Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

GABRIEL BAUER

MASTER THESIS

IMPLEMENT COMMENTING FUNCTIONALITY IN SWEBLE DOCUMENTS

Submitted on 14 August 2019

Supervisor: Dipl.-Inf. Hannes Dohrn, Prof. Dr. Dirk Riehle, M.B.A.

Professur für Open-Source-Software

Department Informatik, Technische Fakultät

Friedrich-Alexander-Universität Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 14 August 2019

License

Erlangen, 14 August 2019

Abstract

A commenting functionality is an essential part of social media platforms and broadly used in the modern internet. However, some collaborative platforms like e.g. Wikipedia, the largest online encyclopedia, are still missing an effective communication channel. This thesis analyses existing commenting solutions for collaborative work, designs a commenting user interface and develops a concept to identify text ranges in an abstract syntax tree. The results are implemented into Sweble Documents which is a platform for distributed collaboration on documents.

Contents

1 Introduction

Commenting functionalities are an essential part of the modern internet and used by many websites like blogs and social media platforms. With the help of a commenting functionality, users are able to discuss aspects of the referenced content. In terms of collaborative work, text processors like Microsoft Word or Google Docs offer a commenting functionality. However, an environment that currently lacks a commenting functionality is the wiki Wikipedia[1].

A wiki is a platform, on which users write and organize content collaboratively. The first wiki became available in 1995[2]. Since then, Wikis have become an increasingly essential part in the modern internet. The best-known wiki and fifth most visited website worldwide[3] is Wikipedia. It is a freely available encyclopedia, which contents are mainly crowd-sourced. In any collaborative system, effective communication channels are needed.

Wikipedia includes only a rudimentary communication system. Each article has an additional ”talk” page, which is essentially another article used to discuss improvements. The talk page does not include the functionality to reference certain contents of the article nor does it separate replies clearly. Instead users have to use Wikipedia's markup language Wikitext in order to make answers distinguishable.

Sweble is a platform for distributed collaboration on documents. It combines the wiki approach with the collaborative concepts elaborated by Github[4], a web platform for collaborative software development built on-top of the version control system Git. Sweble is based on prior research to improve the wiki ecosystem by introducing a machine-accessible format to represent a wiki's contents (Dohrn & Riehle, 2011b) and by implementing a parser, which is able to transfer the contents into this new format (Dohrn & Riehle, 2011a). Sweble's unique selling point is its distributed collaboration. It allows users to have individual variants of a document, while still enabling users to synchronize and merge different variants.

[1]https://www.wikipedia.org
[2]http://wiki.c2.com/?WikiHistory
[3]https://www.alexa.com/topsites
[4]https://github.com/

The goal of this thesis is to design and implement a commenting functionality for Sweble. The requirements for the commenting functionality are presented in Section 2. Section 3 focuses on the fundamentals including user interface best-practices and reviews existing software solutions which integrate a commenting functionality. Section 4 presents the design of the commenting user interface and discusses the technical implementation of text references. A brief overview of the final implementation as well as a description of the key challenges and solutions is given in section 5. In section 6 the results are then evaluated.

2 Requirements

The commenting functionality shall fulfill certain requirements as presented in this chapter. In general, the commenting functionality shall be integrated into Sweble documents and each comment shall reference a specific text range in the document. The commenting functionality shall enable users to perform certain commenting tasks. The first seven functional requirements are defined from the user perspective as acceptance criteria:

1. Users can create comments in documents, reply to comments in form of a single-threaded discussion and view existing comments in the document.
2. As the user views an existing comment, the connection between the comment and the referenced text range is visualized adequately.
3. The project owner can mark comments as either resolved or rejected and the comment creator is able to mark the comment as closed in case the comment was opened by mistake.
4. The project owner as well as all users who participated in a comment get notifications on new replies or in case the comment was marked as resolved, rejected or closed.
5. As the user views an older revision of a document, the comments are displayed in a past state which correspond to the revision time.
6. The user interface shall also be able to handle a large number of comments.
7. Comments shall be preserved in forks.

Requirement 7 is developed in the chapter "Architecture" and requirement 9 is based on findings from the chapter "Fundamentals". All above requirements will be validated using a manual acceptance test in the browser. Besides the user perspective, there are further functional and non-functional requirements for the commenting functionality which are listed below:

8. The comment data shall be integrated and stored in the document format used on Sweble.

9. The user interface (**UI**) shall rely on recognition rather than on recall.
10. The UI shall be easy to use.

Requirement 8 will be validated by fetching an arbitrary document containing comments from the Sweble API[1]. The response should contain the document as well as the comments. Whether the UI relies on recognition (requirement 9) will be validated with a manual acceptance test. All commenting functions should be reachable without prior knowledge. Requirement 10 will be validated by comparing the commenting UI with UI best-practices developed in the chapter 3 (Fundamentals).

[1]Application programming interface

3 Fundamentals

This chapter gives a brief overview of some basics regarding the types of collaboration as well as a summary of UI best-practices discussed in the literature. Section 3.3 presents some existing commenting solutions and compares their implementation with the UI best practices.

3.1 Types of Collaboration

There are different approaches, as how modifications can be handled in a collaborative system. Molli, Skaf-Molli, Oster and Jourdain, 2002 categorized them as asynchronous, synchronous and multi-synchronous environments. Within an asynchronous environment one user at a time has the exclusive right to perform modifications. For example, having a text document on an USB stick and passing it along is an asynchronous environment. Only one user is able to change the contents stored on the USB-stick. A synchronous environment allows all users to simultaneously make changes. Google Docs, which is presented in subsection 3.3.2, is an example for a synchronous environment. Changes are shown in real-time and multiple users can edit a document simultaneously. However, this approach has the limitation that multiple users working on the same part of a document will interfere with each other. The collaboration approach, which solves this limitation, is categorized as multi-synchronous. A multi-synchronous environment enables users to have independent variations of a file or document. Each user is able to generate a private copy of a file and modify it. Multiple copies of a file can then be merged. However, merging files can introduce merge conflicts. Merge conflicts occur if one file contains changes at the same position as another file. While some merge conflicts can be solved automatically, e.g. import statements in programming languages, other conflicts have to be resolved manually. This approach has been elaborated especially in software development. Well-known version control systems (VCS) like Git or Subversion use a multi-synchronous environment. Sweble also uses this approach and applies it onto the concept of a wiki.

VCSs introduced some terms which are shared by Sweble and used in this thesis. In a VCS files are organized in **repositories**, which can contain multiple files in different versions. Within a repository any kind of modification (including creation, modification and deletion of files) is grouped as a **commit**. The user decides at which point multiple modifications are arranged as one commit. Furthermore, the commits represent the history of the repository. The user can view the state of the repository at the time of any commit. To do so, commits have a chronological order. Each commit will reference its previous commit with the exception of the first commit within the project. In order to achieve the multi-synchronous environment, repositories can be cloned. Cloning refers to downloading the repository, which effectively produces a local copy. New commits are created independently from the original repository, but can be pushed back into the remote repository. To avoid having new commits only locally, multiple commit sequences can be stored in one repository. Different commit sequences are accessed through a corresponding branch. Each repository usually has one main branch, often called master or develop.

The way collaborative development is carried out was especially shaped by platforms like e.g. GitHub[1] or Gitlab[2], which are built on-top of the VCS git. Both platforms offer two different ways for users to collaborate. Users can be added to the repository as **collaborators**. Collaborators have access to the repository and are able to make commits directly. Since collaborators have direct access, external users are not added lightheaded. For external users to propose changes, the user has to **fork** the repository. A fork is similar to a clone, as it creates a copy. It creates a new project containing the state of the original repository. The new repository's owner is the user, who forked the project. Modifications can then be applied independently of the original repository. In order to merge changes back into the original repository, the owner may create a **merge request**. GitHub calls this feature as pull request. However, since the term merge request seems to fit the purpose better, it will be referenced as merge request in this thesis. Nevertheless, a merge request is the request to merge changes into the original repository. The repository's collaborators will have to review the changes and either accept and merge the changes or reject them. This second approach is commonly used on open source projects and is also adopted by Sweble.

3.2 User Interface Best-Practices

Since one of the main tasks of this thesis is to design an UI for a commenting functionality, I reviewed literature on user interface design. In general, user

[1]https://github.com/
[2]https://gitlab.com/

interface design as well as user experience is part of the field human computer interaction. There are several principles discussed in literature, some of which are presented below. All these principles are technology and implementation independent.

Molich and Nielsen, 1990 originally defined 9 principles for user interface design. Nielsen, 2005 restated and extended those principles into 10 heuristics, which should be seen as rules of thumb instead of fixed rules. Those 10 heuristics are listed below:

- Visibility of system status
- Match between system and real world
- User control and freedom
- Consistency and standards
- Error prevention
- Recognition rather than recall
- Flexibility and efficiency of use
- Aesthetic and minimalist design
- Help users recognize, diagnose and recover from errors
- Help and documentation

Shneiderman and Plaisant, 2010 defined 3 principles: i) the broad range of different users with different knowledge and skills as well as users with different system requirements should be considered, ii) the system should help users to prevent errors before they are executed, and iii) the following "8 golden rules" should be followed:

- Strive for consistency
- Enable frequent users to use shortcuts
- Offer informative feedback
- Design dialogues to yield closure
- Error prevention/handling
- Permit easy reversal of actions
- Support internal locus of control
- Reduce short-term memory load

In this thesis, the 10 heuristics of Nielsen, 2005 as well as the eight golden rules from Shneiderman and Plaisant, 2010 were combined in four aspects that seemed most relevant.

As emphasized, it is important to **prevent errors**. It is unlikely, that users make errors while commenting. However, an important conclusion arising from error prevention and handling, is that comments should be delete-able by the creator of the comment. In case of a mistake, the creator should have the ability to discard the comment. A delete functionality for creators full-fills the golden rule "Permit easy reversal of actions".

Furthermore, the UI should be **consistent, in the user's language and easy to use** as suggested by Nielsen, 2005 ("Consistency and standards", "Aesthetic and minimalist design") as well as Shneiderman and Plaisant, 2010 ("Strive for consistency"). A very good example for consistent design is the color-coding of specific buttons as well as their positioning. A button that creates an item should always be of the same color throughout the system. Its color should not change throughout the application. Furthermore, a submit and cancel button are often presented together. The relative position of both should be identical within a system.

Another important aspect is to provide "**informative feedback**" (Shneiderman & Plaisant, 2010). This aspect is also stressed by Nielsen, 2005 as he suggests to have "Visibility of system status". In terms of modern web applications, loading bars or spinners are among the mostly used tools to visualize the system status. Due to the rise of single page application frameworks as e.g. React and Angular additional contents are loaded by the application itself and not the browser anymore[3]. On an ordinary web site, each page request loads a new page with images, stylesheet and the HTML file. The browser therefore shows a loading symbol. However, on a single page application only the needed content is fetched asynchronously from an API and the browser will not show a loading symbol. The web application should therefore integrate it's own loading symbol so that the user is informed of the pending network request.

In the context of a commenting functionality, a very important principle is to "reduce short-term memory load" (Shneiderman & Plaisant, 2010) and therefore rely on "**recognition rather than recall**" (Nielsen, 2005). The term recall describes the process of retrieving information from memory, while recognition only implies that something is familiar. An example of recognition vs recall is the use of a command-line interface (CLI) or a graphical user interface. In order to execute a command on the CLI, the user has to remember it exactly. On a graphical user interface, it is sufficient to identify the appropriate item on the screen. Recognizing something is easier, since it offers more clues to retrieve the

[3]https://trends.google.de/trends/explore?date=all&q=React,Angular

information.[4]

3.3 Review of Existing Commenting Solutions

There are well-known software solutions which implement a commenting functionality. I chose three software solutions and analyzed them. The commenting UIs are discussed regarding the UI best-practices presented in the previous section. All presented commenting solutions use a single-threaded discussion. Thus, a reply can only be added to the end of the discussion and not below another reply within the comment.

3.3.1 Microsoft Word

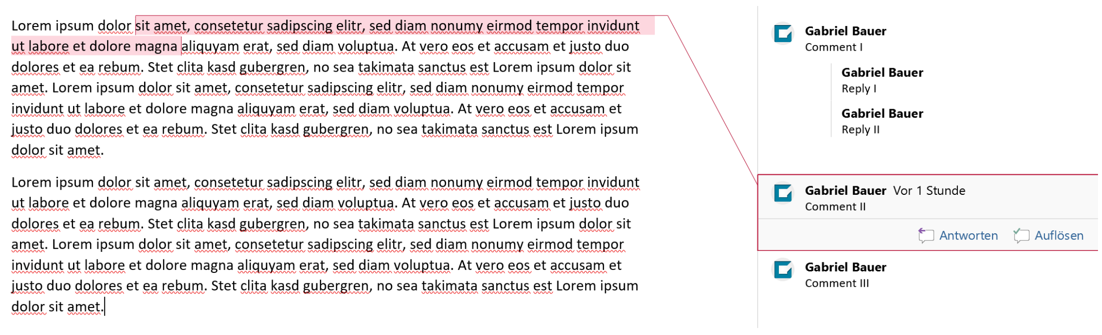
Microsoft Word is a word processor first released in 1983[5]. Nowadays it is part of the office suite "Microsoft Office", which is available for multiple platforms and as web application.

Microsoft Word features a built-in commenting functionality. It is accessible via the "review" ribbon on the top navigation. To create a new comment, the user has to select a text, activate the review ribbon and click on the comment button. If at least one comment is present, all comments for the opened document are shown on the right hand side of the document. The connection between the referenced text and the comment is visualized only if the user hovers or clicks on the comment (compare with Figure 3.1).

Comparing Microsoft Word's commenting UI with the UI best practices, the comment creation in Microsoft Word relies on recall rather than on recognition. The user needs to know where the button is located in order to create a comment. While Word displays a tooltip as a text range is selected, it does not contain a button to create a new comment referencing the selection. Furthermore, the comment view is kept simple. The corresponding text range is only highlighted if the comment is actively selected. Moreover, the user is unable to distinguish which parts of the document are commented on. All comments shown in Figure 3.1 are referencing the first three lines of the document. Without actively checking the text references, it could appear to the user that the last comment is referencing the last section. If users work collaboratively on a document, it is important to see commented text ranges.

[4]https://www.nngroup.com/articles/recognition-and-recall/

[5]https://royal.pingdom.com/first-version-of-todays-most-popular-applications-a-visual-tour/


Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Gabriel Bauer
Comment I

Gabriel Bauer
Reply I

Gabriel Bauer
Reply II

Gabriel Bauer Vor 1 Stunde
Comment II

Gabriel Bauer
Comment III

Figure 3.1: A screenshot of Microsoft Word 2019 showing the comments on the right hand side. The second comment is hovered with the mouse and therefore highlights the corresponding text range.

One further downside is that Microsoft's text processor is unable to handle a larger number of comments. In Word, all comments are displayed on the right-hand side and sorted by the position of the text selection within the document. An active comment does not displace other comments but rather stays on its fixed position in the sidebar. Therefore, in a document with a large number of comments, the corresponding text section may not be visible on the display anymore (see Figure 3.2). In order to find the associated text section the user has to select the comment and scroll towards the text section. Since the text section

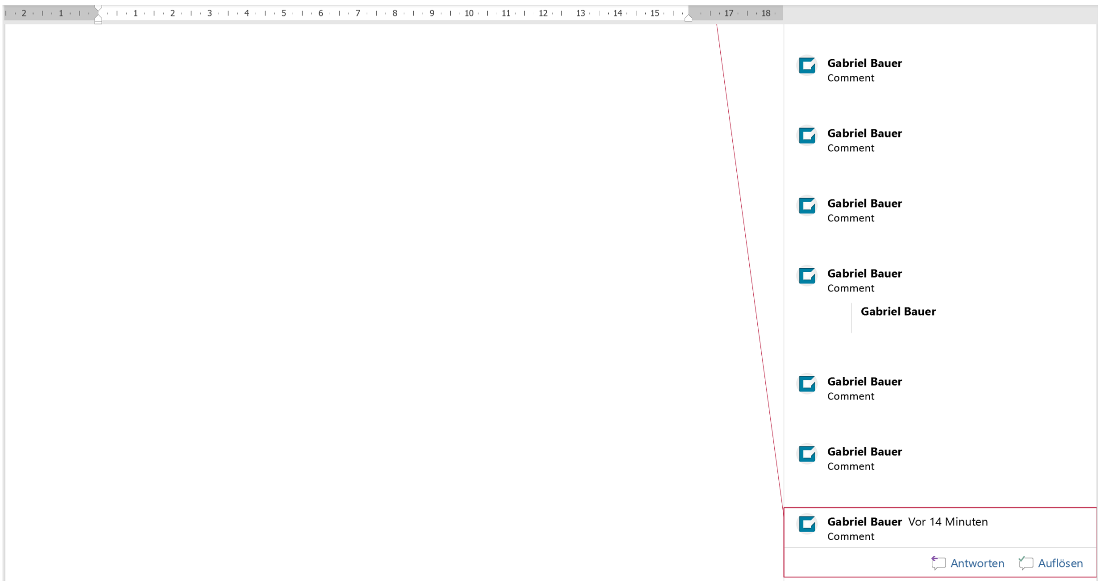


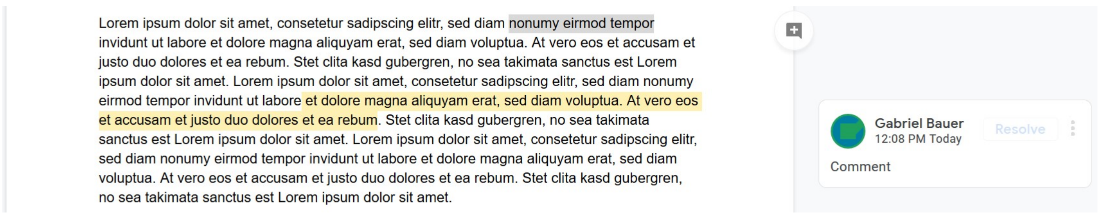
Figure 3.2: Screenshot of a document in Microsoft Word illustrating that in case of a large number of comments the corresponding text may not be visible.

and comment are connected by a line, the user knows in which direction to scroll. However, as soon as the text selection is within the viewport, the comment is not visible anymore. Hence, the handling of multiple comments can be a frustrating user experience.

