue` which allow reading attributes related to the editing mode.

4.5 Usage of HTML Editing APIs for rich-text editors

Most web-based rich-text editors use HTML editing APIs as their basis. The popular editors "CKEditor" and "TinyMCE" dynamically create an `iframe` on

⁹The command invoked using the command identifier

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

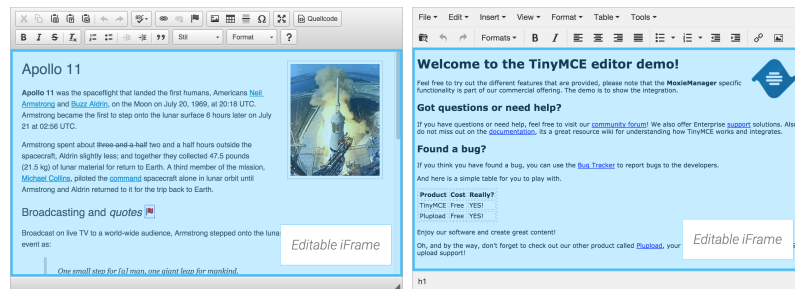


Figure 4.1: Usage of HTML editing APIs in CKEditor and TinyMCE

instantiation and set its `body` to editing mode using the `contenteditable`-attribute. This way, users can type inside the `iframe` which acts as a text input field. Both libraries wrap the `iframe` in a user interface with buttons to format the `iframe`'s contents. Using the interface, the commands of `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` element instead. The user interfaces vary between different editors.

Usually, rich-text editors implemented this way wrap their editing capabilities, including `document.execCommand`, in an API to enrich functionality and provide higher-level concepts. As discussed in **Part II: Discussion**, using HTML editing APIs requires a lot of workarounds, which some editors account for in the implementation of their library. Rich-text editing libraries can be downloaded JavaScript files and included in a web project. To display an editor on a website, it is common to select a `textarea` element on the website, that the library will replace with the rich-text editor. To integrate the editor into web forms, most libraries will mirror their contents to the selected `textarea`, so they can be submitted to a server.

For years, the market of web-based rich-text editors has been dominated by "CKEditor" and "TinyMCE". Both editors remain among the most popular choices. More recently, the number of editors increased drastically. Popular choices on GitHub, rated by the number of "stars", include "MediumEditor", "wysihtml" and "Summernote". As Piotrek Koszuliński points out, most editors "really doesn't[sic] work"[?] for the reasons discussed in **7.3: Disadvantages of HTML Editing APIs**.

Part II

Discussion

Chapter 5

Overview

HTML editing APIs are the standard and the recommended way by the W3C and the WHATWG for implementing rich-text editors on the web. However, its implementations across major web browsers are inconsistent, known to contain numerous bugs and have a limited and imprecise API.

Understanding the origins and the history of rich-text editing on the web poses the question if the paradigms it is based on have been thoroughly reviewed and if alternative ways for an implementation, possibly using hacks, should be considered.

Chapter **6: History HTML editing APIs** will discuss the history and origins of HTML editing APIs. Chapter **7: Advantages and disadvantages** will discuss its advantages and disadvantages and chapter **8: Rich-text editing without editing APIs** will discuss possible alternatives.

Chapter 6

History HTML editing APIs

6.1 Browser support

As discussed in **4.4: HTML Editing APIs** HTML editing APIs have been introduced in July 2000 with the release of Internet Explorer 5.5 by Microsoft and have not been part of any standard of that time.

With the introduction of editing capabilities, Microsoft released a short documentation¹, 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 WHATWG, states that the API's first use case has been for rich-text editing².

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)³. In June 2008, Mozilla added support for the `contentEditable` IDL and `contenteditable` content attributes in Firefox 3.

Mozilla's editing API mostly resembles the API implemented for Internet

¹[https://msdn.microsoft.com/en-us/library/ms537837\(VS.85\).aspx](https://msdn.microsoft.com/en-us/library/ms537837(VS.85).aspx), last checked on 07/10/2015

²<https://blog.whatwg.org/the-road-to-html-5-contenteditable>, last checked on 07/10/2015

³https://developer.mozilla.org/en/docs/Rich-Text_Editing_in_Mozilla, last checked on 07/10/2015

Explorer, however, to this present day, there are still differences (compare[ad,][am,]) concerning the available command identifiers[am,][ad,] as well as the markup generated by invoking commands[ai,].

In June 2006, Opera Software released Opera 9[ap,], providing full support for `contentEditable` and `designMode`[aq,], followed by Apple in March 2008[ar,] providing full support Safari 3.1[can,]. MDN lists full support in Google Chrome since version 4[as,], released in January 2010[at,].

6.2 Emergence of HTML editing JavaScript libraries

Around the year 2003⁴ 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"⁵ 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 Drupal⁶.

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 editing⁷. However, it should be noted, that some projects only use other projects' editors and some projects are stubs. These libraries are usually based on HTML editing APIs.

⁴compare *Meine Tabelle aller Editoren*

⁵Now distributed as "CKEditor"

⁶https://www.drupal.org/project/project_module, last checked on 07/16/2015

⁷<https://github.com/search?o=desc&q=wysiwyg&s=stars&type=Repositories&utf8=%E2%9C%93>, last checked on 07/16/2015

6.3 Standardization of HTML Editing APIs

HTML editing APIs, although not standardized, have been the de-facto standard for implementing rich-text editors on the web. Microsoft's demos, that have been published with the release of these APIs, suggested this application. Mozilla mimicked this API 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 possible to offer a native user interface component, a dedicated rich-text input field, instead of this API.

HTML editing APIs have only been standardized with HTML5, which itself introduces 13 new types of input fields⁸, but none with rich-text capabilities. 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 as follows:

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 addresses these main issues in a message from November 2004⁹. He states

Realistically, I just can't see something of this scoped[sic] [the ability to specify a language for a rich-text input 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

*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*

⁸<https://developer.mozilla.org/en/docs/Web/HTML/Element/Input>, last checked on 07/16/2015

⁹<https://lists.w3.org/Archives/Public/public-whatwg-archive/2004Nov/0014.html>, last checked on 07/16/2015

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 a milestone of the decision to integrate Microsoft's HTML editing APIs in the standard of the WHATWG.¹⁰ In cooperation with the W3C, the work by the WHATWG, including the standardization of the editing APIs, have been incorporated in the HTML5 standard and released in October 2014. When the cooperation ended in Juli 2012¹¹, the WHATWG kept this work, including the HTML editing APIs in the HTML Living Standard.

¹⁰<https://blog.whatwg.org/the-road-to-html-5-contenteditable>, last checked on 07/16/2015

¹¹todo confirm: <http://lists.w3.org/Archives/Public/public-whatwg-archive/2012Jul/0119.html>

Chapter 7

Advantages and disadvantages

7.1 Discussion

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 specifically well. In fact, all major browsers mimicked the API as implemented in Internet Explorer 5.5, even though there was no specification for it. The reasons for this have not been publicly 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". Mozilla adopted Microsoft's API applying practically no change to it. It can be argued that this has been part of the struggle for market shares while competing with Microsoft's Internet Explorer. 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 in getting users to choose Mozilla.

As discussed in section **6.1: Browser support**, 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 this decision. As described in section **6.3: Standardization of HTML Editing APIs**, it has clearly been stated, that the reason for standardizing these APIs for rich-text editing has been to ensure browser support.

The API itself stems from a time when the usage of the web was different from today. JavaScript has only been standardized 3 years before HTML editing APIs have been published. The use cases and products build with the technology are far more complex and elaborate than of this stage of the internet. The requirements of blogging platforms or products like Google's document editor were yet unknown.

The API itself and especially its implementations across various browsers has been criticized by Google[?], Medium[?], CKSource[?]¹ and others and has led websites to exclude users from editing in certain browsers entirely[citation needed]. Sections **7.2: Advantages of HTML Editing APIs** through **7.4: Treating HTML editing API related issues** discuss the advantages and disadvantages, as well as practices for treating the disadvantages of HTML editing APIs.

7.2 Advantages of HTML Editing APIs

Browser support

A fair reason for using HTML editing APIs is their wide browser support. caniuse.com lists that 92.78% of all web users use a browser that fully supports HTML editing APIs[can,].

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.

¹The creators of CKEditor

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.

Possible third-party solutions for other languages

While HTML editing APIs can be used to generate HTML only, its design offers a way for third-party libraries to build on top of that and implement editors that write "BB" code (for instance) and use HTML only for displaying it as rich-text. A dedicated rich-text input might not offer this flexibility.

7.3 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 implementations of all major browsers. Calling the `italic` command Internet Explorer, Firefox and Chrome all generate different markup.

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

Listing 7.1: Markup of italic command in Internet Explorer

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

Listing 7.2: Markup of italic command in Firefox

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

Listing 7.3: Markup of italic command in Chrome

This is a *major* problem for web development, because it makes processing input very difficult. For a content management system or a blogging platform it can be very hard to handle the input of users only because different browsers

are being used. 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 cases² while Firefox's output is breaking semantics entirely and is considered a bad style in terms of the separation of concerns of HTML and CSS³.

Different browsers will not only generate different markup when executing commands. When a user enters a line break (by pressing enter), Firefox will insert a `
` tag, Chrome and Safari will insert a `<div>` tag and Internet Explorer will insert a `<p>` tag. Most features of the HTML editing APIs that generate markup show different implementations across different browsers.

Flawed API

The original and mostly unaltered API is limited and not very effective. MDN lists 44 commands available for their `execCommand` implementation⁴. While other browsers do not match these commands exactly, their command lists are mostly similar. 23 of these commands format the text (for instance to italicize or make text bold) by wrapping the current selection with tags like `` or ``. 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 (`Lorem ipsum`). All 23 commands could be summarized by a single command that allows to pass custom tags or markup that the selected text will be wrapped with it. This applies to inserting elements as well. 7 commands insert different kinds of HTML elements, this could be simplified and extended by allowing to insert any kind of (valid) markup 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

²<https://developer.mozilla.org/en-US/docs/Web/HTML/Element/i#Notes>, last checked on 07/17/2015

³https://en.wikipedia.org/wiki/Separation_of_concerns#HTML.2C_CSS.2C_JavaScript, last checked on 07/17/2015

⁴<https://developer.mozilla.org/en-US/docs/Web/API/document/execCommand#Commands>, last checked on 07/17/2015

format with specific HTML would generate consistent markup (in the scope of a website) and would allow developers generate the markup that fits their needs.

Restrictions

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 editor⁵. 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 workarounds 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, thus limiting the features of an editor. Google names layouting the editor's contents with tab stops as one example.

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 aesthetic 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 simply by not offering input controls for these formatings (assuming no malicious behavior by the user). However contents can be pasted from the clipboard that contain any kind of formatting into ele-

⁵<http://googledrive.blogspot.fr/2010/05/whats-different-about-new-google-docs.html>, last checked on 07/18/2015

ments in editing mode. HTML editing APIs provide no way to define or apply filtering to the formattings of pasted contents.

Recent versions of major browsers allow observing paste events. Chrome, Safari, Firefox and Opera grant full read access to the clipboard contents from paste events. In these browsers, the event can be stopped and its contents can be processed. Internet Explorer grants 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. Overall, 82.78% of internet users support listening to and reading from clipboard events⁶.

When dealing with the clipboard, especially older browsers show an unexpected behavior. Older WebKit-based browsers insert so-called "Apple style spans"⁷ on copy and paste commands. "Apple style spans" are pieces of markup that have no visible representation, 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 contents of the editor.

