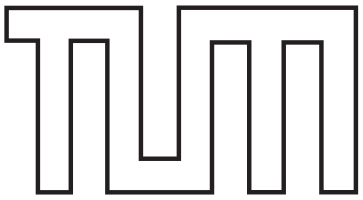
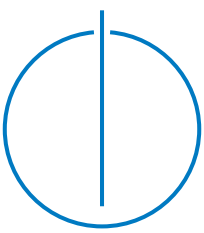


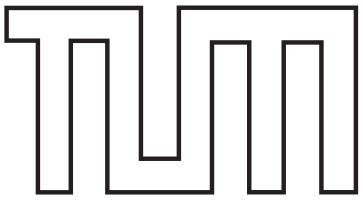
Fakultät für Informatik
Der Technischen Universität München

Master's Thesis in Informatik

Enabling realtime collaborative data-intensive Web Applications - A case study using server-side JavaScript

Tobias Höfler





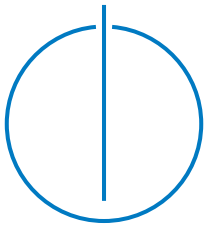
Fakultät für Informatik
Der Technischen Universität München

Master's Thesis in Informatik

Enabling realtime collaborative data-intensive Web Applications - A case study using server-side JavaScript

Realisierung von Echtzeit-Kollaboration in datenintensiven Web Applikationen - Eine Studie mit serverseitigem JavaScript

Author:	Tobias Höfler
Supervisor:	Prof. Dr. rer. nat. Florian Matthes
Advisor:	Sascha Roth, M. Sc.
Submission Date:	15.05.2013



I assure the single handed composition of this master's thesis only supported by declared resources.

München, 15.05.2013

Tobias Höfler

Abstract

The purpose of this Thesis is to analyze, how realtime collaborative data-intensive web applications can be enabled with particular reference to server-side JavaScript.
First, a survey was conducted to analyze the role of server-side JavaScript in organizations. It turns out, that the general satisfaction with JavaScript is quite high and Node.js, a prominent example of server-side JavaScript, is already widely known. It is shown, that Node.js is typically used to implement web applications. The participants additionally confirm, that Node.js is already enterprise ready and that its importance will rise in future.
Furthermore, a collaborative editor is designed, that enables the collaborative editing of text and provides collaborative features like chatting or inviting other users to work collaboratively on the same document. It follows the principles of the realtime architecture, which is presented in this Thesis. It is shown, that this architecture is a good fit for enabling realtime collaboration. In addition, this design is implemented prototypically with technologies like Ember.js, Node.js, socket.io and share.js. At last, this Thesis describes approaches to enhance the collaboration to rich text and arbitrary models.

Contents

List of Figures

List of Tables

Listings

1. Introduction

The Internet is important and a necessary part of our lives. A study of Germany's National Association for Information Management, Telecommunication and new Media (BITKOM) showed, that more than the half of the german population cannot imagine a life without the web[1].
A major reason for this is the changing interaction model of the Internet. In the past it was website-centric. A user, interested in a particular information, started the browser and opened a web-page to retrieve them. To collaborate on an online forum, she filled a form within a website and submitted the newly produced information. The website itself was the central part of the interaction and provided mostly a static set of content. This model shifted to a user centered one, whereas every aspect of the interaction starts and ends at the user. The content itself is not defined by the website any more but rather by the users. They define the information they are interested in and get informed when they occur so which leads to a necessity of sending the data actively to the users. This paradigm-shift can be seen through the whole Web. Prominent examples are social networks like Facebook or Twitter. [Bu11a] showed that one fourth of the internet users are using their favorite social network between one and two hours per day and even eleven percent use them more than two hours per day. Therefore, social networks play an important role in our lives, which actually a clear example for the user centric interaction model. The development of mobile applications running on a smartphone is another example. The applications send their information directly to the user, respectively the smartphone. This model shift also leads to a realtime user experience, as the user receives information when they occur, instead of looking manually for them[2]. [Is11] even describes these types of applications with the term "Data-Intensive Real-Time Applications" (DIRT) which was coined by Brian Cantrill, Vice President of Engineering at Joyent Inc. These applications have in common, that there is a lot of data to be processed and additionally, the

[1] cf. [Bu10]
[2] cf. [Ro10, p. 2]

users need to be notified about them in realtime which is the case for example for Twitter or chat applications[3].