3.3.2 Google Docs

Google Docs is a word processor similar to Microsoft Word and was released (without a beta status) in 2009[6]. Google Docs originated as an web-based application. Today it is also available as mobile app. Compared to Microsoft Word there is no standalone desktop solution (with the exception of Google's own operating system Chrome OS).

The general commenting UI is similar to Microsoft's solution, but has some minor but important improvements regarding UI best-practices. A comment can be created by simply selecting a text range. A small comment button appears on the right-hand side of the document (Figure 3.4). This approach is advantageous, since the process of creating a comment does not rely on the user recalling the button within the menu, but rather on recognizing the comment symbol. Additionally, there is also a button to add a comment within the menu under "insert". Since selecting a text range to add a comment still relys on recall - the user needs to know that a text selection will enable a comment button on the side - the additional button within the menu is convenient.


Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Gabriel Bauer
12:08 PM Today
Comment

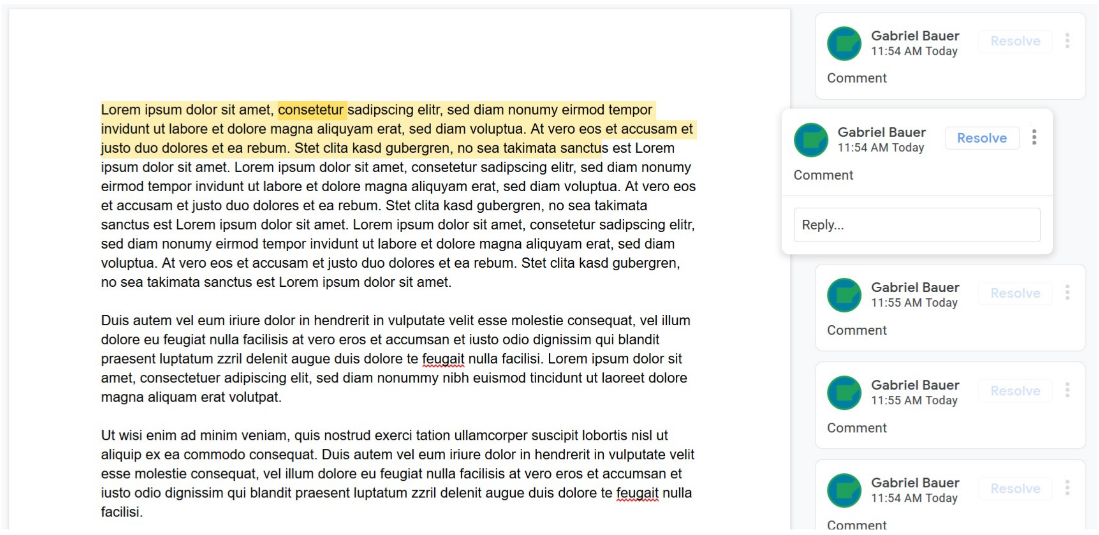
Figure 3.3: A screenshot from Google Docs showing the comment button to add a comment as well as the highlighting of a text referenced by a comment.

All comments of a document are also shown on the right-hand side. As soon as a text range is referenced by a comment, it is permanently highlighted as shown in Figure 3.4. For a user, it is therefore easy to identify text sections which are referenced by comments. Moreover, comments can be selected either by clicking on the text section or on the comment on the right-hand side. The active comment is always displayed directly next to the text section. Other comments referencing text sections near the active comment are displaced to the top or bottom.

While Google Docs is able to handle a large number of comments better than Microsoft Word, it also has its problems. A single text range referenced by

[6]https://googleblog.blogspot.com/2009/07/google-apps-is-out-of-beta-yes-really.html

multiple comments has no seperator between the different text ranges as shown in Figure 3.4. Moreover, hovering over on comment does not highlight the text range - only by clicking on a comment the text range will be highlighted. For a user it is therefore difficult to identify which comment references which part of the document. Another problem is, that active comments will displace other comments. While this helps to prevent that both the comment and text range are rendered within the viewport, it also makes navigating comments difficult. If e.g. the last comment is selected, all other comments are displaced to the top. Those displaced comments may be hidden, if its the start of the document. Since it is not possible to scroll through the comments, the first comment can only be reached by clicking on the corresponding text or by clicking on the top-most comment until the first comment is reached. It is not visible which of the texts belong to the first comment, making the first approach not intuitive. Long comment threads will worsen the situation as it will fill up the viewport with replies and show even less comments.



Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.

Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi.

Gabriel Bauer 11:54 AM Today Resolve
Comment

Gabriel Bauer 11:54 AM Today Resolve
Comment
Reply...

Gabriel Bauer 11:55 AM Today Resolve
Comment

Gabriel Bauer 11:55 AM Today Resolve
Comment

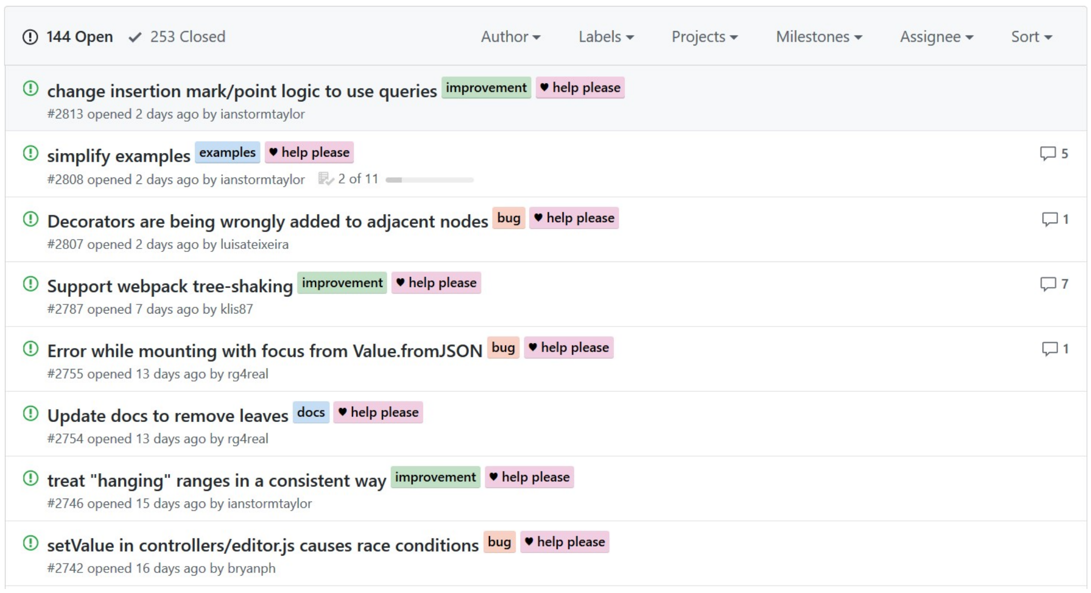
Gabriel Bauer 11:54 AM Today Resolve
Comment

Figure 3.4: Screenshot of a document on Google Docs showing many comments referencing a few lines at the beginning.

3.3.3 Github

As mentioned previously, GitHub is a web platform for collaborative software development and built on-top of the VCS git. While GitHub focuses on source code and Sweble on documents, both share a similar collaborative concept. Therefore, I analyzed how developers collaborate on open source software projects on GitHub. There are two main features within the web interface which are used to discuss aspects of the source code.

First of all there are issues, which are part of every repository on Github. An issue does not reference a specific file within the repository, but rather each issue is specific to one repository. Issues are a single-threaded discussion, to which any user with read-access to the repository can participate. As the term issue implies, they were formerly used to report issues or bugs in the software. However, looking at community driven open source projects, the development process is often organized via issues. Ideas and problems are discussed in an issue before an effort is made. Furthermore, user-defined labels can be assigned to issues, which helps to categorize the issues. Labeled issues are also used to organize and manage community driven projects. Issues can for example be marked under a label ”help wanted”, which is a way of organizing the community’s efforts. An example is shown in Figure 3.5, in which issues are organized with labels. In summary, issues can be used beyond simple discussion, but rather as a way of organizing collaboration and to discuss ideas, problems or improvements of the project.



change insertion mark/point logic to use queries `improvement` `♥ help please`
#2813 opened 2 days ago by ianstormtaylor

simplify examples `examples` `♥ help please` — 5
#2808 opened 2 days ago by ianstormtaylor — 2 of 11

Decorators are being wrongly added to adjacent nodes `bug` `♥ help please` — 1
#2807 opened 2 days ago by luisateixeira

Support webpack tree-shaking `improvement` `♥ help please` — 7
#2787 opened 7 days ago by klis87

Error while mounting with focus from Value.fromJSON `bug` `♥ help please` — 1
#2755 opened 13 days ago by rg4real

Update docs to remove leaves `docs` `♥ help please`
#2754 opened 13 days ago by rg4real

treat "hanging" ranges in a consistent way `improvement` `♥ help please`
#2746 opened 15 days ago by ianstormtaylor

setValue in controllers/editor.js causes race conditions `bug` `♥ help please`
#2742 opened 16 days ago by bryanph

Figure 3.5: Issues filtered by the label ”help please” within the community-driven open source project ianstormtaylor/slate on Github.

The second feature, which is used for collaboration, is the review process of source code changes. As discussed in section 3.1, a user, who is not a collaborator of a project, needs to fork the project and create a merge request. If a user opens a merge request, the collaborators have to review the changes. Every Github user with access to the repository is allowed to discuss the merge request. Each merge request includes a single-threaded discussion similar to issues. Furthermore, within each merge request, developers can comment on specific parts of the source code. This feature is therefore similar to the commenting solutions in

Microsoft Word or Google Docs. However, in software development one line usually contains only one program statement. Github's commenting feature therefore does not reference a text range but rather a single line within the source code.

4 Architecture and Design

As the commenting functionality should be implemented into Sweble, the commenting UI has to fit the application. In order to design the commenting functionality, the structure of Sweble and its collaborative workflow is described below.

4.1 Sweble Platform

As outlined above, Sweble applies the concept of distributed VCSs (and especially Git) onto the wiki concept. Since Sweble is made for general usage, its users usually do not have a software development background. Some of the concepts are therefore hidden within the backend and not visible to the user.

Similar to Github, Sweble organizes documents in repositories. However, they are referred under the broader term **project**. Each project can contain arbitrary **documents**. Documents cannot be organized hierarchically as it is possible in Git repositories. Projects rather contain a list of documents. Documents can be added, removed and modified through commits. The possible contents of a document are mostly defined by the functional range of Wikipedia's contents. Sweble is built on top of the Wiki Object Model (WOM). It is a high-level representation of the markup language Wikitext, which is used on Wikipedia. Therefore, WOM includes all content types which are expressable in Wikitext.

Collaboration is achieved solely through forks. It does not offer the ability to clone a project or to create branches. Each fork creates a new branch of commits, which is visualized in Figure 4.1. The user forking the project will become the owner of the fork and therefore has full access to its contents. Similar to the workflow on Github a user may propose changes to a document through a merge request. The owner of the original document has to review the changes and may accept or rejected them. Sweble offers a more powerful merge process compared to Github or Gitlab, since changes can be merged selectively. Therefore, the original project owner is able to decide which changes to accept, rather than to ask the merge request owner to change the merge request.

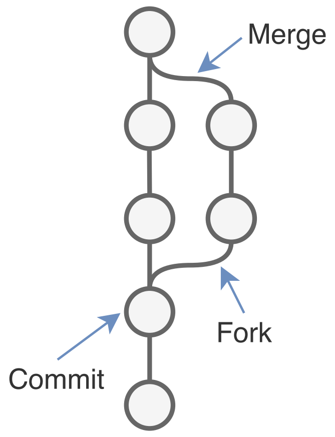


Figure 4.1: A visualization of a commit sequence, including a fork and merge.

The commenting functionality of this thesis can be seen as a starting point for collaboration. Similar to the workflow used on GitHub, the comments within the documents can be used to organize and manage collaboration. Ideas can be discussed with the owner before an effort is made to apply those modifications. One important aspect that emerges from the workflow of Sweble is that comments should be sustained on forks. In order to apply modifications discussed in a comment, it would be useful to have the comments within the fork as well. Therefore, the ideas discussed in a comment can be looked up without switching to the original project.

4.2 Commenting Functionality

The commenting base functionality is similar to Google Docs' and Microsoft Word's approach. A comment is implemented as a single-threaded discussion, which references a text range. Furthermore, comments can be created by any user, who has access to the document. The user creating the comment will further be referenced as the owner of the comment. The initial comment contains the first comment entry. Users with access to the document may reply to the comment and thus create further entries. The comment entries are displayed in a chronological order with the first comment at the top.

The context of comments is shown in Figure 4.2. Each comment contains an arbitrary number of entries. Since the comments correspond to a specific text range in one document, a comment is only present at the context of this document. This approach is different as compared to the issues on GitHub, as issues belong to a specific project. A comment is bound to a specific document in a certain state. The state of the document is defined by the sum of commits that lead to the current contents of the document. As a given commit can exist in multiple forks, the comments for a document is referenced by the triple of project, commit and document.

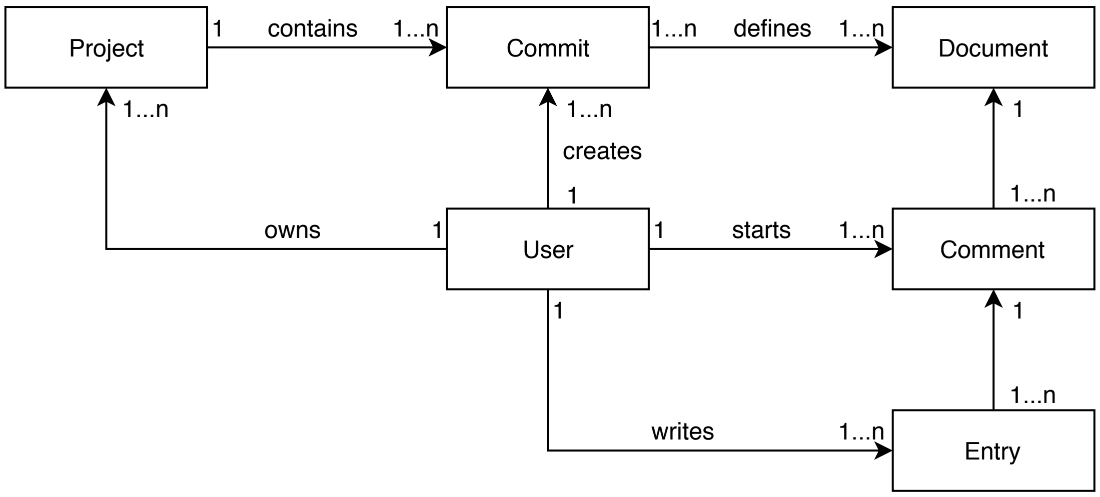


Figure 4.2: The context of the commenting functionality and its relationship to other objects on Sweble.

Each comment is in one of the four states **opened**, **closed**, **resolved** or **rejected**. Initially a comment is opened. The comment can be closed by the author in case it was opened by mistake. Furthermore, the project owner may mark a comment as resolved or rejected. Only if a comment is in the state opened, replies to the comment are enabled. In all other cases a message will be shown, explaining the current state to the user.

4.3 User Interface Design

Since one of the main tasks is to design the commenting UI, I created mocks for all UI elements. The initial mocks are presented below.

4.3.1 Creating a Comment

As shown in section 3.3, there are multiple ways to design the comment creation interface. Google Docs' approach, selecting a text range and showing an icon, is a good implementation regarding the UI best-practices. However, the final design was inspired by the UI of Medium[1]. As the user selects a text, a tooltip

[1]https://medium.com/

appears directly above the selected text presenting buttons to interact with the text range as shown in Figure 4.3. The tooltip above the selection catches the users eyes easier compared to an icon on the side of the document.