Bugs

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

"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 [...] are working on this for years and we still have full bugs lists[sop, /]"

Mozilla lists 1060 active issues related to its "Editor" component[bi,]. Google lists 420 active issues related to "Cr-Blink-Editing"[bh,]. The WebKit project lists 641 active issues related to "HTML Editing"[bg,]. Microsoft and Opera Software allow public access to their bug trackers. As quoted above, some

⁶<http://caniuse.com/#feat=clipboard>, last checked on 07/18/2015

⁷<https://www.webkit.org/blog/1737/apple-style-span-is-gone/>, last checked on 07/18/2015

rich-text editors like CKEditor have been developed for over 10 years and still need to fix bugs related to the editing API[bf,][?]. Some bugs have caused big websites to block particular browsers entirely[med,].

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

7.4 Treating HTML editing API related issues

Since the issues arising with HTML editing APIs are part of the browser's implementation, they cannot be fixed by JavaScript developers. The common approach for most rich-text editors is to use HTML editing APIs and wrap it in a library while using workarounds for its issues and bugs internally. It is to be noted, as Piotrek Koszuliński points out, that the majority of rich-text editors "really do not work"⁸. This is usually the case when the problems discussed in **7.3: Disadvantages of HTML Editing APIs** have not been addressed and the library solely consists of a user interface wrapping an element in editing mode.

Having to account for multiple browser implementations, working around bugs can result in a big file size and a complex architecture. Most edge cases can only be learned from experience, not be foreseen or analyzed by debugging source code.

In practice, there are a few attempts to implement pure wrappers that will take care of the beforementioned issues, to support other developers with a working api. This approach is generally not well-adopted though. In general, most libraries are distributed as independent editors implementing their own solutions for addressing these issues—or are forks of other editors implemented with a different user interface.

⁸<http://stackoverflow.com/questions/10162540/contenteditable-div-vs-iframe-in-making-a-rich-text-w-11479435#11479435>

Subsections **7.4: HTML output** through **7.4: Not extendable** will discuss some approaches to treat the beforementioned issues.

HTML output

Editors like CKEditor offer some configuration on the generated HTML output⁹, but in the case of CKEditor this is very limited. The underlying issue is that HTML editing APIs cannot be configured. The only way to work around this issue is to implement custom methods to apply formatting in JavaScript and not using the `execCommand` interface¹⁰. The proprietary "Redactor Text Editor" demonstrates such an implementation.

Medium.com takes a different approach and implements an extensive framework that will compare the markup of the editor with a model of the visual representation that the markup generates and corrects the DOM on each change¹¹ to conform a defined norm.

Flawed API

HTML editing APIs are usually wrapped in the API of an editor, that offers more functionality than the original API. `execCommand` offers the `insertHTML` command that allows inserting custom elements. As discussed in the previous paragraph, extending the formatting capabilities requires a JavaScript implementation.

Clipboard

For browsers, that do not offer native support to control and process the contents pasted from the clipboard, workarounds must be used. There are two approaches to this

1. Sanitize the editor's contents after a paste event.
2. Proxy a paste event to insert its contents into another element and read the contents from it.

⁹http://docs.ckeditor.com/#!/guide/dev_output_format-section-adjusting-output-formatting-through-co

¹⁰HTML editing APIs can still be used for text input and other functionality

¹¹<https://medium.com/medium-eng/why-contenteditable-is-terrible-122d8a40e480>

The "Redactor Text Editor" uses the first approach. While reading contents from the a paste event is not fully supported, the event itself will be triggered by all major browsers, even most older versions¹². Once the event has finished and the contents have been inserted to the editor, a "cleaned up" procedure can remove unwanted contents.

CKEditor and TinyMCE have been developed before most major browsers supported clipboard events. Both editors implement a technique to permit pasting formatted text, that has been the standard for many years. CKEditor and TinyMCE create a hidden `textarea` element and listen for common "paste" keyboard shortcuts (`ctrl` `v` and `shift` `ins`). When a user presses these keys, the hidden `textarea` will be focused and thereby be the target in which the browser will paste the clipboard's contents. After a short delay, the editors can read the `textarea`'s contents. Since `textarea` elements allow plain text only, the contents will be removed of any formatting and can then be inserted to the editor. However, this does not account for pasting from the context menu. For this CKEditor overrides the native context menu with a custom menu containing a custom "paste" menu item, that will open a modal instructing the user to paste his or her contents using the keyboard shortcuts. TinyMCE overrides the native context menu too, but does not display a paste option. Up to the current versions CKEditor¹³ and TinyMCE¹⁴, this is still the case.

CodeMirror, a web-based source code editor enhances this approach by moving the `textarea` to the cursor's position when the user presses his or her right mouse button. This way a native context menu can be displayed while the paste option would insert the clipboard's contents to a designated `textarea`, that can be read from.

On the downside, the paste event cannot be proxied to the `textarea` if the user uses the browser's menu bar to paste contents.

Bugs

Generally, bugs cannot be fixed. The only way to treat bugs in browsers is by avoiding them, shimming them with JavaScript methods or "cleaning up"

¹²That have a market share BETTER CHECK THIS AGAIN

¹³CKEditor 4.5.1

¹⁴TinyMCE 4.2.3

after they have occurred.

Not extendable

The restrictions the HTML editing API imposed on the contents of the editor is an even bigger problem. Taking the example of layouting with tab stops¹⁵, the only solution is not making the entire contents of the editor editable, but implementing a layouting engine in JavaScript and enabling the editing mode only on parts of the layout.

¹⁵<http://googledrive.blogspot.fr/2010/05/whats-different-about-new-google-docs.html>

Chapter 8

Rich-text editing without editing APIs

8.1 Alternatives to HTML editing APIs

Overview

HTML editing APIs are the recommended way for implementing a web-based rich-text editor. This section will discuss possible alternatives to editing rich-text.

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

website¹. 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 re-rendered, 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 would require implementing a text-laying engine. The `canvas` API is not capable of laying text. On the other hand, making the editor accessible to other developers would be much more complex since the text only exists in an internal representation inside the editor and would not be exposed as DOM component to other developers.

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 accessed 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 layout.

Furthermore, while both alternatives can display rich-text, neither provides an dedicated API to manipulate rich-text, which gives neither alternative an advantage over using regular DOM structures to display rich-text.

Third-party plugins

Another 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.

Manipulation via the DOM APIs

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

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

DOM to display their rich-text contents too. Only the editing (of the DOM), commonly phrased "DOM manipulation", is implemented with HTML editing APIs.

Manipulating the DOM has been possible since the first implementations of JavaScript and JScript. It has been standardized in 1998 with the W3C's "Document Object Model (Core) Level 1" specification² as part of the "Document Object Model (DOM) Level 1 Specification".

Other than for rich-text editing, the DOM and its API is the recommended³ way to change a website's contents and—apart from HTML editing APIs—the only option *natively* implemented in any major browser. Popular libraries like jQuery, React or AngularJS are based on it. The API has been developed for 17 years and proven to be stable across browsers.

MDN lists 44 commands for the `execCommand` interface⁴.

- 23 commands apply text formatting.
- 6 commands insert HTML elements.
- 2 commands remove contents.
- 2 commands remove formatting.
- The other commands enable control over the clipboard, implement undo/redo commands, set settings for the editing mode and one command can select all text of the editable element.

Algorithms 1 through 4 demonstrate alternatives to commands of the `execCommand` interface related to text formatting, insertion and deletion implemented with methods of the "Document Object Model (Core) Level 1" specification.

Algorithm 1 demonstrates a simplified procedure to wrap a text selection in a tag. To implement the `bold` command of `execCommand`, this procedure

²<http://www.w3.org/TR/REC-DOM-Level-1/level-one-core.html>, last checked on 07/10/2015

³recommended by the W3C and WHATWG

⁴<https://developer.mozilla.org/en-US/docs/Web/API/Document/execCommand>

Algorithm 1 Simplified text formatting pseudocode

```

1: procedure FORMAT
2:    $s \leftarrow$  split text node at beginning of text
3:    $e \leftarrow$  split text node at end of text
4:    $t \leftarrow$  new tag before  $s$ 
5:   for all  $n$  in selection do                                 $\triangleright n$  is a node in the selection
6:     Move  $n$  to  $t$ 
7:   end for
8: end procedure

```

can be implemented using the **strong** tag. The text selection can be read with the browser's selection API^{5,6}.

Algorithm 2 Simplified element insertion pseudocode

```

1: procedure INSERT
2:   if Selection is not collapsed then
3:      $s \leftarrow$  split text node at beginning of text
4:      $e \leftarrow$  split text node at end of text
5:     for all  $n$  in selection do                                 $\triangleright n$  is a node in the selection
6:       Remove  $n$ 
7:     end for
8:     Collapse selection
9:   end if
10:  Insert new tag at beginning of selection
11: end procedure

```

Algorithm 2 demonstrates a simplified procedure to insert a new tag and possibly overwrite the current text selection and thereby mimicking `execCommand`'s insertion commands.

Algorithm 3 demonstrates a procedure to mimic the deletion commands of `execCommand`.

Algorithm 4 demonstrates a procedure to unwrap an element, mimicking the commands of `execCommand` to remove formatting.

With formatting and removing text as well as inserting and unwrapping elements, we can find equivalents for all commands of the HTML editing APIs

⁵<https://developer.mozilla.org/en-US/docs/Web/API/Selection>, last checked on 07/18/2015

⁶Internet Explorer prior version 9 uses a non-standard API [https://msdn.microsoft.com/en-us/library/ms535869\(v=VS.85\).aspx](https://msdn.microsoft.com/en-us/library/ms535869(v=VS.85).aspx), last checked on 07/18/2015

Algorithm 3 Simplified text removal pseudocode

```

1: procedure REMOVE
2:   if Selection is not collapsed then
3:      $s \leftarrow$  split text node at beginning of text
4:      $e \leftarrow$  split text node at end of text
5:     for all  $n$  in selection do            $\triangleright n$  is a node in the selection
6:       Remove  $n$ 
7:     end for
8:     Collapse selection
9:   else
10:    Remove one character left of the beginning of the selection
11:   end if
12: end procedure

```

Algorithm 4 Simplified element unwrapping pseudocode

```

1: procedure UNWRAP( $e$ )            $\triangleright e$  is an element
2:   for all  $n$  in  $e$  do            $\triangleright n$  is a node in  $e$ 
3:     Move  $n$  before  $e$ 
4:   end for
5: end procedure
6: Remove element

```

related to manipulating rich-text using only methods specified by the "Document Object Model (Core) Level 1". This shows, that HTML editing APIs are not a necessity for rich-text editing. Chapter **12: Implementation** demonstrates ways to implement clipboard, undo/redo and selection capabilities.

8.2 Rich-text without HTML editing APIs in practice

Google completely rewrote their document editor in 2010 abandoning HTML editing APIs entirely. In a blog post⁷, they stated some of the reasons discussed in section **7.3: Disadvantages of HTML Editing APIs**. 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, abandoning HTML editing APIs enables

⁷<http://googledrive.blogspot.fr/2010/05/whats-different-about-new-google-docs.html>, last checked on 07/18/2015

features otherwise impossible, for example tab stops for layouting[?]. With the Google document editor, Google demonstrates it is possible to implement a fully featured rich-text editor using only JavaScript without HTML editing APIs.