For the realization of web applications, JavaScript plays an important role. Gartner Research denotes JavaScript in an analysis of 2013, as a "safe and preferred (virtually unavoidable for Web projects) option for nearly all nontrivial Web application development efforts" and advices developers focusing on next-generation HTML5 to "build a deep knowledgebase surrounding JavaScript to be successful"[4]. Also other sources show the importance of JavaScript. It is, for example, the most popular programming language at GitHub[5] and it is placed within the top 20 of O'Reilly Publishing's computer books sales in 2012 with a steadily increasing volume since 2004[6]. JavaScript itself mostly runs in browsers, interpreted by browser-specific JavaScript interpreters. Albeit recent trends move JavaScript to server-side application development[7]. A prominent example for this trend is Node.js[8]. Its event-driven approach makes it attractive for server programming and the handling of a lot of concurrent connections, which is important especially for realtime applications[9]. Furthermore, it provides a variety of different third-party packages through its own package manager. Nevertheless, Node.js is still a quite young project; it even did not mature to a stable 1.0 release.

The rise of the Internet's user-centric interaction model with its realtime characteristics and the emerging technology of server-side JavaScript, leads to the main research question of the Thesis:

Is it possible to enable realtime collaborative data-intensive web applications with server-side JavaScript efficiently?

This question implies whether the application of server-side JavaScript is useful for this scenario and if it can be used in enterprises, yet. Furthermore, there is the necessity to investigate in the feasibility of using server-side JavaScript to enable realtime collaboration.

[3] cf. [Is11]
[4] cf. [Ga13, p. 31]
[5] cf. [Gi13]
[6] cf. [Ga13, p. 30]
[7] cf. [Ri10, p. 1]
[8] cf. [Jo13a]
[9] cf. [Is11]

To analyze these questions, this Thesis is structured in the following way. Chapter 2 describes the fundamentals, needed for the Thesis, including an introduction of collaborative writing, different transport technologies and Node.js. For analyzing the role of server-side JavaScript in organizations and to assess its enterprise readiness, a survey was conducted. The results of it are presented in Chapter 3.
In Chapter 4, a design for a collaborative editor is developed to show, what typical design decisions are made for realtime collaborative web applications. Following this design, a prototypical implementation of this tool is presented in Chapter 5 to analyze the feasibility of implementing such an application with Node.js.
Chapter 6 discusses methods to work collaboratively on arbitrary models to present a way of enhancing collaboration to more complex scenarios. At last, Chapter 7 gives a conclusion and shows topics for further research.

2. Foundations

This chapter builds the theoretical foundation for this Thesis. Section 2.1 introduces *collaborative writing* and presents the current industry standard technology for helping to enable collaboration within applications. Technically, there are some special requirements for realtime web applications. Section 2.2 introduces the underlying transport technologies that are necessary for realtime communication. For getting an overview about the current market situation of realtime web collaboration tools, Section 2.3 presents a market analysis and compares some of the current tools. At last Section 2.4 introduces Node.js, a currently emerging technology using server-side JavaScript, which is used for building the prototype, presented in Chapter 5.

2.1. Collaborative Writing

In a globalized world, collaboration gains an increasing importance, with the Internet as its enabler. Especially *Collaborative Writing* has enormous benefits for working within groups. Different authors have always different viewpoints which have to be discussed within the team. This also leads to a common understanding and agreement about the written content[10]. The term *Collaborative Writing* itself has no generally admitted definition, but on the contrary, there are a lot of different words describing similar processes in research[11]. There is also no single clearly defined writing process, instead, there exist multiple different ones. Therefore, there is a necessity of clarifying and defining the term and the processes, so called strategies, more detailed for creating a common understanding and nomenclature.

Collaborative writing involves multiple people and mostly describes the "process of writing as a group"[12]. This group must agree in advance to a common

[10] cf. [Lo04, p. 67]
[11] cf. [Lo04, p. 71]
[12] cf. [Lo04, p. 70]

strategy for the writing process. Section 2.1.1 introduces different writing strategies and builds a taxonomy about the different ones.