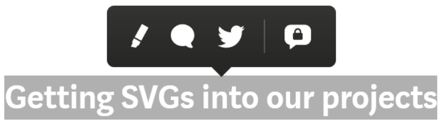


Figure 4.3: A screenshot from medium.com showing a text selection with a respective tooltip. The tooltip enables the user to create a comment.

In terms of this thesis' implementation, the tooltip contains two buttons (compare with Figure 4.4). One button in order to create a new comment referencing the current text selection. The other button in order to view comments. The button to view comments, will be enabled only if the current text selection overlaps with at least one text range referenced by a comment. If no text reference is present in the current text selection, the button is disabled. The initial mock uses a plus and eye icon, which were replaced in the final implementation with icons showing one speech bubble for the create and multiple speech bubbles for the view button.

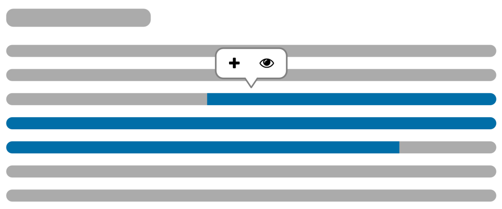


Figure 4.4: A tooltip will be shown if the user selects a text range.

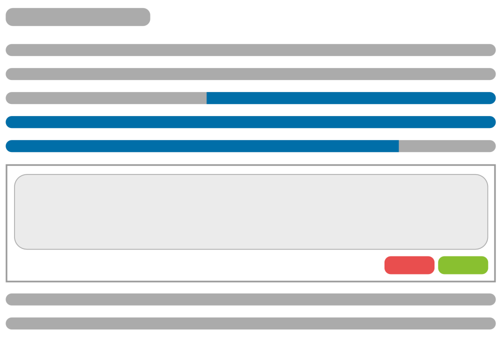


Figure 4.5: The form to add a comment is rendered below the text range.

As the create comment button is pressed, the comment creation form has to be displayed. There are multiple ways to integrate the creation into the UI. A simple implementation would be to use a modal. A modal (or popup) is a window overlaying and disabling the page's contents. However, a modal would deny the user to see the original text. In case the user has to recheck the document, the modal containing the comment form has to be closed and thus the already written comment is discarded. Another solution which is used by Google Docs and Microsoft Word is to display the comment form (as well as the comments) on the right-hand side. While this is generally a good place, it does not work on devices with narrow screens like e.g. mobile devices. A responsive design is not a requirement of this thesis. However, since mobile usage is increasing, the design should be usable on mobile devices in the future. Therefore the comment form is placed directly within the document, underneath the text selection as shown in Figure 4.5. The form will not overlay parts of the document but rather displace the following contents. This approach is different compared to the two word processors of Microsoft and Google. Only Github implemented a similar approach in their review UI for merge requests. Comments referencing source code are displayed beneath the corresponding code line.

4.3.2 Viewing Comments

Comments can be viewed directly within the document. The popup above a text selection contains a button to view comments. The button is only activated, if the current text selection overlaps with a text range referenced by a comment. As the button is clicked, the comment is rendered into the document.

The comment is rendered using the same approach as the comment form. It will be displayed beneath the text range as shown in Figure 4.6. The text referenced by the comment is highlighted. The comment view consists of the entries in a chronological order. Below the entries is an input field in order to reply to the comment. In case that the comment has a large number of replies, the replies can be paginated. Pagination is a technique used to divide a larger number of elements into pages and is commonly used on the web.

Figure 4.7 shows the design of a comment with many replies. It does not use a classic pagination approach but rather an approach used by GitHub on their issue page. In the classical pagination only the oldest few comments are shown to the user. If the user clicks on the next button, the current comments are replaced with some newer comments. Github approach always displays the oldest and newest comments with a load more button in-between. If the user clicks on the button, more comments are added to the page. This approach offers a better handling, since comments are never displaced with other comments. Furthermore, the initial state shows the oldest and newest comments which are often the important parts. The first comments usually explain the problem, while the last comments may offer a solution or compromise.

One of the main challenges is the visualization of text ranges referenced by comments. Microsoft Word does not display any highlighting where comments reference text. The user is therefore unable to efficiently determine if a specific text range has been commented. Google Docs highlights all text ranges by displaying a coloured background. This approach has the disadvantage that a larger number of comments cannot be visualized since there is no distinction between one or multiple comments on the same text range.

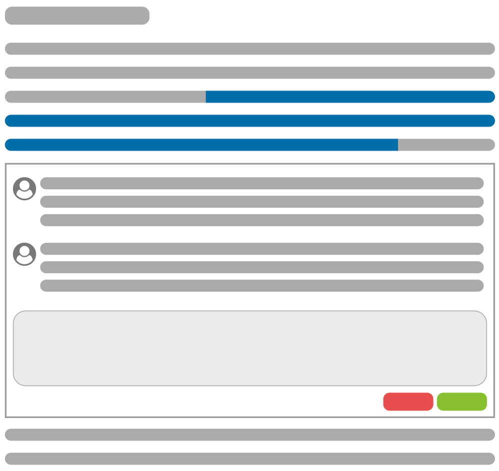


Figure 4.6: A comment with only a few replies, rendered underneath the referenced text range.

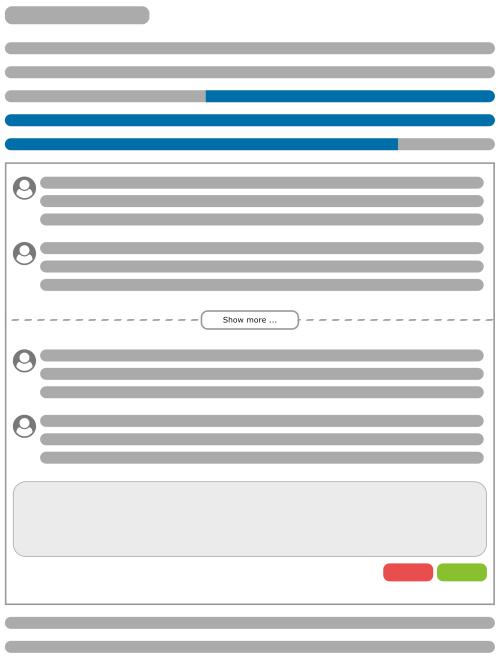


Figure 4.7: A paginated comment thread displaying some of the oldest and newest entries. The button in-between both entry sections loads the remaining entries.

The initial idea was therefore to display a comment counter on the side of each block as shown in Figure 4.8. By clicking on the counter, the first comment would be rendered onto the page. Above the comment view are two buttons in order to navigate to the next or previous comment. However, this approach also has its drawbacks. The user will not be not able to see directly which texts are commented. Furthermore, if a document contains larger text paragraphs or small blocks, the implementation would need to split big and merge small blocks so that the comment counters have a uniform interval.
Another solution would be to work on a per line basis as Github does. However, there is a technical challenge that a line's text content is dependent on the display width. On a narrower display, text will be wrapped into new lines automatically. Text lines can therefore not be referenced clearly. Furthermore, if a comment references a whole paragraph, a comment icon or counter would be shown on each line of the paragraph. The user will not be able to determine if there are multiple comments (one per line) or if there is one comment covering the whole paragraph.

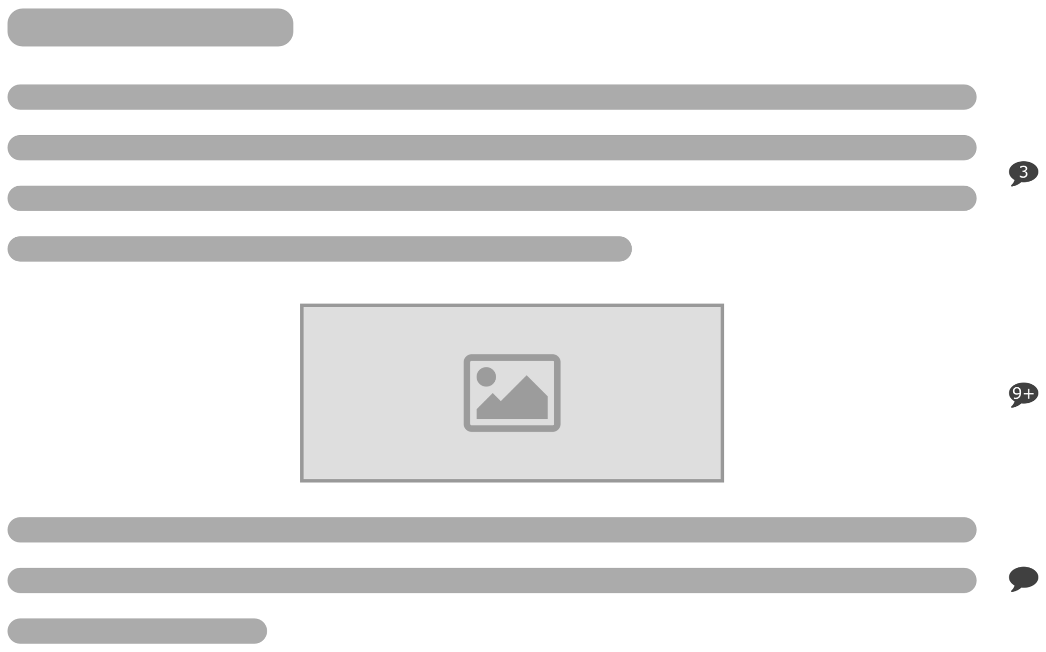


Figure 4.8: Initial design mock of the comment counters displayed on the side of blocks, which show the number of comments for a specific block.

It is debatable whether a comment counter offers a better user experience than Google's solution to highlight the text ranges, since the user looses the ability to distinguish whether a specific text range is referenced by a comment. Therefore, I abandoned the idea of comment counters and the final implementation uses a similar concept as Google's implementation. Instead of colouring the background, which may also be part of the editors capabilities, all referenced text ranges are underlined with dotted line. A dotted line is not included in the Sweble Document editor's capabilities and therefore offers a unique appearance.

4.4 Comment Text Selection

A comment always references one specific text range as it was presented in section 4.2. A data model has to be developed in order to store the text reference. The text range corresponds to specific WOM nodes as presented in the following.

4.4.1 Wiki Object Model

Sweble uses the Wiki Object Model (WOM) to store documents (Dohrn & Riehle, 2011b). The WOM represents Wikitext, which is the markup language used on Wikipedia and initially proposed because wikitext did not offer a high-level representation. However, a high-level representation is needed in order to parse a wiki's contents easily and make it machine-accessible. Furthermore, the semantics of wikitext were only defined implicitly by the software behind Wikipedia. WOM was therefore developed to provide the wiki community with a standardized format to interact, store and share contents. The structure of WOM documents is represented using an abstract syntax tree (AST) and will be further referenced as WOM-tree in this thesis.

The WOM is similar to the Document Object Model (DOM), which represents HTML documents. Every modern browser uses the DOM in order to connect the web-page to "scripts or programming languages by representing the structure of a document"[2]. Similarly to the WOM, the DOM is also structured as a tree. The root of the tree in both object models is a document node. Each node may be the parent to multiple child nodes. An exception are text nodes, which cannot contain child nodes. If a node has no children, it is called a **leaf node**. A text node is therefore always a leaf node.

To transform Wikitext into WOM a Wikitext parser is nessesary, e.g. the Sweble wikitext parser (Dohrn & Riehle, 2011a). Since WOM was developed to represent Wikitext, it includes all content types found in Wikitext. WOM is able to represent any Wikitext formatted document. Since Wikitext uses some standard formattings like bold, italic, etc. many elements can be found in the HTML standard. However there are also certain elements like e.g. an internal link, which are only available in Wikitext and WOM.

The WOM can be serialized into a XML representation as shown in Listing 4.1. On Sweble the XML representation is only used to transfer the WOM to the frontend. The resulting WOM-tree of Listing 4.1 is shown in Figure 4.9. The XML serialization does not contain the root node, the WOM document, which is only present in the WOM.

[2]https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model

```
<article>
 <body>
  <p>
   <text>...</text>
   <b>
    <text>...</text>
   </b>
  </p>
  <p>
   <text>...</text>
  </p>
 </body>
</article>
```

Listing 4.1: A WOM document serialized as XML. The node's attributes and texts are stripped for a better readability.

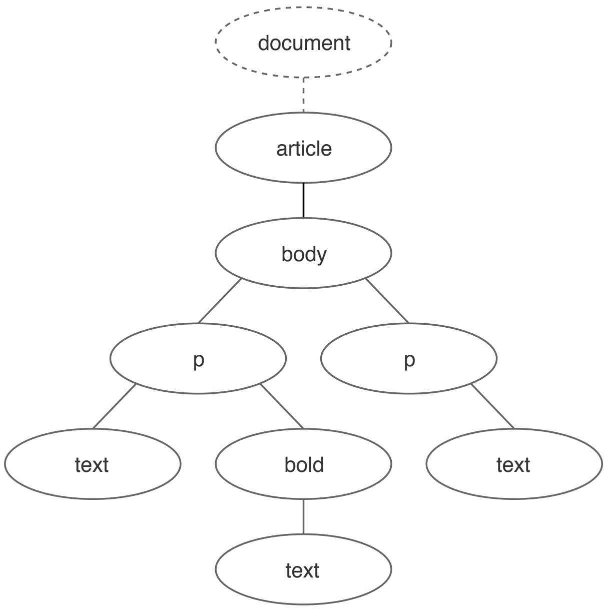


Figure 4.9: The corresponding WOM-tree to the XML serialization shown in Listing 4.1.

4.4.2 Referencing a Text Range

A text range can be expressed by the combination of one start and endpoint within the document. Everything between the two points is within the text range and would therefore be part of a comment's text selection. There are two solutions how the start and endpoint of a text range can be referenced.

Marker Elements

A new WOM element can be defined to mark the start and beginning of a text referenced by a comment. The element would not be rendered and therefore not be visible to the user. The marker element would be a leaf node and thus not contain any children. Per comment two markers have to be inserted into the WOM-tree. Each marker holds a unique identifier referencing the respective comment. The text range can be reconstructed by traversing the tree with a depth-first search. The first encountered marker of a specific comment ID is its starting point and the second marker with a matching ID is the end of the text range.

However, a new comment should not result in new revision of the document. Thus, the markers cannot be inserted on comment creation but rather can only committed if changes are made to the document. Therefore, before the comments are stored within the WOM, they have to be stored separately in a database.

This approach would make the requirement to store the comment data within the document easy to fulfill. The marker element could additionally contain the comment information and all comment entries.

Referencing WOM Nodes

The alternative approach relies on referencing WOM nodes or rather a position within certain nodes. Both, the start and end point of a selection, can be mapped to a specific text node within the WOM-tree. However, it is not sufficient to map the selection points to WOM nodes. Rather the exact position within the node is needed. Therefore per point, an offset is used, which describes at which character within the node the selection starts or ends. The offset is the index into a text node's string. One special case is if the selection is contained in a single text node. Then, the start and end node can be identical and only the offset differs. A text range in which the start and end node as well as both offsets are identical, represents a collapsed text range and is not considered as a valid text reference for a comment. A collapsed text range or collapsed selection can also be seen as the cursor within the document. It has a width of zero characters.

As it will be presented in the implementation chapter in subsection 5.2.4, the browser exposes a selection interface, which is similar to the selection interface used in this approach. The selection interface of the DOM is defined over an anchor and a focus node and two offsets for each node. The DOM selection interface does not use the terms start and end, since the anchor node always references the node from which the selection started and the focus node references the node where the user ended the selection. If the user selects a text and moves the selection further to the right, the anchor node comes first in the DOM tree. However, if the user moves the selection to the left, the focus node comes first in the DOM tree. Since the anchor and focus points are not relevant, they are normalized into start and end points. The start point corresponds to the node, which can be found first within the document.

While the offset is a simple number, referencing the nodes is more challenging. In both, the DOM as well as the WOM, a node has no unique identifier. There are mainly two solutions to reference a node. One approach is to describe the path from the root-node to the specific node. This can be done by a list of numbers. Each number describes which child-node to navigate to in order to reach the referenced node. To reference the bold text node in Figure 4.10, the path can be described by "0, 0, 0, 1, 0". To find the node, one always navigates to the first child (0) expect for the paragraph-node, where one navigates to the second child, which is represented by the numeral 1. Another approach is to give each node a consecutive number. A depth-first search is used to apply the numbers, so that the numbers correspond to the order of the rendered content. A node with a lower number is rendered before a node with a higher number. This approach has the main advantage that a number is simpler and needs less space than an array of numbers. The bold text node in Figure 4.10 is referenced by the number 6. Since the path IDs do not offer any advantages over the node IDs, the path IDs are not considered further. The resulting selection interface is shown in Listing 4.2.