Google’s document editor is proprietary software and its implementation has not been documented publicly. Most rich-text editors still rely on HTML editing APIs. The editor “Firepad”⁸ is another exception. It is based on “CodeMirror”, a web-based source code editor, and extends it with rich-text formatting. The major disadvantage of Firepad is its origin as a source code editor. It generates “messy” (non-semantic) markup with lots of control tags. It has a sparse API that is not designed for rich-text editing and has no public methods to format the text. It is to be noted that Google’s document editor generates lots of control tags as well, but it is only used within Google’s portfolio of office apps where it may not be necessary to create *well-formatted*, semantic markup. A full list of rich-text editors using and not using HTML editing APIs can be found in tables ?? and ??.

8.3 Advantages of rich-text editing without editing APIs

With a pure JavaScript implementation, many of the problems that HTML editing APIs have, can be solved. The issues discussed in **7.3: Disadvantages of HTML Editing APIs** will be addressed hereinafter.

Generated output and flawed API

The generated markup, if implemented through JavaScript and DOM Level 1 methods, can be chosen with the implementation of the editor. Furthermore, the decision of the generated output can be given to the developers working with the editor. Section **7.3: Disadvantages of HTML Editing APIs** describes the inconsistent output across various browsers as well as the restrictions of the API design of `execCommand`. Both issues can be addressed by offering a method to wrap the current selection in arbitrary markup. jQuery’s

⁸<http://www.firepad.io/>, last checked 07/23/2015

`htmlString` implementation⁹ demonstrates a simple and stable way to define markup as a string and pass it as an argument to JavaScript methods. A sample call could read as follows.

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

Listing 8.1: Example calls to format text

This will allow developers to choose which markup should be generated for italicizing text. The markup will be consistent in the scope of their project. Since the DOM manipulation is implemented in JavaScript and not by high-level browser methods, this will also ensure the same output across all systems and solve cross-browser issues. The second example function call in listing 8.1 demonstrates that custom formatting, fitting the needs of a specific project, can be achieved with the same API, giving developers a wider functionality.

8.4: Native components discusses the disadvantage, that when not using HTML editing APIs, native components like the caret or the text input must be implemented with JavaScript as they are not provided without using HTML editing APIs. On the flip side, this allows full control over these components that can be exposed via an API to other developers.

Restrictions

When implementing an editor in pure JavaScript, the limitations imposed by the HTML editing APIs, 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 document editor demonstrates rich functionality that would not be possible with an implementation based on HTML editing APIs.

⁹<http://api.jquery.com/Types/#htmlString>, last checked on 07/19/2015

Clipboard

Without a native text input or an element switched to editing mode with HTML editing APIs, clipboard functionality is not available. Users cannot paste contents from the clipboard unless one of these elements is focused. However chapter **12: Implementation** demonstrates a way that not only allows clipboard support, but also grants full control over the pasted contents.

Bugs

By refraining from using HTML editing APIs, all of its numerous bugs will be avoided. An the implementation can be aimed to minimize interaction with browser APIs, especially unstable or experimental interfaces. DOM manipulation APIs have been standardized for more than 15 years and tend to be well-proven and stable. Bugs that occur will mostly be part of the library and can be fixed and not only worked around. Bug fixes can be rolled out to users when they are fixed. This will free development from being dependent on browser development, update cycles and user adoption.

8.4 Disadvantages of rich-text editing without editing APIs

Formatting

The HTML editing APIs' formatting methods take away a crucial part of rich-text editing. Especially on the web, where a text may come from various sources, formatting must account for many edge cases. Nick Santos, author of Medium's rich-text editor states:

"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.[?]"

An editor implemented *without* HTML editing APIs does not only need to account for content (HTML) that will be pasted into the editor¹⁰ (in fact, content can be sanitized before it gets inserted in the editor, see **12.8: Pasting**), 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). In HTML, the same visual representation of a text, can have many different—and valid—underlying DOM representations. Nick Santos gives the example of the following text[?]:

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 8.2: 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 does not conform this rule.

Native components

As discussed in section **4.4: HTML Editing APIs**, when an element is switched to editing mode using the HTML editing APIs, the user can click inside the text and will be presented with a caret. He or she can move the

¹⁰Medium uses HTML editing APIs

caret with his or her arrow keys and enter text that will be inserted at the appropriate offset. He or she can use keyboard shortcuts and use the mouse's context menu to paste text. Behavior that is common for rich-text input, for instance that a new list item will be created when the user presses "enter" inside a list, is implemented by the browser. None of this is available when not using HTML editing APIs. All of this must be accounted for and implemented in JavaScript. Elements like the caret must be mimicked with DOM elements like the `div` element. The user's input must be read with JavaScript and either move the caret or modify the text.

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[br,]. 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 applications 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.

Part III

Concept

Chapter 9

Approaches for enabling rich-text editing

9.1 Overview

This section will discuss the options to implement rich-text editing without relying (entirely) on HTML editing APIs and approaches to avoid their disadvantages and bugs.

9.2 Native inputs, images and third-party plugins

As discussed in **8.1: Alternatives to HTML editing APIs**, native text inputs cannot be used for rich-text editing, using image elements has no benefits and many disadvantages and third-party plugins lack user adoption. For these reasons, none of these approaches will be considered.

9.3 Enabling editing mode without using its API

One way to enable editing but avoid many bugs and browser inconsistencies, is to enable the editing mode on an element, but refrain from 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 capabilities—all of this would be taken care of by the browser.

This 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 but the input would be handled by the browser's editing mode, the browser may not be able to work on the structures generated by JavaScript and break elements or simply get stuck. This was one of the reasons why Google decided to abandon editing APIs entirely¹. It could be the source to many bugs and ultimately restrict the editors capabilities.

9.4 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 and DOM Level 1 APIs. However, this does not suffice to provide the experience of a text input. The following components, common to text editing, must also be accounted for:

Caret

The caret is an essential part to text editing. 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 behavior depends on the operating system used).

Selection

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 rectangular text selection, independent of line breaks.

¹<http://googledrive.blogspot.fr/2010/05/whats-different-about-new-google-docs.html>, last checked on 07/21/2015

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 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.

Behavior

Rich-text editors (usually) share a certain behavior on user input. When writing a bulleted list, pressing the enter key usually creates another bullet 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 after heading.

9.5 Approaches for 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 behavior for the user's input that is common for rich-text editing.

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 9.1: 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 that the 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 9.1). 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 DOM elements like `divs` to display a text selection and a *hidden* `textarea` to fetch keyboard inputs, to recreate the behavior and capabilities 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. Incorporating feedback is a core philosophy of the WHATWG, the original creators of HTML5.

9.6 Implementation

The library implemented for this thesis uses similar techniques as Ace and CodeMirror to create a rich-text editor. The contents of the editor are represented by a `div` element that contains the formatted text using HTML. The caret as a `div` element styled to mimic a native caret. The text selection is also displayed by using `div` elements that are styled accordingly. The keyboard and mouse input will be read with JavaScript and the contents of the editor will be changed accordingly using DOM Level 1 methods. The user's input will also be read to move the mimicked caret on the website with JavaScript. The specifics on the implementation of these components and how they interact will be discussed in **12: Implementation**.

Chapter 10

Software design

10.1 Implementation as pure library

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 distributed 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 its specific purpose. For instance, a content management system may require an editor 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 that are functional, it can be argued, that choosing between editors is often really a choice of the desired user interface.

Customizing a user interface can be just as complex as writing an interface from scratch. The latter affords to add HTML elements and call JavaScript methods while both require styling. While adding HTML elements to a website is not a complex task, in a worst case scenario, it can be more complicated to customize an interface to specific needs than writing an interface from scratch and being able to define just the elements as they are required.

As for the current state of the internet, web developers cannot easily implement a rich-text for themselves, they have to make a choice between pre-made solutions and customize them. Apart from the perspective of the user interface, integrating a fully featured editor into a software project can be invasive to the structure of the project.

For these reasons the library of this thesis will be implemented and distributed as a pure software library, offering developers an API to create a rich-text editor, rather than a fully implemented rich-text editor as a user interface component.

10.2 API

The library should be capable of any method implemented by HTML editing APIs. However the API design can differ to improve the way it will be worked with. In particular the API aims at providing a quick and simple way to create editable areas and connecting a user interfaces to it.

API Design

The API of this library 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 requirement of the developers should be matched with as little effort as possible. The API should create a workflow for developers that allows them to do what they intend to do and is as easy to use and as plausible as possible. jQuery is an example of incorporating an API that comes close to these goals.

The library's API will have two basic use cases. On the one hand, web developers must be enabled to implement rich-text editors with it. On the other hand, the library should offer interfaces for enabling web developers to extend the library and add features.

Extension For extension, web developers should have precise access to as many components and functions of the library, providing as much freedom and

options as possible. This will include low-level access to components while control and explicitness is more important than simplicity.

All components of the library will be implemented as classes. To provide as much capabilities as possible to other developers, all classes of the library will be exposed in a designated namespace. The classes should conform the best practices of object-oriented programming to support developers in extending the library. The class design should not only consider the specific needs of the core library but also potential use cases for other developers.

For example, with a designated class to show and move a caret, multiple carets can be instantiated for an extension that allows real-time collaboration with multiple users. All available classes will be discussed in chapter **12: Implementation**.

Editor implementation For web developers implementing an editor, the API should be designed to offer methods for the most common tasks related to rich-text editing to allow fast and easy integration in a website. This should be high-level methods as compared to methods required for extending the library. Simplicity is more important than precise control over low-level behavior. For implementing a rich-text editor the exposed methods should cover

1. Formatting and removing formats
2. Insertion
3. Deletion
4. Controlling the caret
5. Controlling the text selection
6. Controlling the clipboard
7. Controlling settings
8. Undo / redo commands

jQuery demonstrates an effective and simple approach to API design, conforming the principles as discussed above. In jQuery all methods remain in a flat

hierarchy within the root of a jQuery collection. Any method that is not a getter allows chaining and most methods are overloaded to allow passing various kinds of parameters, to determine what the function should do. Following these and the above-mentioned principles, the components listed above can be expressed in 11 functions:

```
1 editor.caret([options]);
2 editor.selection([options]);
3 editor.insert([options]);
4 editor.format([options]);
5 editor.remove([options]);
6 editor.settings([options]);
7 editor.copy();
8 editor.cut();
9 editor.paste();
10 editor.undo();
11 editor.redo();
```

Listing 10.1: API for implementing a rich-text editor

The functions in lines 1 through 6 can take various overloaded parameters to determine the specific action. The selection command, for instance, can be called with two numbers to draw a selection from one character offset to another. To draw a selection from characters 10 to 20 `editor.selection(10, 20)` can be called. The function can also be called without passing parameters, to read the selection. `editor.selection()` will return the currently selected text as a string. A full API description can be found at table TODO.

Handling use cases

We can call programmers extending the library "developers of the library" and programmers using the library to implement editors "users of the library". To account for both use cases and maintain a clear software architecture as well as a separation of concerns, all classes that provide functionality to the library must remain in a designated namespace which the library has access to. Developers of the library have access to the namespace and can utilize any of its classes to extend its functions.



Figure 10.1: Diagram of the Type library and its internally used classes (excerpt)

The classes within the namespace will be used by a globally accessible class called "Type" which is the entry point for the users of the library to implement rich-text editors. The Type class provides an API with all of the above-mentioned methods and uses the classes inside the namespace for their implementation. It must be instantiated and be passed an element on the website (for example a `div` element) which it will then use as its "editor contents". The users of the library can build an interface for this editable element and use the instance's API to edit its rich-text contents.