2.1.1. A taxonomy for collaborative writing strategies

Collaborative writing can be done in a lot of different ways and is described differently by researchers. These different ways refer to the "Collaborative Writing strategy". [Lo04] defined it out of former definitions as "a team's overall approach for coordinating the writing of a collaborative document". This section presents a taxonomy of different collaborative writing strategies for giving an overview of the existing ones as well as introducing the one which is followed by this Thesis.

[Lo04] distinguishes between five major collaborative writing strategies that are presented in Figure 2.1. Four of them are collected in two groups.

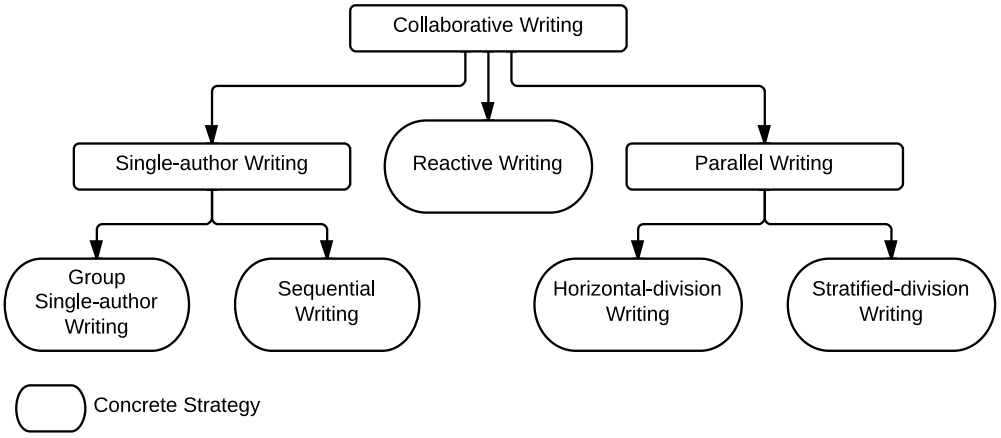


Figure 2.1.: Overview of Collaborative Writing Strategies

Single-author writing strategies are strategies where only one person is actively writing at the same time. When using parallel writing strategies, the document is divided into concrete units whereas each writer can work on such a unit in parallel. Reactive writing does not need any pre-planning and is done ad-hoc in realtime. The concrete strategies are described in the subsequent sections.

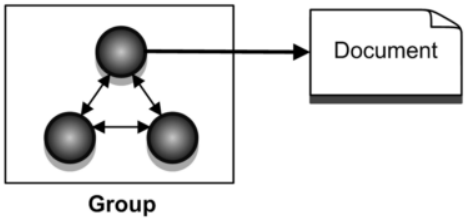


Figure 2.2.: Group Single Author Writing[14]

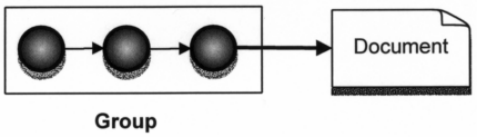


Figure 2.3.: Sequential Writing[15]

2.1.1.1. Group Single Author Writing

The so-called "group single-author writing" is shown in Figure 2.2. Here, the team chose one team member for doing the writing whereas the group discusses what shall be written. This is suitable for smaller groups and simple tasks like the processing of meeting minutes. As only one person writes the whole document, there is no conflicting styling. On the other hand, this method may not results in the groups purpose and reflects more the intention of the single writer[13].

2.1.1.2. Sequential Single Writing

The second strategy of single author writing, the "sequential-writing", is more common. It is presented in Figure 2.3. Here, every author writes his own part and forwards it to the next author, for writing her task. An advantage of this method is, that the persons involved in the writing must not set up a meeting as they can complete the document iteratively, so, there is less organizational effort. At the same time, this can be a downside as there is no common agreement about the content as social interactions are lower than within a meeting. The second disadvantage is, that subsequent writers can easily overwrite the work which was done by the previous ones. Thirdly, one writer has an enormous influence to the whole writing process, meaning, that he can delay the completion of the document, whereas the last writer has enormous control about the final version of the document. Despite of that, it is also difficult to divide the work correctly and to find the right order of writers[16].