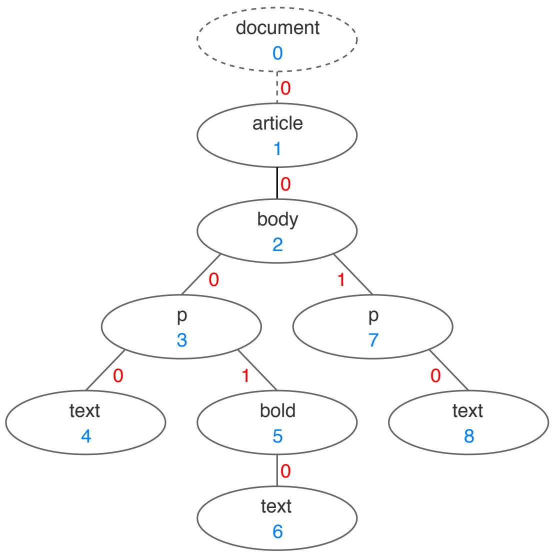


Figure 4.10: A WOM-tree with its nodes numbered by a depth-first search in blue and the child index in red.

```
type Selection = {|
  startNodeId: number,
  startOffset: number,
  endNodeId: number,
  endOffset: number,
|};
```

Listing 4.2: The selection interface used to reference a text range through a start and end point.

Comparison of Marker Elements vs. Node IDs

A major disadvantage of the last approach is that the node IDs are not stable, since the node IDs are relative to a specific WOM-tree. If changes are introduced to the document, the tree structure will also be altered. Depending on the position of a change in the document, the node IDs generated by the deep-first numeration may get invalid. A given node ID may still reference a node. However, the referenced node might have changed. An example is the insertion of a node at the beginning of the document. Before the insertion, the node ID 3 would correspond to the first child-node of the body. As the first node is displaced with the inserted node, its new node ID is 4. Therefore, each time a document with comments is altered, the text references have to be mapped to the changed document. The start and end node IDs need to be updated so that the comment references the original text range. For example, if a single node is inserted at the beginning, both the start as well as end node IDs need to be increased by 1.

The marker elements are unaffected by modifying the document as long as both the start and end markers are not deleted. While this is a major advantage compared to the node IDs, it also demands that every tool that interacts with the WOM has to keep the markers intact and preserve them. This is especially true for the visual editor. It has to be extended in such a way, that markers cannot be deleted accidentally by a user. For example, if a sub-tree containing a marker is deleted, the marker has to be preserved. Implementing this behaviour into the visual editor requires in-depth knowledge of the visual editor. Furthermore, the marker approach also introduces a tight coupling between the content and the comments. In cases where the comments are not needed (e.g. diff view or pull request reviews), the markers are transferred unnecessarily. In addition, the mark approach has no consistent storage of its markers. Since the marker cannot be embedded directly into the WOM as the comment is created, it will still rely on the node IDs to store not yet embedded comments. Therefore, comment markers are present in two places. All current comment text ranges are stored in a database via their node IDs and offsets, while all older comment text ranges are embedded within the WOM. In conclusion, I chose to reference the nodes with the help of the second approach by using node IDs and offsets.

The disadvantages of the marker approach also apply to the requirement to store the comment data directly in the document. Therefore, I decided to drop the requirement and store all comment data separately to the document in respective database tables.

Example

The title of this thesis is used as an example, in which the word "Functionality" is formatted bold. Three WOM text nodes are needed to encapsulate the text, since the bold text has to be wrapped in an additional bold node. An exemplary text selection is shown in Figure 4.11. Figure 4.12 presents the corresponding WOM representation with the text selection highlighted. Parts of the first and last text nodes as well as the whole middle text node are in the selection. The resulting WOM tree is presented in Figure 4.13. As outlined above, the node IDs are computed with a depth-first search. The start node corresponds to ID 4 and the end node to ID 7. The start offset 10, shown in Listing 4.3, corresponds to the position of space after the word "implement". The end offset is computed similarly.

Implement Commenting **Functionality** in Sweble Documents

Figure 4.11: An exemplary text selection on this thesis title.

```
<article>
 <body>
  <p>
   <text>Implement Commenting </text>
   <b>
    <text>Functionality</text>
   </b>
   <text> in Sweble Documents</text>
  </p>
 </body>
</article>
```

Figure 4.12: The WOM structure of Figure 4.11 with the selection highlighted in blue.

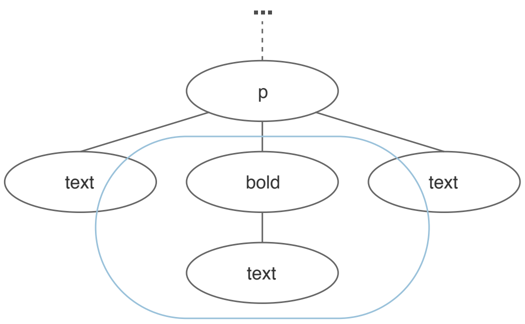


Figure 4.13: The resulting WOM tree to Figure 4.11

```
{
    startNodeId: 4,
    startOffset: 10,
    endNodeId: 7,
    endOffset: 9
}
```

Listing 4.3: The selection object describing the selection shown in Figure 4.11

4.5 Changes to a Document

Modifications to a document have multiple consequences for this thesis' commenting functionality. Not only the node IDs can get invalid, but rather the contents within a text reference can change. If for example all referenced nodes are deleted, the reference would be lost. It is therefore not sufficient to just map the node IDs in case of a change, but rather to analyze the changes in greater depth. As changes are made to a document the corresponding WOM-tree will be modified. To determine the changes,the current and new WOM-tree are available.

However, nodes present in both trees have no connection between each other. A diff algorithm is needed to determine which nodes have been inserted, deleted or modified.

4.5.1 HDDiff

To detect changes between two trees, Sweble uses the diff algorithm HDDiff (Dohrn & Riehle, 2014). HDDiff analyzes the changes, finds a mapping between the nodes present in both trees and returns an edit script. The edit script describes which transformations have to be applied to the original document in order to get the new document. The edit script consists of a list of different operations. Each operation describes a certain type of change that can be applied to the document. Five different operation types are used to describe any modification to a document.

Insert & Delete

The insert as well as the delete operations are the two most basic transformations. All other operations can theoretically be expressed solely through insert and delete operations. An insert operation adds one single node to a specific position within the parent's children as visualized in Figure 4.14. Contrary to the insert, the delete operation removes one node from the tree (Figure 4.15).

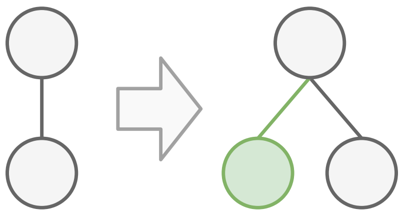


Figure 4.14: Visualization of an insert operation.

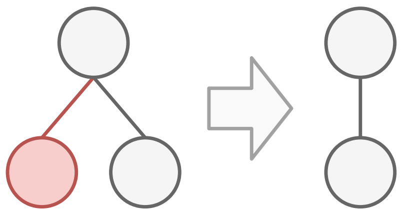


Figure 4.15: Visualization of an delete operation.

Move

The move operation relocates a subtree within the document as shown in Figure 4.16. Compared to an insert or delete operation it may affect more than one node. A move operation can be expressed as a delete and insert operation. It is therefore mainly an improvement to detect movements within a document.

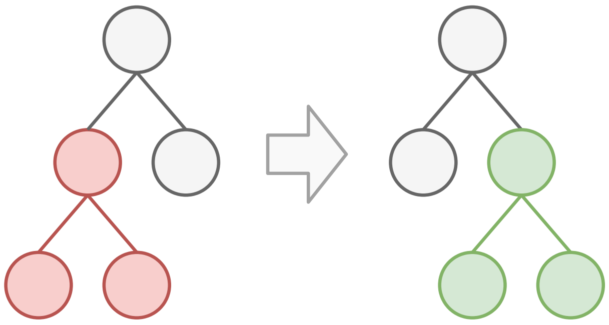


Figure 4.16: A move removes and inserts some nodes at different positions.

Split

A split operation can only affect a single text node, since it splits its text content into two separate nodes. It is used e.g. to improve the move operation detection. Since the move operation only applies to nodes, it cannot represent the movement of a text passage. The split operation is able to extract a moved text passage into its own node. Thus, a move operation is then able to describe the text movement.

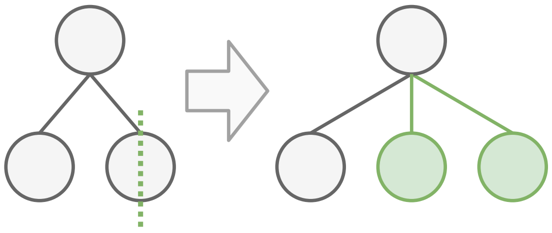


Figure 4.17: The split operation splits a text node at a specific position.

Update

Since the update operation does not affect the tree structure, it differs from all other operations. The update operation rather changes an attribute or the text of a node. In case text should be added to an existing text node or be modified, the node is updated and not replaced. For text updates the operation contains the old and new text. For the commenting functionality an additional algorithm is used to further analyze the changes. To compare both versions of the text, an implementation[3] of the Myer's diff algorithm (Myers, 1986) by Google is used. It produces a list of inserted, deleted and equal characters or rather words (depending on the settings). The update operation of HDDiff can therefore be seen as additional insert and/or delete operation on the scope of text nodes.

[3]https://github.com/google/diff-match-patch

4.5.2 Implications for Texts Referenced by Comments

Changes to a document may alter the content of the referenced texts. If comments exist while many modifications are applied, references may no more represent the original text. Therefore, the changes have be analyzed and the text ranges adjusted appropriately. The goal is that the text reference resembles the original text as close as possible. Since the five operations are able to describe any modifications to a document, they will be used to categorize and analyze the possible effects on the referenced text ranges.

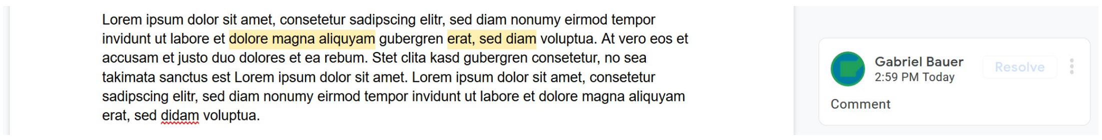
A **split** operation within the referenced text is irrelevant for its contents, since it only affects the structure and not the contents. However, if a split operation applies to the start or end node, it has to be validated whether both generated nodes are within the text range. The latter depends on the selection offsets as well as the split position. If the start node is split at a position after the selection start, both nodes are within the text range. However, if on the other hand, the start node is split before the selection start, the first node is not within the text range and thus the node and offset have to be corrected. The node ID is increased to represent the second half of the split and the offset is subtracted by the split position. The impact to the end node is similar to the start node. It has to be checked, whether both nodes are within the selection.
The **delete** operation is similar to the split operation. If nodes are deleted within the text range no changes are required. Only if an start or end node is deleted, the text range has to be adjusted. If the start node gets deleted, the new start point will be the first character of the next sibling. On the other hand, if the end node gets deleted, the text range's end will be set to the last character of the previous sibling. Furthermore, if the text range only consists of one node and this node is deleted, the reference will be lost. A comment with a lost reference will not be visible in the document anymore.

The remaining operations insert, update and move are able to split the text range and thus produce fragments. An **inserted** node within the text reference is not part of the original text, hence the reference consists of two parts - the nodes before the inserted node as well as the nodes after the inserted node both correspond to the original reference. However, in practice not all inserted nodes will produce fragments. Formatting a text part bold creates a bold node and moves the text into the bold node. Since the text node stays identical, the bold text is still part of the reference. Therefore, only nodes that actually contain content like e.g. text, images or similar nodes will produce fragments.
Moreover, while the **update** operation does not affect the tree structure, it can contain text insertions in the text reference. Similar to the insert operation, an update is therefore able to introduce fragmentation. While one insert operation only produces two parts, an update operation may introduce a large number of

fragments depending on the number and position of text insertions. A special case is the update of the edge nodes. If text is inserted in front of the start of a comment, the start offset has to be increased by number of characters inserted in front. Text inserted in front of the selection end, will increase the end offset. Last but not least, the **move** operation is also able to produces fragments. As parts of a text reference are moved, two fragments are generated. Only if the whole text range is moved, there will be no fragments.

4.5.3 Fragmented References

Fragmented text references have implications towards the user experience and implementation complexity. Microsoft has therefore not allowed fragmented text references in Word. Insertions within a text reference are treated as part of the reference. Copying and pasting parts of a reference will also not produce fragments. However, moving the whole text reference will preserve the comment and corresponding text reference. Google Docs implemented support for fragmented references as shown in Figure 4.18. While typing within a reference will not produce fragments, copy & paste will. Pasting into a reference, moving parts of a reference or simply copying referenced texts creates fragments.



Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam gubergren erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren consetetur, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed didam voluptua.

Gabriel Bauer
2:59 PM Today
Comment

Figure 4.18: Screenshot of Google Docs showing a fragmented reference.

Comment references which undergo many modifications can get heavily fragmented. In the worst case, fragments can be scattered across the entire document. In this commenting functionality, text references are highlighted by underlining them. Therefore, a user would not see the difference between a fragmented reference and multiple references. Moreover, if fragments are spread over a larger document, the scattered references would not be visible within the browser's viewport and thus only offer limited advantages. In order to enhance the user experience it would be necessary to implement some sort of heuristics. Fragments which are too small or which are moved to a remote part of the document, should be neglected. Since more research is needed to implement a good solution and algorithm for fragmented references, I decided to not use fragmentation.

My approach is therefore similar to Microsoft Word's solution, however the handling of move operations was significantly improved. If part of the reference in Microsoft Word is cut and pasted, the comment will always reference the initial, uncut part. This may result in misleading results. In case that the majority of the text reference is copied, the comment will still reference the small part which was left intact. So, instead of referencing the initial text range, my approach references the bigger of both parts. The algorithm will compute all fragments and choose the biggest fragment, which is then used as new text reference.

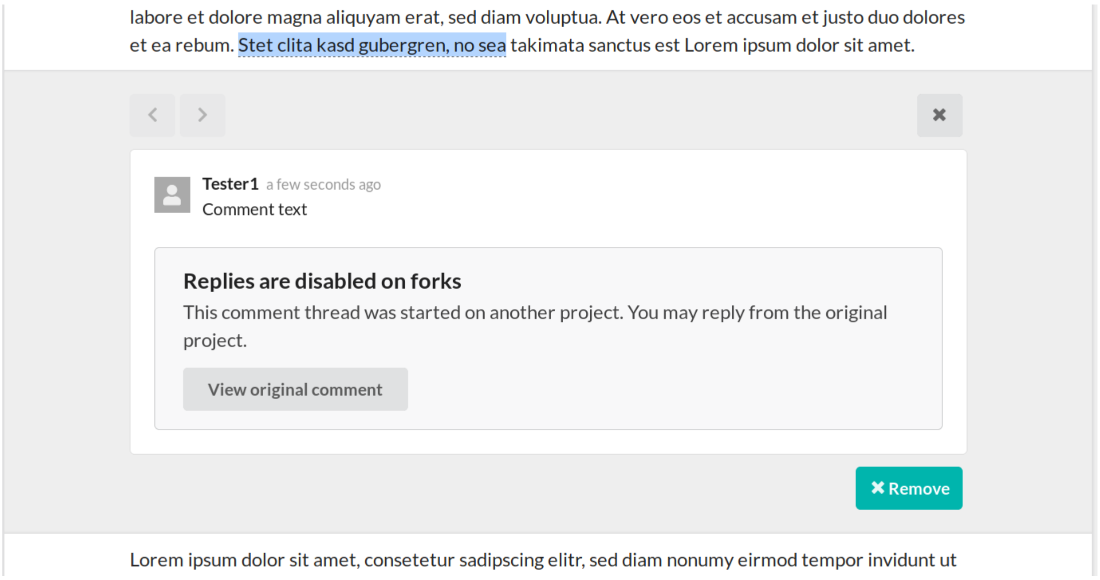
4.6 Comments in a Multi-Synchronous Environment

Since Sweble is a multi-synchronous environment, the commenting functionality has to include a concept of how comments behave in terms of forks and merges. There are different approaches how comments can be integrated. All solutions are technically feasible. It is therefore mainly a decision regarding the user experience and usability. In section section 4.1, I discussed that comments from the original project should be visible in forks, since the comments can be used to support the collaborative workflow. If a comment is forked, the entries can either be visible in both projects or only visible in the project, in which they were created. Of course, both comments would still contain all replies before the project was forked.

Displaying new replies only in the current project, would follow the principle of a multi-synchronous environment and apply the concept of branches to comments. As a project is forked, the comment and its entries would be forked. Replies (similar to commits) would only be visible in the corresponding project. This approach also implicates that comment branches should be merged if a project is merged. However, merging replies will degrade the meaning of the reply sequence, since a comment is a list of chronological entries. Merging two branches would mean to mix two time-lines together, effectively destroying the sequence of comment entries.