10.3 Distribution

The library will be distributed as a single JavaScript file. Extensions by third-party developers will each be distributed as independent and separate (JavaScript) files. By exposing Type and its classes as discussed in section **10.2: API** they can be accessed from other files. This provides a modular "plug and play" system for distributing and loading extensions. To improve loading times, web developers can concatenate Type and its extensions to a single file in a web project.

Chapter 11

Architecture

11.1 Model–view–controller

Model–view–controller (MVC) is a common approach for implementing user interfaces and it can be applied to user interface components too. While this approach can provide clear responsibilities, the problem is that most components, like the caret or the selection, serve a clear atomic purpose and would need to be broken apart into model, view and controller parts themselves, making the architecture fuzzy and complex instead of simplifying it.

Following the MVC architecture, the contents of the editor (the text) can be represented in a model (holding the text data and allowing methods to be performed on) and be rendered with a view (displaying the text in the browser). In contrast to the beforementioned components, this would be a very clean model for implementing the editor’s contents. It is even imaginable to implement multiple renderers in the view layer, turning the editor from rich-text into a Markdown editor, for instance.

Unfortunately, this approach would make the contents of the editor only editable through the API of the ”Type” library. If any other script on a website would change its contents, the library’s renderer would overwrite the changes with the next rendering the data of the internal model. As discussed in **10.2: API**, the library shall leave as much freedom as possible to the developers. This would create a bottleneck and restrict other developers.

11.2 Modular and object-oriented programming

jQuery and CKEditor demonstrate a software architecture in which a base object, which is exposed to other developers as the library, provides an environment to extend its functionality, but does not offer many methods itself¹. The actual functionality of both libraries is implemented through extensions while the libraries are usually bundled with a set of "core extensions" that provide basic features. CKEditor makes use of modular programming techniques by implementing a major part of its editor as plugins that communicate via strictly defined interfaces. jQuery established a paradigm calling any extension a "plugin" but instead of using strictly defined interfaces, developers are encouraged to add arbitrary methods to jQuery's base object, which can then be directly accessed. Extending a base object has many advantages:

1. It provides a namespace for the library
2. It provides a structure for extensions to access each other
3. It approaches modular programming and strong decoupling

Strict modular programming could create a system in which other developers can exchange any component easily to improve performance or enrich functionality. The disadvantage this approach would be that the need for well-defined interfaces can diminish flexibility. Formalizing interfaces would create complex structures and could make it harder for other developers to contribute to the library instead of inviting them. jQuery uses another approach and encourages arbitrary extensions. jQuery's approach demonstrates that this flexibility, in practice, can withstand possible conflicts. In turn, the low barrier for extending jQuery has spawned a rich collection of extensions and a big community of developers. While jQuery technically allows to be extended with complex libraries, it is designed to be extended with simple methods. It is difficult to establish complex interactions between extensions.

¹CKEditor provides a framework for implementing components for it, but does not offer any rich-text functionality in its core. jQuery provides low-level utility methods for JavaScript.

Constructor Pattern & modularized structure

To close the gap between CKEditor’s modular programming approach and jQuery’s simple extension paradigms, object-oriented programming (OOP) can be used. JavaScript does not offer classes and classical inheritance, however the same functionality can be achieved using the constructor pattern and prototypal inheritance (see **12.14: OOP**). Functions following the constructor pattern are often called classes or pseudo-classes. Hereinafter the term classes will be used.

As discussed in **10.2: Handling use cases**, the base class will be globally accessible with the name "Type". It will provide the namespace for classes that extend the library and implement its functionality. A set of core extensions will provide all components needed for a rich-text editor. The "Type" base class can be instantiated and will be the entry point for *users* of the library (see **10.2: Handling use cases**) to implement a rich-text editor. Like CKEditor and jQuery, will implement as little functionality as possible itself. The implementation of the base class as well as the interaction of its extensions will be discussed in detail in chapter **12: Implementation**. Implementing the library’s extensions as classes has many benefits:

1. As compared to CKEditor and modular programming, strictly defined interfaces are not a necessity. This can improve flexibility and lower the barrier for other developers to contribute.
2. As compared to jQuery, classes can have complex interfaces, which allows rich functionality and possibilities in interaction.
3. Classes are a proven concept for encapsulating functionality and data, protecting access and structuring code as well as making it readable.
4. Through JavaScript’s prototypical inheritance, the class can be instantiated as often as desired, but will only be allocated once in the browser’s memory. Thereby the performance will be improved. Instance variables still allow to reuse a class in different contexts with different inherent data.

Part IV

Implementation

Chapter 12

Implementation

12.1 Overview XX

As discussed in **11.2: Constructor Pattern & modularized structure**, Type’s implementation relies on a base class that provides a high-level API for implementing rich-text editors. The library’s functionality is implemented through various other classes encapsulated in a designated namespace. Section **12.2: Technology** will discuss the tools used to develop and build the library. Section **12.3: Base class** will discuss the base class and the namespace it creates. Sections **12.5: Input flow** and following discuss the functionality, architecture and the classes involved in Type by explaining how the user’s input will be read, processed and written to the website.

12.2 Technology

Overview

There are no pre-made solutions or conventions suggesting how to design, structure and concatenate components for a JavaScript library. The most popular JavaScript libraries on GitHub¹ each implement custom and different solutions. Angular.js implements a custom module-system and uses the tool ”Grunt” to concatenate multiple files into one. D3.js uses a Makefile and var-

¹<https://github.com/search?l=JavaScript&q=stars%3A%3E1&s=stars&type=Repositories>

ious Node.js modules for building and concatenation. jQuery uses Grunt for concatenation and runs custom scripts implemented using Node.js to manipulate and clean up the resulting source code file. To support development, the library is split up into multiple files. One file each contains one class. The tools to concatenate, build and check the sources will be discussed in the following sections.

Gulp

Gulp is a Node.js-based task runner that is widely used as a build system for web applications. For this library, Gulp tasks are used to lint the source code, check code style rules and to concatenate and minify the library into a single file, that can be distributed to other developers.

Building

Each file is written as a CommonJS module that will be concatenated using RequireJS in a designated Gulp task to a single file. RequireJS generates code structures around each module so that all modules can access one another. This will increase the library's file size. AMDclean is used to remove as much of this supporting code as possible while maintaining the functionality. When all source code files have been concatenated and cleaned from RequireJS, it will be compressed using UglifyJS2. The resulting file has a low file size and can be distributed for productive use.

Linting

JSLint is used to lint the source code before any concatenation happens. It will also check if the code conforms the JavaScript code conventions introduced by Douglas Crockford. The code mostly conforms these conventions, differing in the way the constructor pattern is implemented to favor better readability. Classes using the constructor pattern are implemented using the conventions of the Ace library and use the prefix convention² for private members.

²https://developer.mozilla.org/en-US/Add-ons/SDK/Guides/Contributor_s_Guide/Private_Properties#Using_Prefixes

Code formatting

JSCS is used to check the code to conform specific formatting conventions—for instance the number of spaces used for indentation or a consistent use of CamelCase. Along with JSLint checking for Douglas Crockford’s conventions this ensures a consistent coding style across all files and classes.

ECMAScript 6

ECMAScript 6 offers many features for JavaScript including classes and modules. It is not supported by most browsers yet, however tools like “Babel” can transpile ECMAScript 6 to ECMAScript 5. ECMAScript 6 and Babel are used by popular websites like “Netflix” and JavaScript libraries like “React”. However, Type will not use this. Even though its features can support the development with syntactic sugar, most of its features can be approached using ECMAScript 5 design patterns. The benefits of *not* using ECMAScript 6 and transpilers like Babel are a smaller file size, better performance and independence from third-party tools.

12.3 Base class

Overview

The `Type` class is the base class (see **11.2: Modular and object-oriented programming**) is the starting point for users and developers of the library. It provides 4 purposes:

1. It can be instantiated to offer a high-level API to manipulate text and perform other rich-text related functions.
2. It provides a namespace for the library’s internal classes.
3. A `Type` instance provides mutual access for the instances of the internally used classes.
4. It exposes its `prototype` as a public shorthand attribute.

Instantiation and usage

To develop an editor with the library, the `Type` base class must be instantiated. As discussed in **10.2: Handling use cases**, all of the library's functionality is provided by this class. It exposes its features as high-level instance methods. The `Type` class is exposed to the `window` namespace and is thereby globally accessible. On instantiation it must be passed an `HTMLElement`, which's contents will act as the editor's rich-text content and all rich-text operations will be performed on it. Just like with HTML editing APIs, developers can build user interfaces around the editable element. An optional second parameter can be passed to the class to define settings for the editor's instance.

```
1 var element = document.getElementById("myElement");
2 var editor = new Type(element, { paste: "text" });
```

Listing 12.1: Type instantiation

The `editor` instance variable now offers methods to format, insert and remove text, manipulate the caret and the selection, dynamically change settings and control undo/redone capabilities as well as trigger clipboard commands. The complete API is listed in Figure ABC. For example, to format the characters 10 to 20 as bold and move the caret behind the formatted text, the following methods can be executed:

```
1 editor.selection(10, 20);
2 editor.format("<strong />");
3 editor.caret(20);
```

Listing 12.2: Example commands to format text

This is just an example to demonstrate the API. It should be noted that the API allows to specify a text range in the format command as well as chaining and the above code can be simplified to a single line.

```
1 editor.format("<strong />", 10, 20).caret(20);
```

Listing 12.3: Example chaining

With the second optional parameter, settings can be passed to the `Type` class on instantiation to determine the editor's behavior. As for the implementation of this thesis two settings are available: To determine the behavior on paste events and to turn off default keyboard shortcuts. A full description can be found in figure ABC. Passing options to the editor can be useful for extensions which can access any option passed by accessing the `Type` base class.

Namespace and references

The `Type` class creates a namespace for other classes used in the library. In JavaScript, functions are first-class objects, namespaces are objects and any object is a namespace. This way, the `Type Function` object (the class itself) can act as a valid namespace. CKEditor takes the same approach and attaches each module to the `CKEDITOR` base class. Any of the classes listed in ???: ?? are attached to the `Type` class this way. As an example, listing 12.4 shows the declaration of the `Type` base class as well as the declarations of the classes `Caret`, `Range` and `Environment`:

```
1 // Declaration of the Type base class
2 function Type() {};
3
4 // Classes defined within the namespace created by Type
5 Type.Caret = function () {};
6 Type.Range = function () {};
7 Type.Environment = function () {};
```

Listing 12.4: Declaration of `Caret`, `Range` and `Environment` classes

The namespace provides a structure for the classes of the library, prevents the pollution of the global namespace and possible name conflicts. In terms of structure, this does not only encapsulate classes related to the library but also allows nesting. This way sub-namespaces can be created, which is especially important for other developers extending the library. The `Type` base class already creates a sub-namespace `Type.Events` for events.

On instantiation, the `Type` class, in turn, will instantiate classes of its namespace that implement its rich-text editing functionality. The `Type` in-

stance will pass a reference to itself to all classes and offers getters for every class instance it created, to provide mutual access for each class of the library.

Exposal of `Type`'s prototype

While `Type`'s functionality is implemented through classes within its namespace, this does not expose its functionality to its instance-API. With the constructor pattern, all methods in the `prototype` of the `Type` class will be available as instance methods. In a simple approach, the library's API can be implemented directly in the implementation of the `Type` class. This contradicts the modular approach of the library. jQuery established an effective principle for extending its API. It exposes the library's `prototype` with a shorthand attribute as `jQuery.fn`. This way, other modules can extend the `prototype` easily and add methods to the library's API. `Type` follows the same principle and exposes its `prototype` as `Type.fn`. The `prototype` could also be accessed without exposing it with a shorthand attribute, but this is intended to clarify its purpose similar to jQuery and encourage developers to extend it.