[13]cf. [Lo04, p. 81]
[15]cf. [Lo04, p. 76]
[16]cf. [Lo04, p. 76-77, 81]

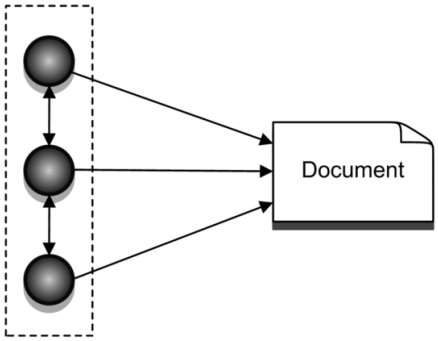


Figure 2.4.: Parallel Writing[17]

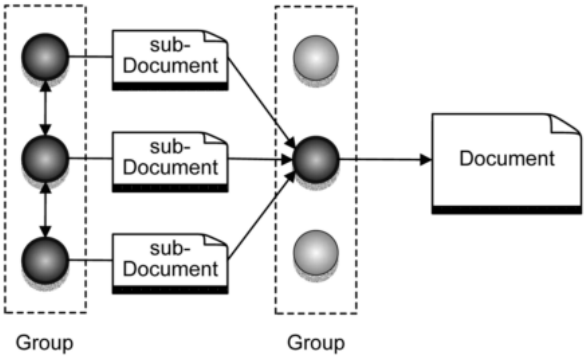


Figure 2.5.: Horizontal-division Writing[18]

2.1.1.3. Parallel Writing

Another group of strategies of collaborative writing is the "parallel writing" which is shown in Figure 2.4.

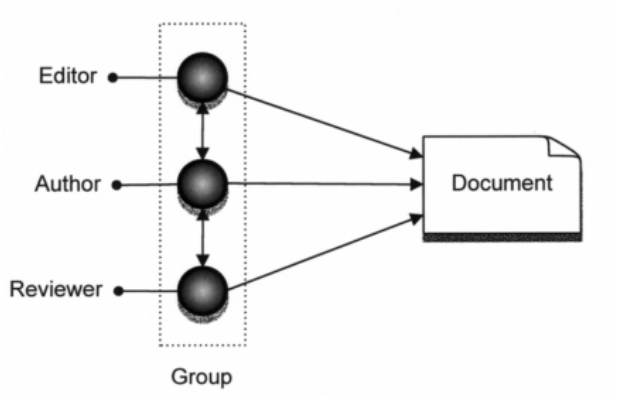


Figure 2.6.: Stratified-division Writing[19]

It includes the "horizontal-division writing" and the "stratified-division writing". A team, using these strategies must split the work into parts which are processed by the different persons of the group afterwards. In contrast to the sequential writing, where the work is divided, too, the persons can work in parallel here which makes these methods more efficient related to the produced content per time. The downsides of them are similar to the ones of the sequential writing, especially a poor social interaction. Additionally, different authors may do a different styling of their content and may produce redundant content. For using these strategies, special collaborative writing technology is needed[20].

The most used strategy of parallel writing is the "horizontal-division writing" which is shown within Figure 2.5. Here, each author produces a distinct part

[18]cf. [Lo04, p. 77]
[19]cf. [Lo04, p. 79]
[20]cf. [Lo04, p. 77-78, 81]

of the document and all parts together build the final document afterwards. The other parallel writing strategy is the so called "stratified-division" where not all members of the group contribute content directly to the document, but take a special role like "author" or "reviewer", depending on their competences. This strategy is illustrated within Figure 2.6.

2.1.1.4. Reactive Writing

The last form of collaborative writing is the "reactive writing" which is shown in Figure 2.7. If this strategy is applied, every participant contributes in real-time. It is called "reactive", because each author reacts to changes and additions of the other authors. This form needs no planning or structuring in advance. In contrast to the previously presented strategies, this method creates a common agreement about the content. The main disadvantages are a complicated coordination and version controlling[22].