To avoid this problem, replies have to be visible in all projects. However, this solution has usability disadvantages. The referenced text ranges are prone to modifications and may be altered with new commits. The original project and its forks may be modified differently. The text reference in one project may be different compared to the same comment's text range in another fork. In a worst case, where two projects have been changed significantly, the referenced texts have no commonalities anymore. Users would debate based on different texts, which would hinder a collaborative discussion.

As a solution, I chose to disallow replies on comments which originate from a different project. Instead an info box is shown with the message, that the comment was created in a different project and to participate, the user has to navigate to the original project (compare with Figure 4.19). The info box contains a button in order to navigate to the original comment. This solution still allow users to follow the discussion without leaving the project. Comments can therefore still support the collaborative workflow. Furthermore, the user gets a better insight how comments work, since the UI communicates clearly where a comment originated.


labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Tester1 a few seconds ago
Comment text

Replies are disabled on forks
This comment thread was started on another project. You may reply from the original project.

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut

Figure 4.19: Screenshot of a comment in Sweble showing that comments are disabled within forks.

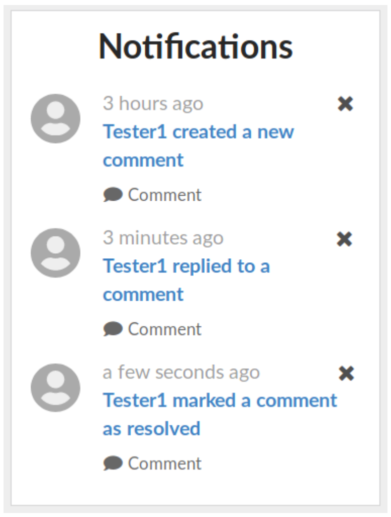
Within the original project, the owner can accept or reject a comment. Further, the owner of a fork should also be able to delete comments. The owner of the fork is able to close such comments. In addition, a state change in the fork should not affect the original project and vice versa. Thus, the state only corresponds to a comment in a specific project.

While the approach of forking comments with replies disabled does not have the challenge of merging comment entries, it still raises the question if comments originating within a fork are preserved in a merge. In Sweble, the user is able to only merge parts of a merge request. There is a merge UI, in which the owner of a project can select which changes should be merged back into the project. Comments referencing change which was not merged, should not be preserved. For all other parts, comments can be preserved. However, the owner of the project may not want to incorporate comments form the fork into the original project. A solution would be to extend the merge UI, so that comments from the fork can be merged optionally. This implies that a reviewer of a merge needs to review the comments in order to decide if they should be merged. Depending on the number of comments, this may be a time-consuming task. Furthermore, the workflow of Sweble predicts that forks are mainly created to introduce changes to a project. Thus, merging comments into the original project does not offer a distinct benefit. For this implementation, I chose not to preserve comments on a merge.

4.7 Notifications

Sweble includes a notification system which is extended and used by the commenting functionality. Comments will generate notifications for the project owner and for user who contributed to the comment. There are three types of comment notifications which are send once a comment is created, a user replied to a comment or as the state has been changed (Figure 4.20). All comment notifications reference the corresponding comment.

The URL to a specific comment contains the identifying triple of project, commit and document as well as the comment id. If the user opens a comment URL, the document as well as the referenced comment is shown. Since the URL contains the specific commit, the link can further be used to reference a past state of the comment. As a past state is viewed, both the document as well as the comment are shown as they were at the time of the commit. The notification always reference the newest state of a comment by using the keyword "master" instead of the actual commit. Internally, the "master" corresponds to the master branch, which always points to the newest commit within the project.


Notifications

3 hours ago
Tester1 created a new comment
Comment

3 minutes ago
Tester1 replied to a comment
Comment

a few seconds ago
Tester1 marked a comment as resolved
Comment

Figure 4.20: Screenshot of the notification system showing the three types of comment notifications.

GET	/comments/	Get comments
POST	/comments/	Create comment
POST	/comments/{commentId}/	Update comment state
GET	/comments/{commentId}/entries	Get entries
POST	/comments/{commentId}/entries	Create entry

Table 4.1: An overview of the commenting API.

4.8 Commenting API

Sweble uses a client server approach. The server has to expose appropriate endpoints, so that the frontend can create, retrieve and modify comments. The base path includes the identify triple of project, commit and resource:

/v2/projects/{projectId}/commits/{commitName}/resources/{resourceName}

This path is already used to e.g. retrieve the document's contents. All API endpoints shown in Table 4.1 are based on this base path.

5 Implementation

This chapter gives a broad overview of the implementation and highlights some of the challenges and used algorithms. All source code examples are excerpts, reduced to the essential parts and should be seen as pseudo-code.

5.1 Software Stack

Figure 5.1 gives an overview of Sweble's architecture which uses a server client approach. The frontend is a single-page application which describes a website

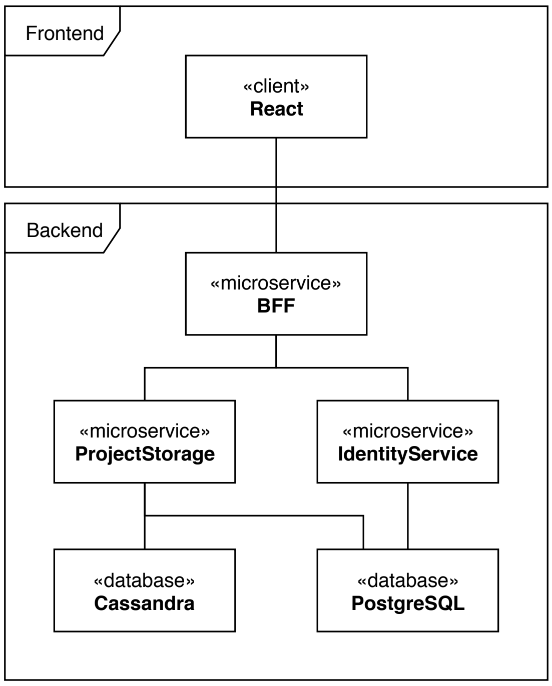


Figure 5.1: UML deployment diagram showing an extract of the Sweble's architecture.

that is loaded once and content changes are performed by modifying the DOM. In order to fetch additional data, the frontend performs REST requests to the backend. The backend is organized in microservices. The only microservice that is accessible from outside is the "Backend For Frontend" (**BFF**). It functions as a proxy and load-balancer. Requests can be transformed within the BFF before they are relayed to the appropriate microservice. The **ProjectStorage** service handles requests regarding projects, branches, commits and documents. It furthermore stores the WOM documents in the database Cassandra. The **IdentityService** is responsible to handle all requests regarding the user.

In this work, the Frontend, BFF, ProjectStorage as well as PostgreSQL have to be extended. The main modification is the UI implementation in the frontend. Furthermore, the BFF has to be extended so that it offers the corresponding API endpoints to the frontend. The actual backend logic will be implemented into the ProjectStorage. The PostgreSQL database scheme has to be extended in order to store the comments.

5.2 Frontend

5.2.1 Used Technologies and Libraries

React is a open-source JavaScript frontend library developed by Facebook. Some if its concepts are presented here, since they are important for parts of the UI implementation. In order to build a rich user interface, React uses a hierarchical component architecture. Each component is able to render HTML (e.g. a button) as well as other React components. Conceptually, React components are similar to function, since "they accept arbitrary inputs [...] and return React elements describing what should appear on the screen"[1]. Since React components may contain other components, a React application can also be seen as component tree. There is one root component and any number of descendant components. In order to be able to render HTML and other components easily, React uses JSX, a syntax extension for JavaScript. The syntax is similar to HTML and can be used directly within JavaScript as shown in Listing 5.1. React components can either be written as a simple JavaScript function, which returns JSX, or as a class. A React class has to implement the *render* method which returns JSX as well. The differences between both approaches are minimal. With the newest React version 16.8 functions as well as classes offer similar functionality. Most of my components were implemented as classes.

[1]https://reactjs.org/docs/components-and-props.html

```
function HelloWorldComponent(props) {
    return (
        <div>
            <h1>Hello World!</h1>
            <AnotherReactComponent />
        </div>
    );
}
```

Listing 5.1: A minimal React component using JSX to render HTML as well as React components.

Out of the box React only supports unidirectional data flow. Data is propagated downwards in the component tree. A component can provide data to its children via so called **props**. The **props** are similar to function parameters, since they are the inputs for the component. In order to return data from a child to its parent (upwards), the parent needs to pass down a callback function via **props**. The child is then able to invoke this callback and pass data upwards in the component tree. Components also need the ability to store state, e.g. whether a popup is shown or not. In order for components to be stateful, React offers components the ability to define their own **state**. The **state** is part of a component's instance and only accessible by itself. The state can be passed to its children via **props**.

A problem that arises from the directional data flow is that communication between components of different subtrees is cumbersome. Without an additional library, the data has to be lifted up towards the first common ancestor and then down into the right component again. A library solving this issue (and other challenges) is Redux. It provides one single, central state container which acts as single source of truth (compare with Figure 5.2). In Sweble, it is mainly used to store the API requests and results as well as essential aspects of the page, e.g. the logged-in user or active project data. Each component is able to access the **store**

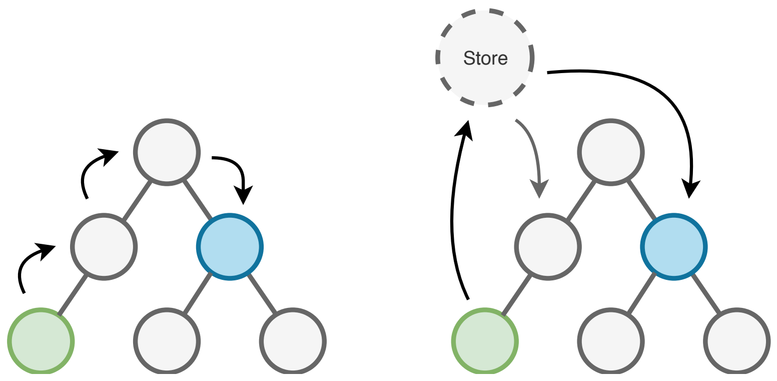


Figure 5.2: Visualization of the communication (arrows) between components (circles) in React with and without a central store like Redux.

(the state container) as well as to trigger modifications. The whole Redux **store** is immutable and usually organized hierarchically. There are different concepts how to structure the **store**. In this thesis the commenting functionality uses its own sub-store. Since the **store** is immutable, changes can not be made directly but rather through a specific mechanism. Components can dispatch **actions**, which will trigger a store modification. This concept is used by the commenting UI and will be described in greater detail in subsection 5.2.3.

Another concept of React is its lifecycle. Since large documents can contain many elements, components involved in rendering documents have to be optimized for performance. Understanding the React lifecycle is crucial for any performance optimization. There are two main phases in React's lifecycle: the render and commit phase. The render phase modifies the DOM, so that it corresponds to the React component. In the commit phase, components can perform changes, e.g. setting a new **state** and therefore activating a popup, starting a network request, etc. Once the commit phase has finished, React will switch into the render phase. In order to not re-render every component in each render phase, React only triggers the rendering of components, whose **props** or **state** have changed. React does not deep-compare **props** or **state**, but rather shallow-compares them. Complex JavaScript types like functions and objects (including arrays) are only compared by their reference. To optimize performance, new objects should therefore only be generated if their content actually changed. A bad practice is the use of lambda functions within JSX. Since a new function will be generated every time the JSX is rendered, the reference is never stable. This will trigger a re-rendering of the component receiving the callback each time the parents re-renders. To solve this, the lambda function has to be defined outside the render function.

Sweble uses the architectural concept of container and presentational components. Presentational components are usually reusable, stateless components, while container components are mostly stateful and not reusable. Containers are used to contain individual pages, as e.g. the document view page. Since the React tree consists of only one root component, the different containers are rendered conditionally depending on the URL (also known as routing). The concept of container and presentational components is not treated as fixed rule but rather as guideline or best practice.

Furthermore, Sweble uses the static type checker Flow. Therefore, the frontend and BFF source code is written with Flow typings. The code excerpts shown in the frontend section are only partially typed to improve readability. Moreover, Sweble uses the UI library Semantic UI. It provides many pre-built components (e.g. buttons, popups and icons), which were partly used to implement the commenting UI.

5.2.2 Structural Overview

The main part of the commenting UI has to be integrated into the document view, as presented in section 4.3. The document view is implemented in the container **RenderResourcePage**. Currently, the **RenderResourcePage** includes the article renderer directly, which renders a WOM document as HTML. Since the commenting functionality has to adjust the article renderer e.g. to underline text references, the main commenting container is placed between the **RenderResourcePage** and the article renderer as shown in Figure 5.3. The commenting functionality includes two sub-containers. The **CreateComment** component, responsible to render the form in order to submit a new comment, as well as the **ViewComments** component, which implements the UI to display a comment. Both components are inserted into the WOM-tree so that they are rendered at the right position. An decorator approach is used to insert the

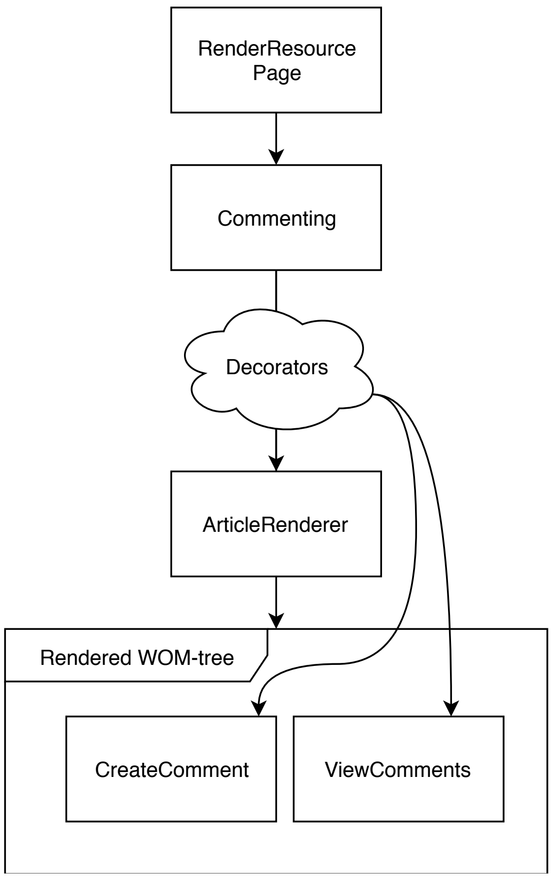


Figure 5.3: Overview of the frontend commenting implementation. Each box represents a React component and the arrows visualize the parent-child connections. The decorators are part of the propagated **props** and render the CreateComment and ViewComments components into the rendered WOM-Tree.

components into the actual document tree (presented in subsection 5.2.7).

5.2.3 Commenting Container

The root level of the frontend implementation is implemented as the **Commenting** container. It embeds and manages all commenting sub-components as well as the article renderer. The commenting container holds the **state**, whether the comment create form or the comment view should be rendered. If neither is active the container will enable the selection popup, which is rendered when the user selects a text range.

As the container is mounted, it loads the comments for the current document. This logic is implemented in the *componentDidMount()* method. The container uses Redux to store the state of the network request. Since the **store** is immutable, a specific mechanism of **actions** and **reducers** has to be used to modify it. Actions are plain JavaScript objects, which contain a type (in form of a string) and a payload. Actions can be dispatched by any React component. In case of the comments component, the corresponding action is dispatched as the container is mounted. The action's payload contains the current project, commit and resource name. These data are needed in order to generate the API endpoint URL and are provided to the container via its props from the **RenderResourcePage**. The expected payload is defined as part of the action and reducer. As the commenting container dispatches the *load-comments* action with its type and payload, a corresponding reducer mutates the store. The reducer does not generate a deep copy of the store but rather a shallow copy.

After the reducer mutated the store, it contains the information that the request is loading. However, the actual network request to fetch the comments is started by another library called Saga. With the help of the Saga library it is possible to listen for specific actions and execute code as the action is detected. For the "load-comments" action, the corresponding Saga contains the network request logic. The Saga code will use the action's payload to build the corresponding endpoint URL. As the network request finishes, the saga dispatches a corresponding action by itself. The reducer corresponding to the action mutates the store, which will then contain the result of the network request.

To store the state of a network request, Sweble defines four states to categorize a request (compare with Figure 5.4). If a request isn't send yet, the entry will be in the **initial** state and as the network request is loading, it is in the **loading** state. Since a network request can either succeed or fail, a **loaded** or **error** state represent a finished network request. Both contain additional information. In case of an **error**, the state contains information to the error and in case of **loaded**, the state includes the network request's results. The result of the "load-comments" request is an array of comments.