12.4 Api

The methods of `Type` instances can be added by any of the classes of the library. However, to achieve a clear separation of concerns, maintainability and to conform the modularized structure, `Type`'s instance API is implemented through a designated module that adds all methods to the API.

The module for this, called "CoreApi", is the only exception in `Type`'s implementation—it extends `Type.fn` directly with all methods listed in XYZ, *without* using a class.

Similar to the principles of MVC in which the controllers should be kept small and the implementation should reside in the model layer, the functionality of the API resides in their designated classes, while the API methods are kept small. The classes are written in a way that allow the API methods to require not more than one function call for their implementation and at most translate their parameters for the class methods.

12.5 Input flow

To enable reading the user's input and writing it to the editor's contents, the classes `Input`, `Contents`, `Writer`, `Formatter`, `Caret` and `Selection` will be instantiated by the `Type` base class.



Figure 12.1: Components instantiated by the `Type` base class

The `Input` class will listen to keyboard input and mouse input. It is responsible for setting the caret and the selection using the `Caret` and `Selection` classes and uses them to determine which part of the text should be changed or formatted when the user enters text, uses keyboard shortcuts or uses his or her mouse or touch device. It passes formalized edit operations to the `Contents` class which will emit events for an `UndoManager` that enables undo and redo operations. The `Contents` class uses the `Writer` and `Formatter` instances to manipulate the visible text on the website. These classes perform the actual DOM operations on the contents of the element passed to `Type` on instantiation

(see **12.3: Instantiation and usage**).

Usually, text input fields contain one caret and display one text selection at a time. For this reason the `Type` base class instantiates the `Caret` and `Selection` classes for shared usage within an editor's instance. Of course, this behavior can be extended, for example by instantiating multiple `Carets` for real-time text collaboration.

12.6 Input reading

There are various input methods with which users can interact with native inputs. This includes using hardware devices as well as virtual (on screen) devices:

- Hardware keyboard input
- Virtual keyboard input
- Mouse (pointer) input
- Touch input
- Game controller input (on game consoles)
- Remote control input (on smart TVs)

When mimicking a native input, in a best-case scenario, all these input methods should be accounted for. Fetching input includes two scenarios: The user clicks, touches or focuses the input in any way and does so at any position inside the input. If the user points (touches, clicks, etc.) in the middle of the text, the caret should move to that position. In environments without hardware keyboards, the library must ensure that a virtual keyboards show up. Once the input is focused, text input must be fetched and written to the contents. There are various options to fetch user input, which will be discussed in the following sections.

Events

One way to fetch user input is by listening to events.

Keyboard Text input can be read through `KeyboardEvents`. Keyboard events will be triggered for virtual keyboards and for hardware keyboards. When the user presses a key, the event can be stopped and the according characters can be inserted at the offset of the caret. As a downside, listeners for keyboard events cannot be bound to an element that is not a native text input, that means keyboard events must be listened to on the `document` level. This does not only have (minor) performance downsides but also requires more logic to decide whether a keyboard input should be processed and ultimately stopped or ignored and allowed to bubble to other event listeners of a website. In particular, 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.

Mouse (pointer) and touch To support clicking or touching inside the editor's contents `MouseEvents` 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.

Remote controls Although some smart TVs offer keyboards, mice, pointers similar to Nintendo's Wii remote, input via smartphone apps and many other input devices, button-based remote controls are offered with almost any smart TV and remain an edge case for interacting with a text editor. In such an environment, users commonly switch between elements by selecting focusable elements with a directional pad. Only using events would not account for this since there would be no focusable element representing the editor. Recent browsers on Samsung's and LG's smart TVs are based on WebKit³ while Sony's TVs use Opera. Before 2012 Samsung's browser was based on Gecko. All of these browsers and browser engines support keyboard events triggered by virtual keyboards to fetch their input.

³<http://www.samsungdforum.com/Devtools/Sdkdownload>, last checked on 07/22/2015

Clipboard Another problem with relying entirely on events is the lack of native clipboard capabilities. Unless a native text input (including elements 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.

Hidden native input fields

As discussed in **9.5: Approaches for imitating native components**, the source code editors Ace and CodeMirror use a hidden (native) input field to fetch the user’s keyboard input. While it appears to the user he or she is entering text in a syntax-highlighted representation of the source code, in reality the user enters his or her text in a *hidden textarea* element. The input will be read from the `textarea`, processed and displayed with syntax-highlighting using HTML. This solves many problems that occur with relying solely on events:

- The hidden `textarea` can be focused with the tab key.
- The hidden `textarea` can be focused with remote controls.
- Virtual (on screen) keyboards will show up when the `textarea` is focused.
- Keyboard shortcuts for clipboard events work.
- It can display a native context menu that allows pasting.

Implementation

The `textarea` is created when the editor gets instantiated. Since browsers scroll the `textarea` into view when it receives the focus, it is positioned in the visual representation of the editor, scrolling the editor into view⁴. This perfectly mimics the browser’s native behavior. To maintain the illusion that the user actually writes inside the visual representation of the editor the `textarea` is hidden.

⁴This does not mean the editor will be scrolled into view on instantiation, but when the user focuses it, for example with the tab key.

Focus

Whenever the user clicks inside the editors visible contents, the mimicked caret will be moved (see **12.9: Caret**) to the according text position. To enable text input, the hidden `textarea` will be focused on click or touch events. The `textarea` is natively focusable using the tab key or a remote control on a smart TV. It will also trigger focus and blur events. This way, it is possible to display the caret when the `textarea` receives the focus and read its input as well as hiding the caret on blur and thereby perfectly mimic the native behavior for input events.

Virtual (on screen) keyboard support

The `textarea` will be focused when the user clicks or touches inside the editor as well as with the tab key and remote controls. Focusing a text input triggers the display of native virtual keyboards.

Pasting

When a the `textarea` is focused, pasting via keyboard shortcuts is natively available. To enable pasting with the context menu, CodeMirror demonstrates a technique where the `textarea` will be moved to the pointer's position on a `mousedown` event. Following the order of `MouseEvent`s, this will be completed before the context menu will be triggered. This way it will be triggered on the `textarea` and contain a paste option. The paste event will insert the contents from the clipboard to the `textarea` from which the contents can be read.

Reading input

`textarea` elements support `input` events which can be used to read the text entered by the user. The input can be processed as discussed in **12.5: Input flow** and be removed from the `textarea`. In practice, this means that once a single character has been entered in the `textarea` it will be read from the `textarea`, inserted into the editor's contents and the `textarea` will be cleared again. Input reading requires further processing before it can be passed on to

trigger a change in the editor's contents. The specifics on processing the input will be discussed in section **12.7: Input Pipeline**.

Editing mode

Using a `textarea` element allows plain-text input only. This is not a problem for regular keyboard input but rich-text contents pasted from the clipboard will be inserted as plain text and all formattings will be removed. To come around this issue, a `div` element in editing mode can be used instead of a `textarea` element.

As discussed in **Part II: Discussion** HTML editing APIs are very problematic and a key factor of this thesis is implementing a rich-text editor without using them. However using an element in editing mode only for input reading is not affected by these issues. Whenever a single character will be entered it will be read and immediately removed from the editable element. Formatting commands will not be used at all. Problematic text input behavior, for instance different markup that will be generated by the entering a line break, will not occur since the the editable area will only be used for reading input, the text that will be inserted in the editor will be generated by the library (see **12.7: Input Pipeline**). The only difference between a `textarea` and the editable element lies in the different contents it accepts for pasting. **7.3: Clipboard** discusses the problem that text pasted from the clipboard cannot be processed with native APIs across all browser. In this case, the clipboard contents will be pasted to a designated field from which it can be read, isolated from the editors contents, which solves this problem (see section **12.8: Pasting**).

12.7 Input Pipeline

Overview

Before the text read from the hidden input field will be passed on to the `Contents` class, it will be passed though a series of `Filters`, called the "input pipeline". The input pipeline has 3 basic responsibilities.

- Stop and dispatch input that that should trigger functions of the library

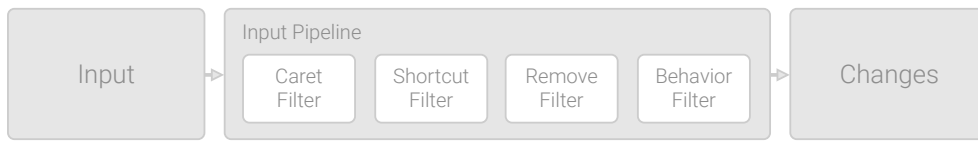


Figure 12.2: Input pipeline with sample filters

- Implement rich-text editing behavior
- Filter and transform text

The input pipeline is part of the **Input** class. The pipeline itself is an array of input filters that can be added, removed and ordered with a designated API.

Input filters have a public API that specifies for which input they should be called. For instance, a filter can specify to be called when the user presses the `ctrl``s` key combination. A filter can specify a handler for the input. For pressing `ctrl``s`, a filter can specify to call a "save" function. The input is passed to the filters as an event (see **12.13: InputEvent**) that can be stopped from bubbling. This way it can be prevented that pressing `ctrl``s` will also insert an "s" character to the editor. The order in which filters will be called is important. Some filters process the input and must cancel the event not only to prevent a character to be inserted, but also to prevent other filters from taking action.

The basic filters implemented for this library will be listed hereinafter, grouped by the responsibility as listed above.





Triggering functions

Caret Commonly, when a user presses one of the arrow keys inside a text, the caret will be moved and of course, in a browser environment, this is not different. To mimic this behavior arrow key input will be intercepted by the **Caret** input filter class, which will move the caret that has been instantiated by the **Type** base class (see **12.5: Input flow**). It will also account for modifier keys and the operating system used and move the caret accordingly.

Command The **Command** input filter class checks for and intercepts keyboard shortcuts commonly used for text formatting.

-   Formats the currently selected text bold.
-   Formats the currently selected text italic.
-   Formats the currently selected text underlined.


To format the text, by default, the tags **strong**, **i** and **u** will be used. Since in some cases this might not be desired, these default keyboard shortcuts must be opted-in by setting an option on instantiation (see **12.3: Instantiation and usage**).

The input event implements an abstraction to either check for the  key or the  key depending on the operating system of the user (see **12.13: InputEvent**) so that the above-mentioned shortcuts can (only) be triggered with the  key instead of the  key on the OS X systems.

Rich-text behavior

As discussed in **9.4: Native text input imitation**, most rich-text editors support the user with a common behavior reacting to the user's input. This behavior can be abstracted in a simple way using the input pipeline. The following paragraphs describe the rules and filters implemented by default. Further rules can easily implemented and added to the input pipeline as input filters.

Headlines When a user presses  while the caret is located at the end of a headline, a new text paragraph will be created behind the headline and the caret will be placed inside it.

Lists Pressing  inside a list item creates a new list item behind the current item and the caret will be placed inside it.

Filtering and transformation

Line Breaks To display a line break in the contents of the editor a **br** or **p** tag must be inserted⁵. When the user presses the  key it does not suffice

⁵To enforce a line break other tags. like the **div** tag can be used, but would not be semantically appropriate in this context

to insert a carriage return and/or line feed. This input will be intercepted and instead a `br` tag will be inserted to the editor's contents. As discussed in **12.7: Overview**, it is important that this filter will be invoked after the **Headlines** and **Lists** filters, so they can apply their behavior and prevent this filter's behavior.