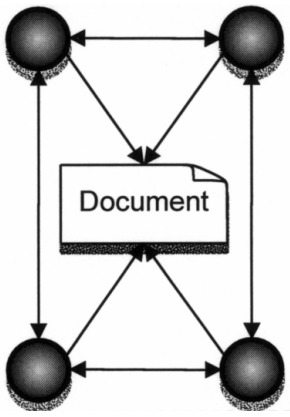


Figure 2.7.: Reactive Writing[21]

2.1.2. Operational Transformation

Operational Transformation (OT) is a technology, helping to enable collaborative applications by offering a lot of different concepts in the field of collaboration. First, OT was introduced in 1989 by C.A. Ellis and S.J. Gibbs in their paper "Concurrency Control in Groupware Systems"[23]. Followed by a lot of research, some improvements and corrections in the subsequent years, it became a core technology for collaboration in industry, being used by a lot of popular collaboration tools like GoogleDocs, IBM OpenCoWeb, Novell Vibe or Codoxware[24].

The basic idea of OT is illustrated in Figure 2.8 via a collaborative writing example.

[21] cf. [Lo04, p. 80]
[22] cf. [Lo04, p. 78-79, 81]
[23] [EG89]
[24] cf. [SWF12, p. 1391]
[25] cf. [Su11]

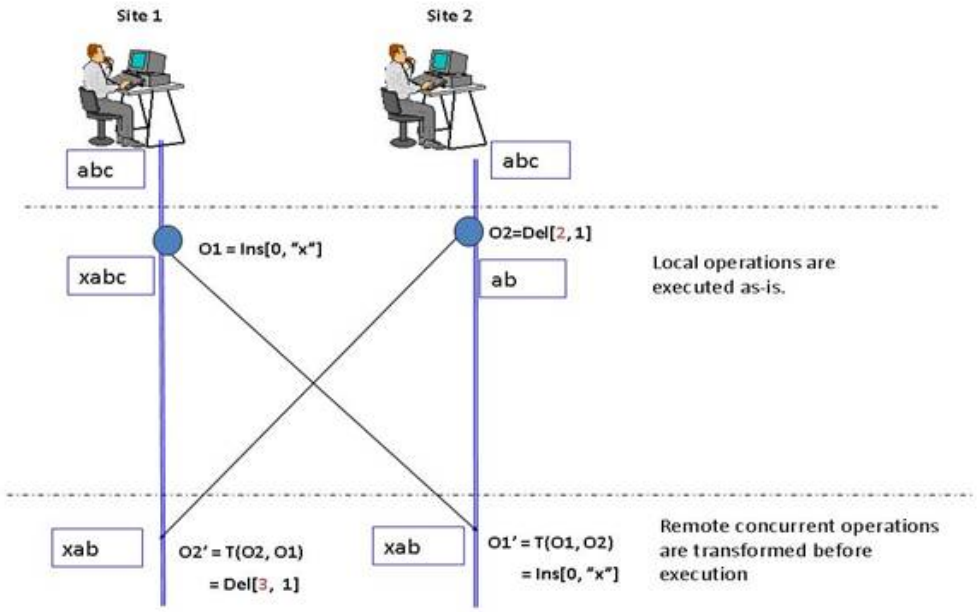


Figure 2.8.: Exemplary Scenario to show the basic OT Idea[25]

There are two persons shown, each one at a different location. For a better nomenclature, the person at site 1 is called $P1$ and the person at site 2 is called $P2$. Both partners start with the same document which contains just the three letters "abc". Now, $P1$ and $P2$ start to edit or manipulate the document. Such a document manipulation is represented by an operation. $P1$ adds the letter "x" at position 0 (operation $O1$) and $P2$ deletes the one letter "c" at position 2 (operation $O2$). Within an OT system, local operations are executed as they are and remote operations are transformed before their execution. This results at site 1 in an immediate execution of $O1$ and transformation of $O2$ and exactly the other way around at site 2. The transformation of an operation is necessary, to reflect the impact of an operation, executed before. In the example at site 1, $O2$ gets transformed to $O2' = Del[3, 1]$, to reflect the impact of $O1$, which was executed before. This concretely means an incrementation of the position for the deletion, as $O1$ added the character "x" before the original position of $O2$. The execution of a remote operation without this transformation would result in a falsely result. For example applying $O2$ at site 1 after $O1$ without a transformation would delete character "b" instead of "c" which would result in "xac". At site 2, $O1$ gets transformed too, to $O1'$ but it is just the same as $O1$, as $O2$ has no impact on $O1$. After the execution of the own and transformed,

remote operations on both sites, the document results in "xab" for $P1$ and $P2$, so, they have consistent documents.