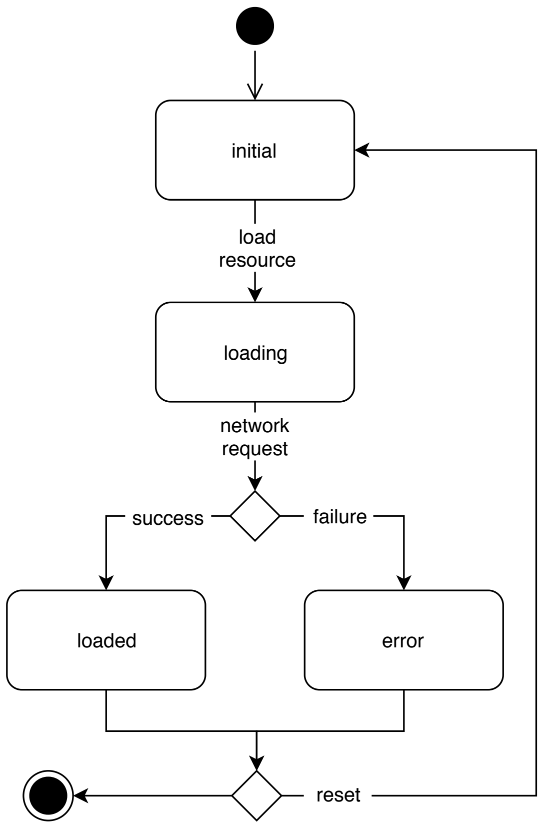


Figure 5.4: UML state machine showing the four states to categorize a network request.

The commenting container has to access the state within the **store**. Instead of accessing the **store** directly (via a reference), parts of the store are mapped into the commenting container's **props**. The container includes a *mapStateToProps* function, which gets the entire store and extracts the relevant parts. Passing parts of the store via the **props** has the advantage that a change in the corresponding sub-store will trigger the lifecycle of the component. As the component is re-rendered and the comments are in the **loaded** state, the container can insert the text underlining into the article renderer.

In total, there are four network requests, which are mapped into the **store**. Two requests to get and two to create a comment or comment entry. Since a network request can be categorized in one of four states, each request implementation contains four actions. Each action represents a different network state. Corresponding to each of the four actions, a reducer has to be defined. Furthermore, the saga code has to be defined for each request. Therefore, to perform one network request, which is mapped into the store, a lot of boilerplate-code has to be written. Hence, Sweble contains helper functions to reduce some of the needed code automatically. The *redux.js* source code file mainly contains the type definitions and the code to perform the network request. The actual actions and reducers are generated with the help of the helper functions.

5.2.4 Selection Popup

The first step in creating a comment is the selection of a text range. Thus, the frontend implementation has to detect a selection within the document. I implemented the **SelectionHandler** component that is able to detect text selections within its children. It expects the article renderer as its child. The **SelectionHandler** relies on the Selection Web API[2], which is supported by all modern and commonly used browsers[3]. However, the underlying specification by the World Wide Web Consortium (W3C) is still marked as working daft[4]. Therefore, certain aspects of the API could change in the future.
While the term selection implies that a text range is selected (and highlighted on the screen), a selection can also be collapsed. A collapsed selection has a width of zero characters and is not highlighted on the screen. In a text processor, a collapsed selection represent the cursor position.

Selection Detection

The Selection Web API offers two methods to get the current selection. There is a *selectionchange* event as well as a *window.getSelection* method. While the event is faster to implement, it has the disadvantage that it emits multiple times and not only once if the selection is final. If the user selects a text range, each newly selected character will change the selection and trigger a *selectionChange* event. Each time a selection is detected it has to be checked, whether the selection is within the **SelectionHandler's** children. In order to save some CPU cycles I implemented the detection using *window.getSelection* as well as mouse-up events. In general, a text selection is completed as the user releases the mouse key. Every-time the mouse-up event fires, the **SelectionHandler** will call the *window.getSelection* method.

The mouse-up event listener can be attached to arbitrary DOM nodes. If an event occurs within a child, the event will be propagated upwards - also known as event bubbling. Since the **SelectionHandler** captures events within the wrapper, one solution would be to attach the listener to the wrapper. However, this approach would not capture all events, which correspond to a valid selection. The user can

[2]https://developer.mozilla.org/en-US/docs/Web/API/Selection
[3]https://caniuse.com/#feat=selection-api
[4]https://www.w3.org/TR/selection-api/

start the selection within the wrapper and end the selection outside. This may still produce a valid selection but the mouse event would be triggered outside the wrapper and not be detected. Therefore, to detect every selection, I attached the listener to the document node.

A problem that arises from using mouse events was, that mouse events in quick succession would produce an invalid state, e.g. showing a popup where no selection is present. If mouse events are produced rapidly, there can be a delay between the mouse event firing and the *window.getSelection* providing the right data. To overcome this issue I used the *setTimeout* function, which will invoke a callback after a specified number of milliseconds. In the **SelectionHandler** the timeout is set to zero milliseconds. Even with a timeout of zero milliseconds, the callback will not be invoked right away but rather be delayed. JavaScript is executed synchronously as part of the browser event loop. The event loop "orchestrates the main thread of the browser, which includes JavaScript, events, and rendering"[5]. As the *setTimeout* function is executed, the callback is queued as a task and will be invoked by the event loop once the JavaScript call stack is empty (Archibald, 2015). At the time the call stack is empty the click event has been processed completely. The *window.getSelection* method then returns the selection, which is actually visible in the browser.

Selection Filtering

As a selection is detected, the selection object from the Selection Web API is filtered, transformed and normalized. First of all, the selection has to be filtered whether it is within the **SelectionHandler**. The SelectionHandler's children are wrapped in a div node with an unique ID (compare with Listing 5.2). The unique ID is a statically defined number and will be increased each time the SelectionHandler is mounted into the React-tree. Since the unique ID is a counter, multiple SelectionHandler can be used on one page. The selection includes a reference to two nodes (*anchor* and *focus*) within the DOM, which define the range of the selection. In order to verify if a node is within the SelectionHandler, the DOM is traversed upwards until either the body or wrapper node is reached. If one of the nodes reached the body, null will be emitted as selection, since the selection is (partly) not within the wrapper.

```
class SelectionHandler extends React.Component<Props> {
  static wrapperCounter = 0;
  wrapperId = SelectionHandler.wrapperCounter++;

  componentDidMount() {
    document.addEventListener('mouseup',
      () => window.setTimeout(this.checkForSelection, 0),
```

[5] https://2018.jsconf.asia/

```
    );
  }

  checkForSelection = () => {
    const rawSelection = window.getSelection();
    let selection = null;
    if (this.isValidSelection(rawSelection)) {
      selection = normalizeSelection(rawSelection);
    }
    // Only calls the callback if the selection has changed
    this.emitSelectionChanged(selection);
  }

  render() {
    return (
      <div id={this.wrapperId}>
        {this.props.children} // Contains the article renderer
      </div>
    );
  }
}
```

Listing 5.2: Excerpt of the SelectionHandler's source code showing the main algorithm to detect selection within its children.

Computing Node ID and Offset

In case both nodes are within the wrapper, the node IDs and offsets have to be computed. The node IDs are computed once and then attached to the nodes as data attributes, as will be described in subsection 5.2.7. The node IDs can be accessed directly through the DOM. However, it is not sufficient to get the node IDs from the nodes which are referenced in the selection. The article renderer's decorator system has the ability to add additional nodes into the DOM. Thus, a referenced node may not be a representation of a WOM node and would not contain a node ID attribute. To overcome this issue, the DOM-tree is traversed upwards until a node with a node ID attribute is found. Computing the offsets is similar to the node IDs. The selection interface contains the offset to the referenced node. However, a single WOM text node could have been split into multiple parts by the decorators. To get the offset, which corresponds to the WOM text, the algorithm has to find the first node representing a WOM text. Each DOM node representing a WOM node has the WOM-type attribute, which is used to detect text nodes. The offset can then be computed recursively by adding the text lengths of previous siblings and ancestors to the selection's offset.

Before emitting the selection, it has to be normalized. Instead of including anchor and focus entries, the **NormalizedSelection** interface contains a start and end entry (compare Listing 5.3). While the anchor in the selection object corresponds to the point where the selection started, the start in the **NormalizedSelection** references the point of the selection, which comes first in the document. In order for the **SelectionHandler** to pass the **NormalizedSelection** to the component's parent, the **SelectionHandler** accepts a *onSelectionChange* callback as prop. The **SelectionHandler** will call the callback each time the selection changes. If no (valid) selection is present, null is passed as argument to the callback and otherwise the **NormalizedSelection** is passed. Subsequently, identical values are not emitted.

```
export type SelectionNode = {|
  +offset: number,
  +womId: number,
|};

export type NormalizedSelection = {|
  +start: SelectionNode,
  +end: SelectionNode,
  +type: 'Caret' | 'Range' | 'None';
  +ref: ?Range,
|};
```

Listing 5.3: The NormalizedSelection JavaScript interface written in Flow syntax. The type and ref property in the NormlaizedSelection are taken from the Selection Web API.

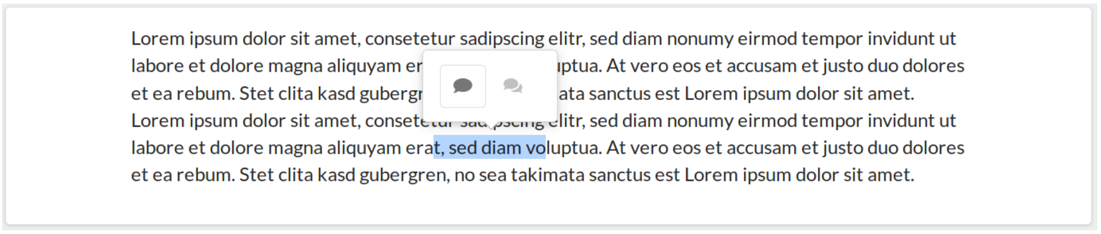
Popup

As a selection is detected, the Popup has to be displayed. The Semantic UI library includes a popup component, which does not have to be inserted at the exact position within the DOM. The popup is rather positioned absolutely with the help of Cascading Style Sheets (CSS). Therefore, the decorator system of the article renderer is not needed. The popup component expects a reference to an element as a prop, which will be used to position the popup above it. The **SelectionPopup** component combines the **SelectionHandler** with the Semantic UI popup as shown in Listing 5.4. It uses the React state to store whether a selection is active or not. Since the popup contains two buttons to create or view comments, the component's props include the *handleCreateComment* and *handleViewComments* callbacks to lift the click event up to the Commenting component. The view comments button is only active if the selection overlaps with a text reference of a comment. To determine if the button is active, the *selectionHasComment* callback props is needed. It returns a *boolean* value, which describes whether a given selection overlaps with any comment text range. Another way to check the comments and selection for overlaps, would be to pass the comments array into the **SelectionPopup**. Since the **SelectionPopup** does not require knowledge of the acutal comments, the callback approach was used.

```
class SelectionPopup extends React.Component<Props, State> {
  render() {
    return (
      <SelectionHandler onSelectionChange={...}>
        {this.props.children} // Contains the article renderer
      </SelectionHandler>
      <Popup
        // Selection reference to position the popup:
        context={this.state.selection.ref}
        content={...} // The popup's contents
      />
    );
  }
}
```

Listing 5.4: Code excerpt of the SelectionPopup showing the usage of the SelectionHandler in combination with the popup

One problem with the Semantic UI's popup is that a change of the provided node reference will not trigger a re-render. So if the user clicks on one referenced text and then on another, the popup will stay in the old position. This problem can be solved by disabling the popup first and then activating it again with the newly referenced selection. However, setting both states in the same lifecycle will not disable the popup, since the second state will overwrite the first state. Thus the states have to be set in two different cycles. To do so, the components state includes the property *nextSelection*. Since disabling the popup triggers a new cycle, the *nextSelection* property is checked in the *componentDidUpdate* lifecycle method. If the *nextSelection* is set, it will be used as new reference for the popup. This approach forces the popup to re-render, since it will be hidden shortly for one lifecycle.



Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut
labore et dolore magna aliquyam er uptua. At vero eos et accusam et justo duo dolores
et ea rebum. Stet clita kasd gubergr ata sanctus est Lorem ipsum dolor sit amet.
Lorem ipsum dolor sit amet, consete elitr, sed diam nonumy eirmod tempor invidunt ut
labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores
et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

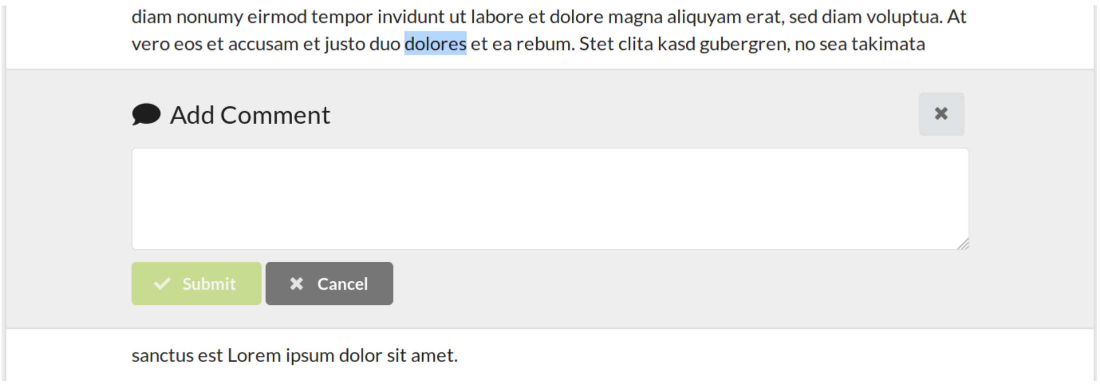
Figure 5.5: Screenshot of the final selection popup.

Another initial challenge was that the activation of the popup would trigger a re-rendering of the article renderer, which will discard the selection. Browsers discard a selection if a re-rendering of the corresponding nodes occur. This effectively lead to an unusable UI, since the selection would be discarded as soon as the Popup is shown, thus removing the popup again. This issue occured as the popup state was part of the commenting container. The commenting container regenerated the decorators everytime it was updated. Since the popup triggered an update, new decorators were passed into the article-renderer, which re-rendered the WOM document. To solve this issue, the state of the popup was integrated into the **SelectionPopup** component, which will not trigger an update of the commenting container.

5.2.5 Create & View Comment Containers

The **CreateComment** and **ViewComments** containers implement the majority of the UI. Their source code contains mostly markup in form of HTML and Semantic UI elements. Both containers use Redux to create or reply to comments. Furthermore, the **ViewComments** container uses Redux to load the comment entries. The container gets the current comment id via its props and validates whether the entries are loaded and correspond to the given comment id. If the comment entries are not loaded or reference a different comment id, the container will dispatch an action to load the comment entries. As the state of the comment entries may be loading, the container renders a loader.

As shown in Figure 5.6 the UI is positioned below the text reference. To achieve this behaviour in React, the container has to be positioned after the last word in the text line. However, the last word in the text line depends on the screen size. It will differ between a smartphone and a desktop. Another approach is to use CSS as shown in Listing 5.5. The CSS specification includes a *float* property, which allows elements to be surrounded by text. It is generally used to position an image on one side of a text, while the text wraps around the image. The container is inserted directly behind the text reference. By combining the float property with a *width* of 100% the text will wrap around the container at the top and bottom.


diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata

sanctus est Lorem ipsum dolor sit amet.

Figure 5.6: Screenshot of the final create comment UI showing the interface placed beneath the corresponding text range.

Furthermore, the commenting UI should interrupt the white background of the document, so that there is a clear separation between the document and the commenting. However, the commenting UI's width is restricted by the width of the text, which is less then the width of the white background. To overlay the whole background, the container has to be as large as the article renderer's parent. The commenting UI width can be enlarged by using a negative *margin* on the container and hiding overflows (in CSS: *overflow: hidden*) on the article renderer's parent. Without the *overflow* property, the container would extend beyond the width of the parent. In order for the container's children to have the same width as the document, the negative *margin* has to be balanced in an inner element with a positive *margin*.

```
<span class="commenting__inline-box__outer">
  <span class="commenting__inline-box">
    <span class="commenting__inline-box__inner">
      <!-- The content -->
    </span>
  </span>
</span>

.commenting__inline-box__outer {
  float: left;
  width: 100%;
}

.commenting__inline-box {
   margin: 10px -200px 10px;
   display: block;
}

.commenting__inline-box__inner {
  margin: 0 180px;
  padding: 20px;
  box-sizing: border-box;
  display: block;
}
```

Listing 5.5: The combination of HTML and CSS used to position the commenting UI below the referenced text line.

One problem with this approach that I could not solve completely, is that React will throw warnings regarding the DOM nesting:

```
validateDOMNesting(...):
<form> cannot appear as a descendant of <p>
```

Since the containers are embedded within the article renderer, its parent element is usually a paragraph. Rendering a block element as e.g. a *form* or *div* within a paragraph makes no sense regarding the semantics of the elements. To limit this warning all my components use a *span* instead of a *div* element. However, the components of Semantic UI are using *forms* and *divs*. Therefore, some warnings are still produced. Since all commonly-used browsers will render it correctly and since the warnings are only visible in the developer console, there is no real-world benefit in replacing all Semantic UI elements.