Remove To delete text from the editor the **Remove** filter checks for `backspace` and `del` key inputs. Depending on whether there is a text selection or not, it either deletes the selection's contents or one character left/right of the caret.

Spaces Browsers display adjacent spaces as a single space. This is an unusual behavior for text editing. The **Spaces** input filter checks if adjacent spaces are being entered and inserts non-breaking spaces.

Events

As an alternative approach input filters could be implemented as input event handlers. With this, the same functionality could be achieved without an input pipeline. A designated pipeline however provides a clearer mental model for processing the input as it allows a separation of concerns. An input filter has a specific purpose and context as compared to an arbitrary input event handler. It can also be made sure that filters will be called in the right order and that any input filter will be run before triggering an input event. This way input event handlers only receive actual text input for the editors contents while keyboard shortcuts and other keypresses have been filtered out.

Extendability

The input pipeline is intended to be extended. It serves as an entry point for other developers to process input. For this, the **Input** class provides an API to add, remove and reorder input filters.

12.8 Pasting

As discussed in **12.6: Hidden native input fields**, all text input will be read from a designated input field. It is useful to distinguish between regular

keyboard input from input pasted from the clipboard since the clipboard can contain rich text contents. Developers implementing an editor with Type should be able to determine which formattings should be allowed in the editor, i.e. pasted from the clipboard. This requires two steps.

1. Determine if an input has been made through typing or pasted from the clipboard.
2. Process the clipboard contents and make them accessible to developers.

Paste detection As discussed in **7.3: Clipboard**, modern browsers trigger paste events, which can be listened to. Not all browsers allow reading contents from a paste event, but this is not necessary. The paste event will insert its contents into the designated input element from which its contents can be read, after the event has completed. Some older browsers, specifically Opera versions older than 12.1 do not trigger paste events at all. For legacy support, the browser can be tested for an available clipboard API and in case it is missing the regular text input can be checked for its text length. With the system discussed in **12.6: Hidden native input fields**, an input will always have the length of a single character. If the input is longer than that, this either means more than one character has been inserted or a single formatted character has been inserted⁶. These cases can only happen when contents have been pasted from the clipboard. However, if a single unformatted character has been pasted, it cannot be distinguished from a regular text input. According to statcounter.com, 0.08% of all internet users use Opera version older than 12.1. It is to be noted that the use case this feature is designed for, is to sanitize pasted input, which is not necessary for a single plain text character input, although it must be acknowledged that there can be use cases requiring to register any paste event for other reasons.

Processing To process the pasted contents and possibly prevent inserting the contents to the editor an `InputEvent` will be generated and passed through the input pipeline. Any filter can be implemented to treat or ignore paste events. Users of the library can set an option on instantiation to determine

⁶The input field will contain markup of more than one character

how to treat pasted contents. These options include to allow plain text only, to allow any formatted text or specifying rules to allow specific formattings only. A full API description can be found at ABC. These options are implemented in the `Paste` filter, that will either let any contents pass through (allow any formatting) or filter out specific or all HTML tags.

12.9 Caret

The `Caret` class provides all functionality to place and move a caret in a text. It provides methods to be moved left, right, up and down in a text as well as to be placed at a specific position in a text. The visual representation of the caret is a `div` element, styled to imitate a text caret. Using a CSS3 animation, it imitates the "blinking" common for native text carets.

It is to be noted that the elements for carets as well as for the text selection will not be written to the editor's contents. As discussed in ??: ??, the editor's contents should not contain any markup other than for the text itself. Instead this and all other elements will be stored in a designated `div` element at the end of the website's `body` and be positioned using CSS.

The challenge with this class is that it must be able to be moved within text and in any kind of formatting, represented by any combination of DOM nodes. To be moved across letters and text lines, the caret must take into account that:

1. Letters have different widths and heights
2. Different fonts have different letter dimensions
3. Different formattings like a headline, italicized text or text with a specifically set font size, result in different letter dimensions

`CodeMirror` solves these problems by measuring each letter with the use of text ranges. Browsers offer a `Range` interface, a construct that has a start- and end-offset in a text. A range has methods to read its x- and y-coordinates on the website. These methods can be used to span a range over a single character, read its offsets and place the caret next to it using CSS and giving it the same height as the character.

To move the caret left or right, the according characters left and right of its current offset will be measured using this method. To move it up and down across text lines, the caret must check the offsets of every character, starting from the character of the current offset, until it reaches the character above or below it that is closest to its horizontal position. As discussed in section **12.15: Cache**, a cache to store positions cannot be applied. The complexity of this is method $O(n)$, however in practice, the number of characters this will affect is limited by readability and usability of the text editor. Mobile devices, that generally have less performance than desktop machines, have smaller screens displaying less characters per line. While this is not necessarily the case, it can be expected by good software design.

BiDi support

todo

IME

todo

<http://marijnhaberbeke.nl/blog/browser-input-reading.html>

https://en.wikipedia.org/wiki/Input_method

12.10 Selection

Using a designated input element for input reading comes with the cost of having to emulate the text selection. When the input field is focused, any selection on the web site, including that of the editor, will be removed. When text is selected, the input field does not necessarily have to be focused. To read inputs, it can be focused on a "keydown" event, which will only remove the text selection when the user enters text. This is not problematic since selections will be removed on native inputs when a user enters text too. However, if the user right-clicks in the editor, the input element will be focused to enable pasting from the context menu (see **12.6: Pasting**). This will remove the text selection on any right click.

The W3C specifies an API to add multiple ranges to a selection, which should appear as multiple selections to the user. This way the element for input reading could be focused while other parts of the website, i.e. the editor's contents, could display a selection at the same time. However, while the API is available across all major browsers, it is dysfunctional and documented to not be working.

CodeMirror, ACE and Google's document editor each implement text selections by displaying `div` elements that mimic the look of a native selection. Type uses the same technique to show a text selection while the input element is focused. This mimicked selection replaces native selections entirely and will be created dynamically when the user clicks in the text and drags his or her mouse across the text.

The downside of this technique is that copy commands will not work anymore due to the fact that there is no actual text selection that can be copied, even though it appears to the users there is one. To treat this issue, CodeMirror adds the contents of the imitated selection to the input field and selects these contents with a native selection. This allows the user to use keyboard shortcuts at any time and to copy text with the context menu. When the user types, the selected contents in the input field will be overwritten by the browser, so this does not affect input reading.

12.11 Contents

The **Contents** class provides an API to add, remove and format text. This functionality is implemented through the **Writer** and **Formatter** classes. Its central responsibility is to proxy commands to these classes and to trigger **ChangeEvents** (see **12.13: ChangeEvent**) that contain a formalized description of the edit operation. **ChangeEvents** will be observed by the **UndoManager** (see **12.12: Undo Manager**) to implement undo / redo functionality.

This architecture has been chosen not only to achieve a decoupled software architecture with a clear separation of concerns, but to allow extensions to observe content changes. For instance, the Etherpad extension uses this to populate changes to other clients (see **12.16: Type Client implementation**).

Writer

The `Writer` class implements functionality to add and remove contents to and from the editor. Along with the `Formatter` class, this is the lowest layer of the editor that will perform DOM operations to modify the contents in the browser.

Formatter

The `Formatter` is one of the key classes of the Type library. As discussed in **8.4: Formatting** it must generate *well-formatted* markup while being able to work on any *ill-formatted* markup it will be given. There is a virtually infinite number of edge-cases for markup that formatting commands can be applied to. Assume we have the following string.

```
1 <p>Lorem ipsum <em>dolor<u> sit amet</u></em> consec</p>
```

Listing 12.5: Markup with highlighted target for formatting

Listing 12.5 represents markup for the formatted string "Lorem ipsum dolor *sit amet*". The highlighted part (yellow) represents the part of the text that should be formatted using a formatting command.

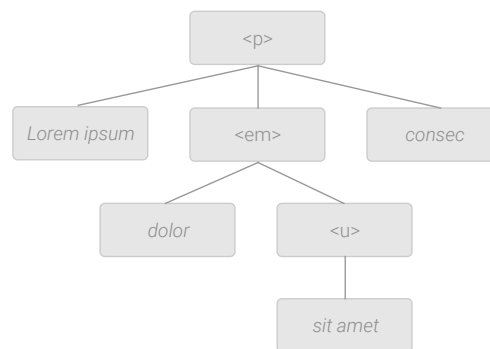


Figure 12.3: DOM representation of figure 12.5

Figure 12.3 shows the DOM representation of the markup of *Listing 12.5*. We can split up the text node "Lorem ipsum" into two text nodes "Lorem" and " ipsum" to create a distinct node in which the formatting (yellow highlight) starts and do the same with the ending text node "sit amet". This gives us a

distinct nodes for the start and the end of the selection. When we traverse each node from the start to the end, every traversed node falls on either of the following cases:

1. It is the start node
2. It is the end node
3. It is a node between start and end node (but is not and does not contain the start or end node)
4. It contains end node

To generate *well-formatted* markup as discussed in **8.4: Formatting**, the following 2 rules must be followed:

1. The markup must conform the validation rules of HTML as specified by the W3C.
2. The markup must use as little DOM nodes as possible.

Algorithm 5 demonstrates a recursive algorithm, that will start iterating from the given start node over all its subsequent siblings until there are no siblings anymore *or* it found the end node *or* it found a node containing the end node.

- If the end node was found, the algorithm will wrap all all nodes it found with a node that applies the desired formatting.
- If there are no more siblings, it means it has reached the last sibling inside the node containing the start node. By the validation rules of the W3C, nodes are not allowed to intersect. The algorithm will wrap all nodes it found so far with a node that applies the desired formatting and recursively applies itself with the start node being the next node in the document flow and the end node remaining the same end node.
- If an element has been found that contains the end node, the algorithm will wrap all nodes it found so far with a node that applies the desired

Algorithm 5 Recursive algorithm to apply text formatting

```

1: procedure FORMAT( $s, e$ )  $\triangleright$   $s$  and  $e$  are the distinct start and end nodes
2:    $c \leftarrow s$ 
3:   Let  $a$  is an empty array
4:   while  $c \neq \text{null}$  and  $c \neq e$  and  $c$  does not contain  $e$  do
5:      $a.\text{push}(c)$ 
6:      $c \leftarrow c.\text{nextSibling}$ 
7:   end while
8:   if  $c = e$  then
9:      $a.\text{push}(c)$ 
10:  end if
11:  if  $c$  contains  $e$  then
12:    FORMAT( $c.\text{firstChild}, e$ )
13:  end if
14:  if  $c = \text{null}$  then
15:     $n \leftarrow$  next node in document flow
16:    FORMAT( $n, e$ )
17:  end if
18:  Wrap all nodes from  $a$  with DOM node to apply formatting
19:  Connect siblings to wrapping node if they have the same tag
20: end procedure

```

formatting (to avoid intersecting nodes). It will then apply the formatting algorithm recursively to the first child of the containing node. If this node in return is another container to the end node, the recursion will repeat until a sibling of the end node or the end node itself has been found.

Each rule performs the minimally necessary steps to format contents while conforming the HTML validation rules of not intersecting DOM nodes. By adding only the minimum of nodes, this will ensure simple (and valid) markup.