5.2.6 Article Renderer

To render documents, Sweble includes the article renderer. The article renderer is a React component, which is able to transform a WOM document into a HTML representation. For each WOM element it contains a corresponding React component able to render the specific element. To map a WOM element to the corresponding React component, the article renderer uses the factory pattern. The factory takes a WOM element and returns a component instance.

For the article renderer to generate a tree, the WOM nodes are mapped recursively. Each React component representing a WOM element will use the factory to render its children. The factory is passed into each component via the props. Starting at the top of the document, the article element will be mapped to its React component. This component will invoke the factory to map each of its direct WOM children. In case of the article, the only child is a body element which will be rendered as the article's child. As the body component is generated, it will also map its children which can contain multiple elements and set them as its children. As all elements are rendered, the structure of the WOM and DOM tree are identical.

Depending on the use-case of the article renderer, certain contents have to be modified, wrapped or added. As the commenting UI has to modify text (e.g. underline text references) or insert the commenting UI directly into the document, this implementation needs to be able to manipulate the article renderer. For such use-cases the article renderer includes a decorator system. Multiple decorators can be passed into the article renderer via the props. Each decorator is able to change the rendering, modify attributes and overwrite the components children. Since multiple decorators need to be able to perform changes, they have to be chain-able. The actual generation of the component cannot be performed until all decorators were invoked. Otherwise, the component would not get the changed props from some of the decorators. Therefore, the return value of the decorators is a function, which accepts the props and generates the actual element (compare with Listing 5.6). Each decorators implements the *applyTo* method, which will be called by the factory for each element. The method gets the components **props**, the *createElement* function as well as the component class, which corresponds to the React component.

```
type DecoratorReturnType =
  (props: ReactArticleElementProps) => React.Element<any>;

class Decorator {
    applyTo(
        props: ReactArticleElementProps,
        createElement: DecoratorReturnType,
        componentClass: ArticleReactElement,
    ): ?DecoratorReturnType {
        return createElement;
    }
}
```

Listing 5.6: The decorator JavaScript interface.

5.2.7 Decorators

The commenting functionality relies on some newly implemented decorators, which are presented in this subsection. The decorators may accept parameters in their constructor, as used by most of the commenting decorators. The decorator array which is passed into the article render, is generated by the commenting container.

Attaching Node IDs

As mentioned in the subsection 5.2.4, the node IDs are computed once and then attached to the corresponding DOM nodes. The JavaScript WOM implementation already handles the computation of the node IDs. Therefore, each WOM element already contains its ID. A decorator is used to attach the node IDs from the WOM element to the React element. The **AttatchNodeIdDecroator** manipulates the props which will be passed into the React component. The decorator adds a data attribute containing the node ID to the element's props (compare Listing 5.7).

```
applyTo(props, createElement) {
  const nodeId = props.womElement.nodeId;
  if (nodeId != null) {
    overrideReactProps(props, { 'data-node-id': nodeId });
  }
  return createElement;
}
```

Listing 5.7: Code excerpt of the AttatchNodeIdDecroator showing the code to add an attribute to the props.

Appending Elements & CSS Class Wrapper

Both the **AppendDecorator** as well as **ClassNameDecorator** are simple structural decorators. The ClassNameDecorator wraps the WOM element in a span element containing one or multiple CSS classes. The AppendDecorator appends a React element to a WOM node. It is used to append the CreateComment or ViewComments containers into the WOM tree at a specific position. As shown in Listing 5.8, a React fragment is used here since React disallows to return an array of elements. The fragment is a special element as it will not be rendered to the DOM. It's sole purpose is to wrap both elements into a single node, which can then be returned.

Furthermore, a key prop is attached to the fragment. React expects that each element in an array contains an unique key property. Since the decorator may apply to a node which is rendered in an array, a structural decorator should always use a key. The key is provided by the decorator base class, which generates a unique ID for each decorator.

```
applyTo(props, createElement) {
  return (props_) => (
    <React.Fragment key={this.getDecoratorKey(props_)}>
      {createElement(props_)}
      {this.elementToAppend}
    </React.Fragment>
  );
}
```

Listing 5.8: Code excerpt of the AppendDecorator showing the usage of React Fragments to append a React element in the WOM tree.

Decorating Text Ranges

A general limitation of the decorator system is, that only one decorator is able to change an element's children. The children of a React element are propagated via the props. The children are therefore a property within the props, which can be modified by the decorators. However, multiple decorators setting the children property would overwrite the changes of the previous decorator. Resolving this conflict within the decorators is complex, since the decorators do not have knowledge of the previous decorators.

The commenting functionality has to split text nodes in order to apply an effect only to a specific range. Furthermore, multiple decorators can affect the same text ranges within a single text node, e.g underlining the text references and highlighting an active selection within a single node. A central mechanism is needed to perform the text node splitting.

To solve this limitation, I implemented the **TextRangeDecorator**. It is a higher order decorator which accepts other decorators as input and applies them to a certain text range. It's constructor accepts an array of **TextDecorators** (compare with Listing 5.9). The TextDecorator interface contains a decorator, the text range and a definition how to apply it. The text range is from the type **SimpleSelection**, a simplified version of the NormalizedSelection, which only contains the necessary node IDs and offsets. A SimpleSelection can be generated from NormalizedSelection as well as the text range definition in comment models. The SimpleSelection can therefore be used for the initial create comment view, underlining text references and highlighting the text reference in the comment view. The apply definition (*applyTo* property) mainly depends on the kind of the decorator. For example, the CSS class decorator has to be applied to all text splits, while the append decorator should only be applied once to the last node or text split.

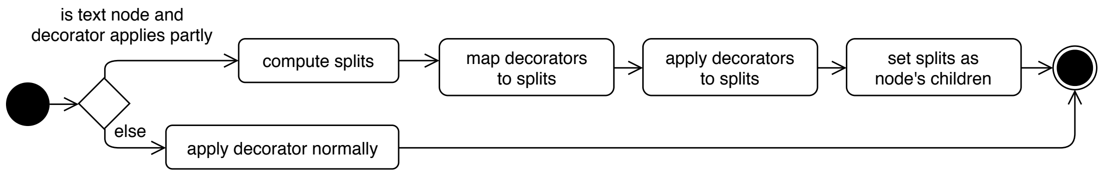
```
type TextDecorator = {|
  +textRange: SimpleSelection,
  +applyTo: 'start' | 'end' | 'all',
  +decorator: Decorator,
|};

class TextRangeDecorator extends Decorator {
  constructor(textRangeDecorators: TextDecorator[]) {
    super();
    this.textRangeDecorators = textRangeDecorators;
  }
  // ...
}
```

Listing 5.9: The interface of the TextRangeDecorator.

As every Decorator, the TextRangeDecorator is invoked for each WOM node. The first step for each node is to check whether any of the given decorators apply to the specific node. Whether a decorator applies to a node depends on the text range and the *applyTo* property. If *applyTo* is set to either 'start' or 'end', only the start or end node is affected. If *applyTo* is set to 'all', all nodes which are part of the text range are affected. Since this check will be performed for each node, it important to mind the performance. To improve the speed of the check, whether a decorator applies to a node, the TextRangeDecorator computes all affected node IDs once in the constructor. Those node IDs are stored in a JavaScript set which is able to quickly check if a given node ID is present in the set.

The following algorithm steps can be categorized into different segments as visualized in Figure 5.7 and are executed for each decorator. It has to be checked whether the decorator applies to the whole node or if the node is a text node and the decorator applies only partly. In the first case, the algorithm can be applied normally by invoking the *applyTo* method of the decorator directly. A decorator applies only partly to a text node if the current node is either the start or end node of the decorator's text range and its offset does not cover the whole node. In this case the text node has to be split.



is text node and decorator applies partly
compute splits
map decorators to splits
apply decorators to splits
set splits as node's children
else
apply decorator normally

Figure 5.7: UML Activity Diagram showing the performed steps for each decorator within the TextRangeDecorator.

The first step in splitting a text node is to compute all split positions. The algorithm goes through the partly applying decorators and collects their offsets. The offset depends on whether the current node is a start or end node. As all offsets are collected duplicates are eliminated and the offsets are sorted. In a second step the decorators are mapped to the offsets. Whether a decorator applies depends again on the current node as well as on the current offset generated in the first step. The map of offsets and decorators is then used to apply the decorators. The algorithm goes through each map entry, gets the corresponding text split, converts it into a React element and applies the corresponding decorators. As a last step the resulting array of React elements is set as the original node's children.

An example for the TextRangeDecorator on the basis of a single text node with the contents "Implement Commenting" is presented in the following. As example two decorators are applied to this single text node. The corresponding input to the TextRangeDecorator is outlined in Listing 5.10. A visualization of both text ranges is shown in Figure 5.8.

```
[
  {
    textRange: {
        startNodeId: 4,
        startOffset: 0,
        endNodeId: 4,
        endOffset: 9,
    },
    applyTo: 'all',
    decorator: ..., // A decorator instance
  }, {
    textRange: {
        startNodeId: 4,
        startOffset: 5,
        endNodeId: 4,
        endOffset: 13,
    },
    applyTo: 'all',
    decorator: ..., // A decorator instance
  }
]
```

Listing 5.10: Exemplary input for the TextRangeDecorator.

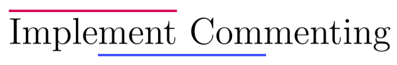


Figure 5.8: Visualization of the TextRangeDecorator input shown in Listing 5.10 using the text "Implement Commenting".

The text positions at which the text has to be split are extracted from both *textRange* properties. In this example both text ranges affect a single node and both the start and end offsets have to be used. The split positions are therefore 0, 5, 9 and 13. Since 0 is at the beginning of the text, the position is omitted. The resulting splits are "Imple" (0-**5**), "ment" (6-**9**) and " Com" (10-**13**) as well as the rest "menting" (14-20). The decorators are then mapped to the splits. The first decorator is applied to the first and second split, while the second decorator is applied to the second and third split. When all decorators were applied to the splits, the TextRangeDecorator replaces the text node's child (the simple text) with all four splits.

5.3 Backend

Three parts of the backend were extended in this thesis: the backend-for-frontend (**BFF**), **Project Storage Service** as well as the **PostgreSQL**'s database scheme.

5.3.1 Backend For Frontend

The BFF is built on top of NodeJs and written in JavaScript and thus able to share type definitions with the frontend. In terms of the commenting functionality, the BFF only functions as a bridge between the frontend and the microservices. Since all commenting APIs are located in the project-storage-service, each request regarding comments is forwarded to this microservice. Before forwarding the request, the BFF will validate whether the request is valid. For example, in case certain properties are missing, the BFF will reject the request directly.

In terms of the commenting functionality, the BFF is responsible to transform the request and response payloads. Within the payload entities like users, projects or comments are referenced by a URL. This URL contains the internal IP of the microservice, which is not accessible by the client. The URLs have to be mapped to the external endpoints as shown in Listing 5.11. To map the response payloads, the JSON from the microservice is parsed into an JavaScript class and mapped to a response class, which converts the URLs. The resulting object is then sent to the frontend as JSON.

```
// Internal URL
http://172.42.42.14:8080/v2/projects/1
// External URL
http://localhost:6001/api/v2/projects/1
```

Listing 5.11: The internal and external URLs, which are converted by the BFF.

5.3.2 Project Storage Service

The Project Storage Service is one of two main backends and written in Java. Its main purpose is to store and manage projects including documents. Figure 5.9 gives an overview of the project storage service's structure. The three layers are implemented as Java packages. Each layer contains their own models which are used to store the comment or comment entry data.

project-storage-service

The project-storage-service package includes the REST[6] endpoints. All commenting endpoints are defined in the **CommentResource** class. The class validates and extracts the parameters from a request and invokes the corresponding method within the **ProjectStorageManager**. The ProjectStorageManager will fetch additional data which are required to process the request. The frontend does not distinguish between commits and branches. The ProjectStorageManager validates which type was provided and fetches the commit or branch. As all required data are available, the ProjectStorageManager will invoke the **CommitStorageAPI**.

[6]Representational State Transfer

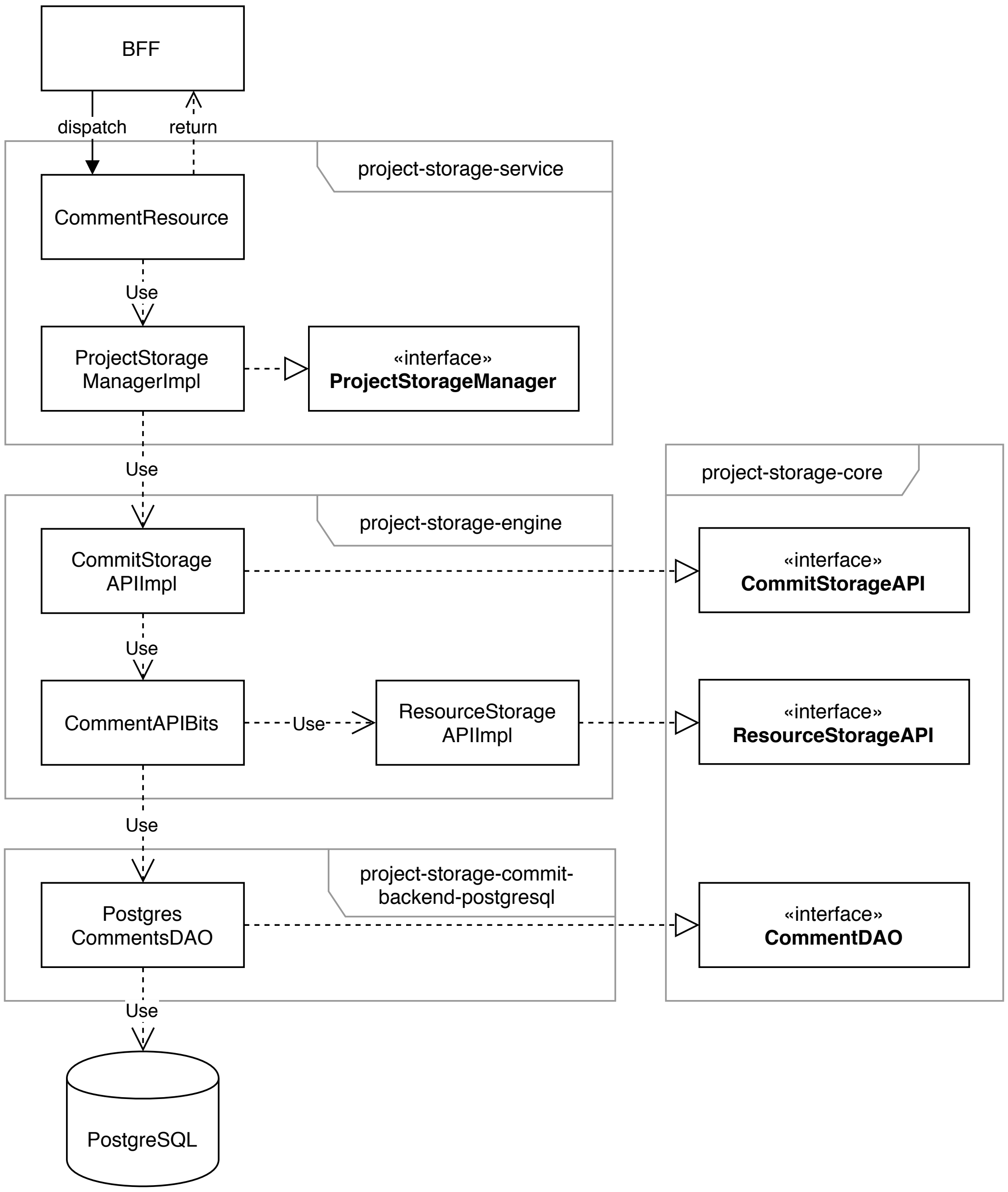


Figure 5.9: Excerpt of the project storage service implementation

project-storage-engine

Most of the business logic is implemented in the project-storage-engine. The CommitStorageAPI includes low-level functions to create and manipulate the comment models. The offered functions are not a direct representation of the REST endpoints. The ProjectStorageManager invokes multiple methods in order to fullfill a single request. The CommentAPIBits, which is invoked by the CommitStorageAPI, consist mainly of the algorithm to compute the text reference updates (presented in subsection 5.3.4). To compute reference updates, the current and the changed WOM are needed. The CommentAPIBits uses the **ResourceStorageAPI**, which is also part of the project-storage-engine, to fetch both WOMs. The ResourceStorageAPI uses the database Cassandra to store documents.

project-storage-commit-backend-postgresql

The database connection is handled in the project-storage-commit-backend-postgresql package. It contains the logic to store, fetch and modify data in the PostgreSQL database. Each database table has a Java class representation. Those models are generated from the database scheme by the jOOQ[7] library.

project-storage-core

The project-storage-core package includes interfaces for the engine and postgresql package as well as the models for the business logic.

5.3.3 Identifying Comments

In the previous chapters, the comments of a document were identified by the triple of project, commit and document. However, internally text selections are not identified by a commit but rather by a change. A change represents an update of a specific document, while a commit comprises a set of changes. The text references have to be updated only as the document changes. As a commit is created, text references of an unchanged document would still reference the previous commit. Therefore, the direct connection between the text references and commit would be lost. Text selections could only be retrieved by iterating through prior commits.

To solve this limitation, the selection references the change. As the document is modified, a change is created and the selections are updated. The new selections will reference the new change. Therefore, all selections of a document will

[7] https://www.jooq.org

reference the last change. To retrieve the selection for a given document, the algorithm has to determine the last change of a document which can be retrieved from any given commit.

A commit includes the last change for each document. Each commit references a B-Tree, which consist of all latest changes. In the B-Tree the changes are identified by the corresponding document's name. Therefore, to get the latest change for a given document, the algorithm has to traverse only the B-Tree until the change is found. This change can then be used to identify all text selections which were present at the time of the change.

5.3.4 Updating the Node References

One main part of the backend implementation is the algorithm to update text references as a document is modified. The project storage service includes the endpoint to store a new revision of a document, which is handled within the ProjectStorageManager. The algorithm to update the references is located in the CommentAPIBits and invoked in the ProjectStorageManager after the commit has been created.

As outlined in the previous subsection, the created commit includes a set of changes. Each change represents the modification of a document. To update the references both the old and new WOM documents are needed. Therefore, for each change contained in the commit, the previous change is retrieved. A map containing the old and new changes is then passed into the CommentAPIBits class.

For each map entry, the algorithm will retrieve the selections which reference the old change. In case that no selections are present, the document includes no comments and the algorithm will proceed to the next change. As changes are present, both versions of the document have to be analyzed. The first step is to retrieve both WOM documents from the ResourceStorageAPI and to apply the HDDiff which returns a list of EditOps.

The general concept to apply the EditOps to text references is to go through each EditOp individually. The text reference which consists of two node IDs is converted into a list of nodes. The list includes all nodes which are part of the text reference. The operations can then be applied to this list. For example, a delete operation may remove a node from the list. As a document can contain multiple selections, each operations is applied to each selection as shown in Listing 5.12.

```
private void applyEditOps(
    List<CommentSelections> selectionList, List<EditOp> editOps
) {
  for (EditOp editOp : editOps) {
    switch(editOp.getType()) {
      case DELETE:
        EditOpDelete delOp = (EditOpDelete) editOp;
        for (CommentSelections selection : selectionList) {
          selection.remove(delOp.getDeletedNode());
        }
        break;
      // Further cases for MOVE, SPLIT and UPDATE
    }
  }
}
```

Listing 5.12: Excerpt of the CommentAPIBits showing the main loop to apply the editOps to the text selections.

The list of nodes is not stored as variable, but rather contained in a specialized class called **DiffNodeRange**. It is the representation of a single text range containing the list as well as the offsets. It includes methods to apply certain operations onto itself (compare with Listing 5.13). The *remove* method will delete a node from the list and the *split* method will insert the second half of the split into the list. Both methods will validate the start or end offset in case one of the edge nodes is affected. The update operation is handled by the *checkOffsets* method. It needs the diff between both texts (represented as a list of diffs) to calculate the new offsets. The diff of an update operation is computed in the CommentAPIBits and passed into each DiffNodeRange.

```
public class DiffNodeRange {
    private List<DiffNode> nodes;
    private int startOffset;
    private int endOffset;

    public DiffNodeRange(
      List<DiffNode> nodes, int startOffset, int endOffset
    );

    public boolean hasCollapsed();
    public boolean remove(DiffNode node);
    public void split(
      DiffNode splittedNode, DiffNode otherHalf, int splitPos
    );
    public void checkOffsets(
      DiffNode node, LinkedList<DiffMatchPatch.Diff> diffs
    );
}
```

Listing 5.13: Interface of the DiffNodeRange class which includes the methods to apply some edit operations.

The move operation is not handled in the DiffNodeRange but rather in the CommentSelections which is able to represent fragmented references (compare with Listing 5.14). Since I wanted to offer a proper handling of the move operation, the internal representation has to be able to handle not only one but multiple text ranges. Each text reference will be mapped to one **CommentSelection** which contains one DiffNodeRange initially. The delete, split and update operation are forwarded into each DiffNodeRange. The move operation is handled within the CommentSelections class. It invokes the *remove* method of each DiffNodeRange to remove the nodes and creates a new DiffNodeRange which consists of the moved nodes. As all edit operations are applied, the CommentAPIBits will use the largest DiffNodeRange as new text reference.

Once all operation were applied, each CommentSelection and the list of nodes has to be converted back into a representation of node IDs. This transformation as well as the initial transformation from the node IDs to the list is backed by a map containing each node and its ID. This map is generated for both document version. The map based on the old document is used for the first transformation (node IDs to a list) and the other map is used for the second transformation.

```java
public class CommentSelections {
    private List<DiffNodeRange> ranges;

    public CommentSelections(List<DiffNode> initalNodes);

    public boolean hasVanished();

    public void remove(DiffNode node);
    public void move(DiffNode movedNode);
    public void split(DiffNode splitedNode, DiffNode otherHalf,
        int splitPos);
    public void checkOffsets(DiffNode node, LinkedList<
        DiffMatchPatch.Diff> diffs);
    public DiffNodeRange getFirstRange();
}
```

Listing 5.14: Interface of the CommentSelections which is an internal representation of fragmented references.

5.3.5 Database Layer

Scheme

Sweble uses the database migration tool Liquibase[8] to apply changes to the

[8]https://www.liquibase.org/

database. The SQL statements required to extend the database schema are defined in their own migration file in the project-storage-commit-backend-postgresql package. The new database tables are shown in Figure 5.10. The main commenting table contains the owner, project as well as the time of creation. All other information are stored in additional tables which reference the main table. The comment entries are stored in the Comment_Entry table and the text references in the Comment_Selection table. Since text ranges will change as the document is adjusted, multiple selections refer to one comment. Depending on the commit and change the document is viewed in, the equivalent text reference is fetched from the database.

Another newly added table is the Comment_State table which is an improvement for the history view of comments. The state of a comment could also be stored within the selection. However, a new selection is only created if the document was actually changed. Viewing a document at a commit which did not affect the document, could show a different comment state as to what the actual state was. Therefore, the state is stored within its own table. A new entry is created everytime the state is changed. The Comment_State table is joined as the comment is selected.

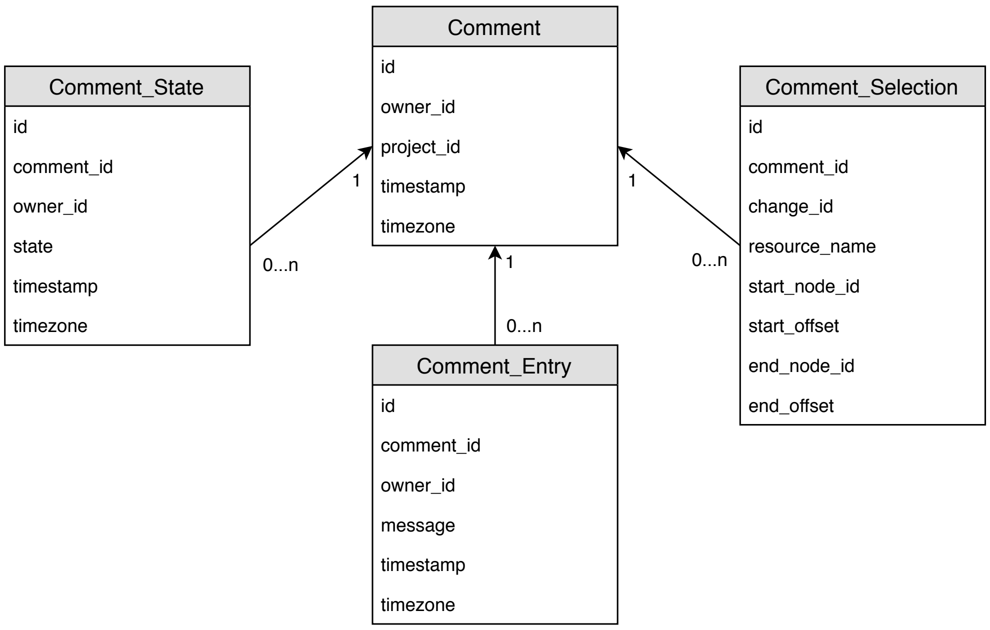


Figure 5.10: Database scheme of the commenting functionality

Freeze Comments within Commits

A requirement is to be able to view the state of a comment at an older commit. Whether a comment is present at a given commit, is determined by the presence or absence of a selection entry. Each commit contains the last change of a specific resource. A text selection can therefore be retrieved based on the change id. Comments created after the time of a given change will not be fetched, since the comment has no selection which references the older change id.

To retrieve all comment entries at the time of a given commit, the timestamps of the commit and comment entries are compared. All entries which timestamp is less or equal to the timestamp of the commit will be fetched from the database. The comment state is treated similar, since a timestamp is used as well. In case of the comment state, only the most recent state will be joined to the comment, while in case of the entries all of them are retrieved.

The method to retrieve the comments for a given change is located in the CommentsDAO class and shown in Listing 5.15. The *listComments* method returns an array of a triple containing the comment, state and selection. All three backend models will be combined into a single comment model in the core package. There are two methods to retrieve the comments. The first method needs only the change ID as parameter and fetches the most recent comment. The second method needs an additional timestamp to filter the comments which is extracted from the commit at which the comments are viewed.

```
public List<Triple<
  BackendComment, BackendCommentState, BackendCommentSelection
>> listComments(long changeId);

public List<Triple<
  BackendComment, BackendCommentState, BackendCommentSelection
>> listComments(long changeId, DateTime beforeTime);
```

Listing 5.15: Method signature of the listComments function located in the CommentsDAO class.

6 Evaluation

In this chapter, the requirements from chapter 2 are evaluated with their corresponding evaluation schema. The first 7 requirements were verified with a manual acceptance test in the browser. Starting point of all commenting requirements is the document view page. A document can only be created in the scope of a project. Therefore a project is created as first step and the document as second step. From this starting point, the additional steps to verify each requirement as well as the expected outcome are listed below.

1.1 Creating a comment

- Select a text range in the document
- Click on the speech-bubble icon within the appearing popup

A commenting form is shown below the text selection.

1.2 Viewing a comment

- Click on a text range which is referenced by a comment. A referenced text range is marked with a dotted underlining.
- Click on the speech-bubbles icon within the appearing popup

The comment is shown below the referenced text.

1.3 Replying to a comment

- Open an existing comment as described above.

At the end of the comment, a form to submit a reply is displayed.

2. Visualization of the connection between a comment and its referenced text range

- Open an existing comment.

The referenced text range of the viewed comment is highlighted with a blue background color. Furthermore, the comment view is displayed directly below the text range.

3. Marking a comment as resolved, rejected or closed

- View an existing comment.
- As project owner: The comment view includes a button to mark the comment as resolved or rejected.
- As comment starter: The comment view includes a button to mark the comment as closed.

After the user marked the comment as resolved, rejected or closed, the comment view shows an info box with the current state.

4. Comments trigger notifications

- Logged in as test user: Create a comment in the new project.
- Logged in as project owner: Reply to the comment.
- Logged in as test user: Mark the comment as closed.

The project owner receives two notifications - when the comment is created and when the comment is closed. The user who created the comment receives one notification, that a user replied to the comment.

5. Viewing older revisions of a document

- Modify the document. (First revision)
- Create a new comment.
- Modify the document again. (Second revision)
- Create a reply in the comment.
- Switch to the revision (commits) page and select to view the second revision.

One comment is present in the document with the initial comment text. The reply is not visible.

- On the revision (commits) page select to view the first revision.

No comment is present in the document

6. The UI is able to handle a large number of comments

- Create multiple comments on similar text ranges.

The comment view offers buttons to view the next and previous comment at the top of the comment box. It is therefore possible to view all comments even if the text ranges overlap each other.

7. Comments are preserved in forks

- Create a comment.
- Fork the current project.

The document within the forked project contains one comment.

All above listed requirements were tested successfully. The expected outcome could be observed in the browser. Those requirements are therefore fulfilled.

8. Store comments in the document data format WOM

As outlined in this thesis, storing comments in the document itself are disadvantageous. Instead, the comments are stored separately to the document in the database. This requirement was not fulfilled.

9. The UI relies on recognition rather than recall

Creating and viewing comments requires prior knowledge since a click or text selection is needed to activate the commenting popup. The initial step to access the commenting functionality therefore relies on recall. A solution would be to offer an alternative way to create a comment e.g. by adding a dedicated comment button to the action navigation. Another improvement would be to offer a tutorial to new users which explains the comment functionality. All other components of the commenting UI rely on recognition, since they are reachable through a directly visible button.

10. Easy to use UI

All parts of the UI offer "informative feedback" since all network requests will be represented by a loader in the frontend. Furthermore, all components are equipped with error handling. In case that an error occurs, the frontend will display a description of the problem. The commenting system also allows comment starters to close comments which were opened by mistake.

7 Future Work

As outlined in this thesis, two aspects of the current commenting functionality could be improved. First of all, the concept of comments in a multi-synchronous environment could be extended. Currently, comments are visible within fork with its replies disabled. However, the UI lacks the option to exclude comments in a fork and the possibility to delete all comments that originated from a different project. Furthermore, for certain cases, it can be useful to be able to preserve comments on merges. Another improvement which was presented in this thesis is the integration of fragmented references. However, fragmented references might have a negative impact on the usability. More research is therefore needed to determine if fragmented references provide a benefit for users. An alternative solution would be to store the original text reference along the comment. Users would see the current text range highlighted in the document, but are able to check the original text via a button.

Beyond those additions, comments could be integrated closer into the multi-synchronous workflow As discussed in this thesis, comments can be seen as a starting point of collaboration. As improvements are discussed in a comment, the comment could be used to simplify the process of making a fork, applying changes and submitting the merge request. Each comment could include a button to make modifications and propose the discussed changes. As the user clicks on this button, a simplified collaboration workflow is started. The UI could show the visual editor and the selected comment side-by-side, enabling the user to recheck the discussion. As the user finishes the changes, the system would generate a merge request automatically referencing the specific comment. This approach would greatly simplify the collaboration process for novice users and cloud therefore increase the number of collaborators.

8 Conclusions

In this thesis, I designed and implemented a commenting functionality for Sweble Documents, summarized UI best-practises relevant for the design of the commenting functionality and presented existing commenting solutions. The final commenting design combines ideas from existing solutions with new concepts as e.g. the placement of the commenting UI directly in between text passages. The commenting UI combines some of the best interface features in order to provide a satisfying user experience. Furthermore, I developed a concept to store and identify text ranges in WOM documents with the help of node IDs and offsets. The implementation consists of an algorithm to analyze changes in a document and to update the text ranges referenced by comments appropriately.

While this thesis focused on Sweble, the approach can be adopted by other wiki systems. Since WOM, the format used by Sweble, is a representation of Wikitext, most of the concepts can be transferred easily. The largest wiki Wikipedia lacks a modern communication channel for collaboration. A commenting functionality as presented in this thesis could simplify collaboration and could lead to an increased participation.

Acknowledgments

I would like to thank my supervisor Dipl.-Inf. Hannes Dohrn for his guidance in the development and writing phase of my thesis as well as for his detailed explanations of existing concepts in Sweble.

I am very grateful to my father Michael Joachimski for his support and feedback while writing this thesis.

References

Archibald, J. (2015). Tasks, microtasks, queues and schedules. Retrieved from https://jakearchibald.com/2015/tasks-microtasks-queues-and-schedules/

Dohrn, H. & Riehle, D. (2011a). Design and implementation of the sweble wikitext parser: Unlocking the structured data of wikipedia. In *Proceedings of the 7th international symposium on wikis and open collaboration* (pp. 72–81). ACM.

Dohrn, H. & Riehle, D. (2011b). *Wom: An object model for wikitext* (tech. rep. No. CS-2011-05). University of Erlangen, Dept. of Computer Science.

Dohrn, H. & Riehle, D. (2014). Fine-grained change detection in structured text documents. In *Proceedings of the 2014 acm symposium on document engineering* (pp. 87–96). ACM.

Molich, R. & Nielsen, J. (1990). Improving a human-computer dialogue. *Commun. ACM*, *33*(3), 338–348. doi:10.1145/77481.77486

Molli, P., Skaf-Molli, H., Oster, G. & Jourdain, S. (2002). Sams: Synchronous, asynchronous, multi-synchronous environments. In *The 7th international conference on computer supported cooperative work in design* (pp. 80–84). IEEE.

Myers, E. W. (1986). Ano (nd) difference algorithm and its variations. *Algorithmica*, *1*(1-4), 251–266.

Nielsen, J. (2005). Ten usability heuristics. http://www. nngroup. com/articles/ten-usability-heuristics/(acc-essed ...

Shneiderman, B. & Plaisant, C. (2010). *Designing the user interface: Strategies for effective human-computer interaction.* Pearson Education India.