To format the nodes that have been collected by the algorithm, they will be wrapped by a DOM node that applies the desired formatting. The wrapping function will also remove any nodes between the start and end node that have the same tag as the node used for the formatting to clean up the markup. If the start or the end node has been contained by a node of the same tag as the node the text should be formatted with, the containing nodes will be used as start or end node. As a last step to improve the markup, the nodes

left and right of the formatting node will be unified with the formatting node, if they have the same tag. These steps will simplify potentially *ill-formatted* markup that the formatting command affects. All DOM manipulations will be performed at the end of the algorithm, when all nodes have been read to improve performance.

12.12 Undo Manager

The `UndoManager` class will observe `ChangeEvents` (see **12.13: ChangeEvent**) and keep a history of the performed changes. Every change must include an identifier of the source of the change. The `Contents` class will declare the `Input` class as the source for the change. This is to enable extensions to work with the `UndoManager`. The Etherpad extension (see **12.16: Real-time collaboration with Etherpad**) will change the editor's contents too. If, at a given time, the local user added some text and after that another connected client would add some text a few characters before the local user's change, the `UndoManager` must account for both changes to undo the local user's change.

Change Listener

In its core the changes observed by the `UndoManager` will be kept track of by the `ChangeListener`. It will accumulate every change in a data structure and take a "snapshot" every 500 milliseconds and store it in a history. The `UndoManager` uses this history to implement and offer undo and redo commands.

This functionality has been abstracted to be reused by other extensions. The Etherpad extension uses this class to accumulate and publish local changes every 500 milliseconds to its connected clients independently of the `UndoManager`.

Undo / Redo

The `Change` event can contain operations to

- Insert or remove text
- Format text

Inserting and removing can be undone easily by performing an insertion to undo a remove operation and vice versa. Formatting text is more difficult since `Type` allows formatting with arbitrary markup. This can be solved by keeping track of the offsets and characters affected by the formatting command and removing the HTML tags at the corresponding positions.

12.13 Events

Overview

It is possible to trigger custom (native) events using the `CustomEvent` interface on modern browsers⁷. Internet Explorer 9 and below allow this through similar interfaces. These interfaces could be used for triggering events for components of `Type`. However, this would trigger events that are only relevant within the library in the global namespace. To avoid this, `Type` implements its own event system that populates events only within the library. Events from within the editor that can be useful to the website or web application should still be triggered as native events in the browser's global namespace. During the development of the library no such use case has occurred.

Event Api

The `EventApi` class provides an API to add and remove event listeners as well as to trigger events. It provides instance and static methods.

```
1 // Static methods
2 EventApi.on(eventName, eventHandler);
3 EventApi.off(eventName, eventHandler);
4 EventApi.trigger(eventName, eventObject);
5
6 // Instance methods
7 EventApi.prototype.on(eventName, eventHandler);
8 EventApi.prototype.off(eventName, eventHandler);
9 EventApi.prototype.trigger(eventName, eventObject);
```

Listing 12.6: EventApi methods

⁷https://developer.mozilla.org/en/docs/Web/API/CustomEvent#Browser_compatibility

Using the **OOP** class (see **12.14: OOP**), these methods will be inherited by the **Type** class. This way, using the **trigger** method, events can be triggered within the scope of a **Type** instance and be observed using the **on** method. Event handlers can be removed using **off** method.

The static methods will also be inherited by the **Type** class. This is necessary to trigger events that are *Type* specific but not *instance* specific. Most importantly a "ready" event will be triggered on every instantiation of the **Type** class. This way plugins and other third-party scripts can run initialization routines.

Plugins and third-party libraries can trigger arbitrary events and pass along arbitrary data. As a paradigm and in terms of a *good programming style*, event objects should be passed. Event objects are inherited from the **TypeEvent** or conform its API.

TypeEvent

```
1 // Gets or sets data
2 TypeEvent.data([options]);
3
4 // Stops the event from bubbling
5 TypeEvent.cancel();
```

Listing 12.7: TypeEvent API

The **TypeEvent** is a generic, general-purpose event. It can store arbitrary data and offers an API to be stopped from bubbling. A full API description can be found in Listing XY.

InputEvent

The Input **InputEvent** will be triggered by the **Input** class after a keyboard input has passed the input pipeline. It inherits all methods from the **TypeEvent** to conform the event system and contains information on the key and the modifier keys pressed. The key is represented with its key code and a key name. The key name will be mapped from the key code and is implemented with a list of readable names including "backspace", "enter", "space", all letters and many others. The list not complete but can be extended during further

development. The modifier keys include "shift", "alt", "ctrl" and "meta". "meta" is the browser's name for the `⌘` key on OS X systems. OS X uses the `⌘` key as modifier the same way Windows and Linux use the `⌃` key. To support developing an editor for both platforms the "cmd" modifier will be set when a user holds the `⌘` key on OS X *or* the `⌃` key on other platforms.

PasteEvent

The `PasteEvent` will be triggered when the user pastes contents from the clipboard *before* it will be inserted to the editor's contents. It contains the clipboards contents and can be cancelled so that other developers are free to manipulate and stop a paste event.

ChangeEvent

As discussed in section **12.11: Contents** a `ChangeEvent` will be triggered when changes have been made to the editor's contents. It contains an array of formalized data structures describing the changes that have been made. There are two types of data structures, one for text insertion and removal and one for text formatting.

```

1  {
2      "offset": 10,
3      "operation": "insert",
4      "contents": "Lorem ipsum",
5  }
6  {
7      "offset": 10,
8      "operation": "remove",
9      "contents": "Lorem ipsum",
10 }
```

Listing 12.8: Data structure for insert and remove operations

Lines 1–5 in *Listing 12.8* show the data structure for an insert operation.

```

1  {
2      "offset": 10,
3      "characters": 5,
```



```
4      "operation": "format",
5      "parameters": "<strong />",
6  }
7  {
8      "offset": 10,
9      "characters": 5,
10     "operation": "removeformat",
11     "parameters": "<strong class='important' />",
12 }
```

Listing 12.9: Change data structure

These data structures contain the information to undo user input and to redo the same input again.

12.14 Utility classes

OOP

The **OOP** class extends the constructor pattern with basic classical inheritance. It provides the method **inherits** that will duplicate and copy the **prototype** from one **Function** object to another. It also copies attributes and methods defined on the **Function** object itself to implement inheritance of static definitions. It adds the attribute **_super** to the inheriting **Function** object referencing its parent class to enable child classes to access their respective superclasses.

Range

The **Range** class is an abstraction for the native **Range** interface. The native implementation is prone to bugs on many browsers. Instead of fixing the API by shimming its methods, the **Range** class implements all methods related to ranges while trying to interact as little as possible with the native API. On top of the methods of the native **Range** interface, this class implements additional methods required for **Type**.

Dom Walker

Working with text implies having to traverse the DOM, i.e. the nodes inside the text often. The `DomWalker` utility class solves this problem. The DOM API offers methods to access a node's siblings, children and parents, but it must always accounted for cases when any of these is `null` (there is no parent, sibling or child) or when they overflow the bounds of the editor's contents. But more importantly, for text editing, it is usually necessary to access the next (or previous) node in the document's content flow which can either be the parent, sibling, child or a node that can only be accessed by traversing multiple nodes. Also, it is often the case that it is not only necessary to fetch the next node, but to apply a filter to only fetch a specific node, for instance a text node or only a text node that has contents visible to the user⁸. Browsers offer a native API for this, called `TreeWalker`, but it is said to be slow⁹, only partially supported by Internet Explorer 9¹⁰ and has been criticized for its verbose API¹¹.

The `DomWalker` can be instantiated by providing a starting node and the type of nodes it should traverse. The latter argument can either be a string identifying a pre-made filter of the `Type` library or a custom filter function. Pre-made filters include:

- "text" - A text node with visible contents
- "textNode" - Any text node, visible or invisible
- "visible" - Any visible DOM node

A `DomWalker` instance offers the high-level methods `next`, `previous`, `first` and `last` (amongst others) for traversal. A full API description can be found in XYZ. All methods can be called statically without having to instantiate a `DomWalker`.

⁸Any text node consisting of whitespace only will not be displayed by any major browser

⁹<http://jsperf.com/qa-vs-node-iterator>

¹⁰https://developer.mozilla.org/en-US/docs/Web/API/NodeIterator#Browser_compatibility

¹¹By John Resig, author of jQuery <http://ejohn.org/blog/unimpressed-by-nodeiterator/>

Text Walker

The **TextWalker** class acts as a container for all functions related to measuring text offsets. It provides utility methods to determine the character offset from one text node to another or, vice versa, which text node can be found at a text offset, starting from another given node. Both methods are required by various classes and are thus, centralized.

Dom Utilities

The **DomUtilities** class encapsulates common methods for all DOM operations other than traversal. It has no inherent purpose but many other classes perform the same DOM operations, which hence reside in a common library to avoid code duplication.

Utilities

The **Utilities** class is a general-purpose class that contains methods to extend JavaScript's features. It contains methods to work with data structures and to detect object types.

Environment

The **Environment** class checks and provides informations on the current browser environment and its features. This class is especially important to mimic native behavior for user interaction. For instance, as discussed before, either the control key or the command key should be used to implement keyboard short-cuts depending on the operating system. To check for specific feature support, it is favorable to use duck typing within each class.

Settings

The settings class stores settings required for Type's modules, for instance the `id` of the DOM-container which all helper DOM-nodes from other classes will be appended to.

Development

The `Development` class is intended to contain utility methods to support the development of the library. As for the development of this thesis, it was sufficient to implement logging methods.

12.15 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. When the user edits the contents of the editor, these texts will change and the cache must be updated. A cache must also account for external changes. The DOM3 Events specification¹² offers `MutationObservers` to check for DOM changes. This feature is not supported by Internet Explorer version 10 or less¹³. Internet Explorer 9 and 10 offer an implementation for `MutationEvents`¹⁴. 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."¹⁵ Apart from that, the benefits of a cache may not significantly 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 noticeably affect the CPU. For this reason, the editor does not use any caching. Implementing an editor that is stateless in regards of its contents can also improve stability.

¹²<http://www.w3.org/TR/DOM-Level-3-Events/>, last checked on 07/21/2015

¹³<http://caniuse.com/#search=mutation>, last checked on 07/21/2015

¹⁴<http://help.dottoro.com/ljfvvdm.php#additionalEvents>, last checked on 07/21/2015

¹⁵<http://www.w3.org/TR/DOM-Level-3-Events/#legacy-mutationevent-events>, last checked on 07/21/2015

12.16 Real-time collaboration with Etherpad

Overview

To achieve real-time collaboration with multiple Type editors, Etherpad¹⁶ can be used. Etherpad is a web-based collaborative real-time text editor with rich-text capabilities. It provides a server, written in JavaScript using Node.js as well as a web-based rich-text editor. Both components are distributed in one package and are meant to be used together.

To achieve real-time rich-text collaboration, multiple web-based clients communicate with a server via WebSockets using socket.io. Each client owns a local version of the document that it needs to sync with the server. The server uses an operational transformation algorithm to merge each change to the document accounting for all changes and then urges each client to update their local contents according to the final document.

Changesets

For this, Etherpad uses the concept of so-called "changesets". Each client sends its local changes, debounced to an interval of 500 milliseconds, as a serialized string—the changeset—to the server. The changeset includes all text insertions, removals and formattings of the last time frame. Along with the changeset, it sends a document revision number that the changeset is based on to the server. The document revision number increases with every changeset that has been accepted and applied to the document on the server side. The document on the server side is saved as a stack of changesets, which ultimately form the current document. For performance reasons, snapshots can be taken that save the document as formatted text.

Based on the revision number that the client provides with the changeset, the server can apply it to the version of the document the client was working on. The server will apply the resulting changes to all newer revisions of the document (if present) and send a changeset and the latest revision number back to the client. The changeset sent to the client includes all operations

¹⁶Etherpad has been completely rewritten under the name Etherpad Lite. However, its official website no longer links to its former source code. For simplicity, the name Etherpad will be used, referring to its rewrite as Etherpad Lite

it needs to perform to update its local version to the newest version on the server.

As a last step, the client must apply the changeset it got from the server to its local document to display the most recent version to the user and update its local revision number to what it got from the server.

Merging

In a collaborative environment, it can happen that two (or more) clients send different changesets to the server that are based on the same document revision. It is the responsibility of the server to merge both changes so that it preserves either intent. As explained in the "Etherpad and EasySync Technical Manual"[citation needed], to solve this, for a document X with the conflicting changesets A and B , the server computes the new changesets A' and B' such that

$$XAB' = XBA' = Xm(A, B)$$

where $Xm(A, B)$ is the merge of A and B applied to the document X . The changesets A' and B' will be sent to the respective clients, which will apply it to their local documents to sync with the document on the server. To compute a changeset A'

- Insertions in B become retained characters in A'
- Insertions in A stay insertions in A'
- Retain whatever characters are retained in *both* A and B

For B' this applies vice versa.

Etherpad Client implementation

Clients interact with the server via WebSockets using socket.io. To sync their own changes with other clients, a client does 4 things.

- Request a the full document from the server
- Send a changeset to the server

- Receive acknowledgement from the server for a submitted changeset
- Receive a changeset from the server submitted by another client

When Etherpad's client connects to the server it receives an initial snapshot of the entire document as a string. To submit changes, the client uses a three-step architecture. The client stores any local changes that have not been sent to the server yet in a changeset Y . Any changeset sent to the server must be acknowledged by the server as it has applied the changeset to its document. Any changeset that has been sent and not been acknowledged yet will be stored in a as the changeset X . The document as it is acknowledged by the server is stored in a changeset A . The document visible to the users can be expressed by the representation of $Y \cdot X \cdot A$, i.e. applying each changeset to the next.

Whenever a user applies a local change, the changeset Y will be updated. Every 500 milliseconds, but not before a changeset submitted to the server as been acknowledged, the changeset Y will be sent to the server and Y will be assigned to X . Y will be set to a changeset that contains no changes. When the client hears the acknowledgement for X from the server, X will be applied the changeset A and X will be set to contain no changes.

This architecture supports receiving changesets from other clients as they must be applied to a client's local changes (committed and uncommitted) as well as the document version as acknowledged by the server. When a client receives another client's changeset B it will perform 4 steps.

1. Compute a new changeset by merging $Y(X \cdot B)$ and apply it to the document visible to the user.
2. Apply B to A .
3. Compute a new changeset by merging B and X and assign it to X
4. Compute a new changeset by merging $(X \cdot B) \cdot Y$ and assign it to Y

The operations needed to merge the changesets on the client, are the same operations for merging changesets on the server.

Type Client implementation

Etherpad's technology can be used to enable real-time collaboration for Type. While Etherpad offers a web-based client, its implementation has three flaws:

1. It cannot be integrated easily in other web applications.
2. It does not generate semantic markup. It is cluttered with control sequences.
3. It's hard to extend.

Etherpad does not provide a documentation on its client-server protocol, but it can be reverse engineered. It is possible for third-party libraries to communicate with an Etherpad server alongside Etherpad's "native" clients, as long as a third-party library (like Type) conforms the protocol.

Etherpad's collaboration functionality comes with a cost in file size for Type and may only be used in specific use cases. This is why this feature is implemented as an optional extensions (compare **12.17: Extending**) in a separate file. To enable collaboration the designated JavaScript file needs to be added to the website.

```
1 <script src="type.js"></script>
2 <script src="type.etherpad.js"></script>
```

Listing 12.10: Enabling real-time collaboration to Type

`type.etherpad.js` adds the classes it requires to the `Type` namespace and adds a static constructor to the `Type` library:

```
1 var element = document.getElementById("myElement");
2 var editor = Type.fromEtherpad(element, "http://example.com/
  editor/myEditorId");
```

Listing 12.11: Static constructor to generate a collaborative Type instance

The constructor used in line 2 of *Listing 12.11* will connect to an Etherpad server and load the contents of a document and append them to the element given as first argument.

Todo Will use the change listener to watch changes. Has an own class to serialize changesets. One class for pushing changes to the server. Uses caret class to display other collaborators. And has a class to apply incoming changes.

This architecture provides an unobtrusive way to integrate real-time collaboration in the Type library. It does not depend on a specific implementation of an editor. Developers are free to implement any editor specific to their needs with integrated real-time collaboration.

12.17 Extending

Overview

Type’s modular structure is designed for extension. Type’s `prototype` has been exposed as `Type.fn` and all its classes in the `Type` namespace. This provides other developers with all of Type’s functionality in a structured and accessible manner. Type is designed to lower the barrier for and encourage developers to extend Type by giving freedom and possibilities in how to implement an extension and trying to avoid compulsorily use of interfaces or configurations.

jQuery demonstrates a similarly liberal approach for writing extensions and experience shows that name conflicts are minimal and “good” extensions are naturally favored over “bad” extensions by the community of web developers.

API extension

```
1 Type.fn.myMethod = function () {};
```

Listing 12.12: Example Type instance API extension

As discussed in **12.3: Exposal of Type’s prototype**, to add a method to Type’s public API, its base class’ prototype can be extended with a function using the `Type.fn` shorthand attribute. Static constructors can be added by extending the `Type Function` object.

```
1 Type.myConstructor = function () {
2   return new Type();
```

```
3  };
```

Listing 12.13: Example custom static constructor

Namespace extension

As discussed in **12.3: Namespace and references**, to implement extensions for Type, the Type namespace can be used to add custom classes or sub-namespaces.

```
1  Type.MyClass = function () {  
2    var caret = new Type.Caret();  
3  };
```

Listing 12.14: Example Type namespace extension and usage of a built-in class.

All other classes that Type uses are exposed in this namespace and can be used by extensions.

Plugin API

A plugin may need to be initialized when an editor will be instantiated. To support this, Type will trigger an event on instantiation and pass the Type instance to the event handler

```
1  Type.on('ready', function(typeInstance) {});
```

Listing 12.15: Example event handler for a Type instantiation

To store and read data specific to an instance, Type offers the **data** method, that will return an **Object** for arbitrary access.

```
1  Type.fn.myMethod = function () {  
2    this.data("myPlugin").foo = 'bar';  
3    var bar = this.data("myPlugin").foo;  
4  };
```

Listing 12.16: Example calls to format text

To give each plugin an own namespace, an arbitrary identifier must be passed as a **String** to the **data** method, which will provide a unique **Object** for different string identifiers. This can possibly cause name conflicts if two plugins choose to use the same string. Developers are advised to always use their own extension name as identifier. Experience with jQuery's plugin system as well as jQuery's **data** method shows that while this cannot prevent name conflicts, it is rarely a problem.

Part V

Evaluation

12.18 Mobile Support

12.19 Development / Meta

Crockford style is a bad idea. I will change it to Standard or Airbnb <https://github.com/airbnb/javascript/tree/master/es5>

12.20 Outlook

Over time, the bugs of HTML editing APIs will decrease. Its clipboard capabilities are on the way to be expanded. The API still is still limited and needs a revision. It is even imaginable to rethink the way `contenteditable` works. Editors that, for instance, implement layouting, like Google's document editor, still cannot be implemented with the way HTML Editing APIs are designed.

To allow a transition from current HTML editing APIs and an interface with a cleaner and richer functionality, it is thinkable to introduce a new "class" alongside the old API. This has been done with other functionality, for example mal aus MDN raussuchen. This way the old API can die gracefully while web developers slowly adopt. It can be hoped that if the API is much better, the adoption will happen quickly.

As discussed in **7.2: Possible third-party solutions for other languages** my design as a library with a super duper api allows implementing highlighting for other languages like bb code or markdown. *There should be a part in CONCEPT that explains this idea, either explaining its made for extensibility or in how cool my api is i mean the design as a lib and not as an editor is*

besseres undo durch erkennen von ganzen worten (wenn man leerzeichen und so drückt)

Events zu allen gelegenheiten triggern für andere developer

DOCH document model benutzen weil der shit von prosemirror einfach so geil ist.

Auf der anderen Seite ist so was wie der Medium editor mit meiner Version viel besser

List of Figures

4.1	Usage of HTML editing APIs in CKEditor and TinyMCE	19
9.1	Rendering of highlighted source code in Ace and CodeMirror	50
10.1	Diagram of the Type library and its internally used classes (excerpt)	56
12.1	Components instantiated by the Type base class	67
12.2	Input pipeline with sample filters	73
12.3	DOM representation of figure 12.5	80

Listings

4.1	An element set to editing mode	16
4.2	Emphasizing text using the HTML editing API	17
4.3	Creating a link using the HTML editing API	18
7.1	Markup of italic command in Internet Explorer	28
7.2	Markup of italic command in Firefox	28
7.3	Markup of italic command in Chrome	28
8.1	Example calls to format text	42
8.2	Different DOM representations of an equally formatted text . .	44
10.1	API for implementing a rich-text editor	55
12.1	Type instantiation	64
12.2	Example commands to format text	64
12.3	Example chaining	64
12.4	Declaration of Caret, Range and Environment classes	65
12.5	Markup with highlighted target for formatting	80
12.6	EventApi methods	84
12.7	TypeEvent API	85
12.8	Data structure for insert and remove operations	86
12.9	Change data structure	86
12.10	Enabling real-time collaboration to Type	94
12.11	Static constructor to generate a collaborative Type instance . .	94
12.12	Example Type instance API extension	95
12.13	Example custom static constructor	95
12.14	Example Type namespace extension and usage of a built-in class.	96
12.15	Example event handler for a Type instantiation	96
12.16	Example calls to format text	96

Bibliography

[bf,] 2 active tickets by component – ckeditor.

[aw,] 7 user interaction — html5.

[ar,] Apple - press info - apple releases safari 3.1.

[bi,] Bug list.

[bg,] Bug list.

[med,] The bug that blocked the browser — medium engineering — medium.

[can,] Can i use... support tables for html5, css3, etc.

[at,] Chrome releases: Stable channel update.

[as,] Content editable - web developer guide | mdn.

[av,] Html standard.

[bh,] Issues - chromium - an open-source project to help move the web forward. - google project hosting.

[ad,] Methods (internet explorer).

[am,] Midas - mozilla | mdn.

[br,] Minimizing browser reflow | best practices | google developers.

[aq,] Opera 9.0 for windows changelog.

[ap,] Opera changelogs.

[sop,] Paste as plain text contenteditable div & textarea (word/excel...) -
stack overflow.

[ai,] Rich-text editing in mozilla | mdn.

[ag,] Top 9 browsers from june 2014 to june 2015 | statcounter global stats.

[ah,] The whatwg blog — the road to html 5: contenteditable.

Part VI

Appendix

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 .1: HTML Editing API

Declaration of Academic Integrity

I hereby confirm that the present thesis on “A WYSIWYG Framework” is solely my own work and that if any text passages or diagrams from books, papers, the Web or other sources have been copied or in any other way used, all references – including those found in electronic media – have been acknowledged and fully cited.

.....

(Name, Date, Signature)