There are two layers, which are responsible for these transformations. The one is the transformation control algorithm, which decides, when an operation is ready for transformation, which operations should be transformed and in which order the transformations are carried out[26][27]. The transformation itself is done via a transformation function. This function takes two arguments, the transformation target, which is the operation that will be transformed and the transformation reference, which is the operation that influences the target operation. The output of this function is the transformed target operation. A detailed description of the the transformation control algorithm or the transformation function including further references can be found in [Su11].

The above example was quite basic, and it wouldn't work without the so called *operation context*. Each operation is executed and also only valid on a specific state of the document, the operation context. Figure 2.9 shows a more complex example to explain the concept of the context.

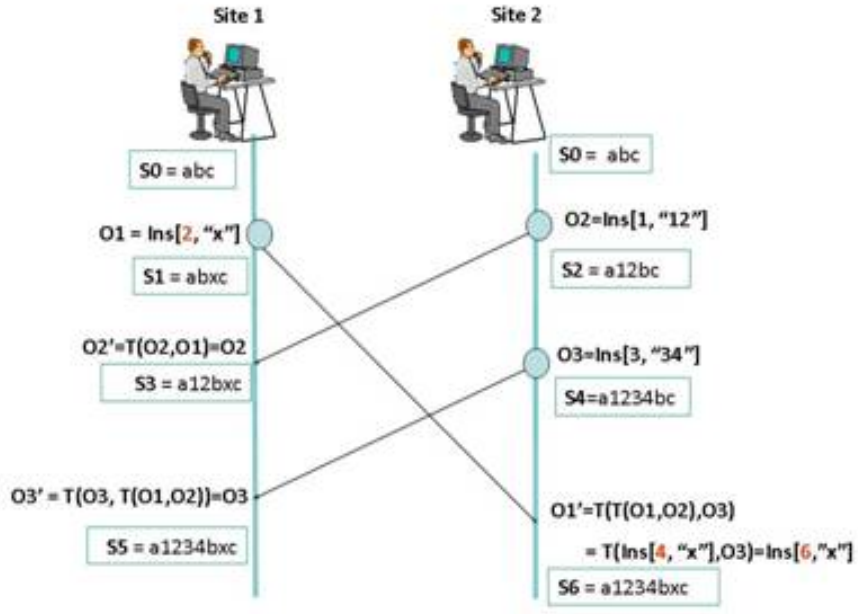


Figure 2.9.: Operational Transformation - Operation Context[28]

[26] cf. [Su11]
[27] cf. [Su04, p. 438]
[28] cf. [Su11]

Both sites again start with the same document, therefore also with the same document state or operation context $S0$. The operations $O1$ and $O2$ both are generated out of this state, so, their context is $S0$. After applying $O1$ the state of the document changes to $S1$ at site 1. At site 2, $O2$ changes the document state to $S2$ which is the context for the operation $O3$ generated afterwards and so on. Every generated operation refers to a specific context, which can only be changed by operational transformation.
This context builds the foundation for a correct application respectively transformation of each operation and leads to two fundamental conditions for the consistency maintenance of OT:

1. **An operation On can only be correctly applied, if these definition context is equal to the execution context**
   The original context of an operation is the so called "definition context", the context, where the operation is applied, is called "execution context". An operation On can only be correctly applied, if these two contexts are equal. The definition context of $O1$ in the example shown in Figure 2.9 is $S0$, which is the correct execution context at site 1 too. When $O1$ arrives at site 2, the execution context is $S4$, so if it would be applied, the resulting state would be "a1x234bc", which is not correct as the rule, mentioned above was not obeyed as $S0 \neq S4$. Instead, $O1$ must be transformed as shown in Figure 2.9 to $O1' = Ins[6, "x"]$, which can be applied now correctly on $S4$ and results in the same result as $O1$ would produce in $S0$.

2. **Two operations can only be transformed if they are defined in the same document state**
   In Figure 2.9, $O1$ and $O2$ are generated out of the same document state $S0$, so they can be transformed with each other correctly as they refer to the same positions within the text, which is necessary for a correct application. A contrary example are the operations $O1$ and $O3$, which does not have the same context ($S0$ and $S2$), and therefore cannot be transformed with each other directly. Instead, $O1$ must be first transformed with $O2$ resulting in $O1'$, which has the context $S2$, just the same as $O3$, therefore, they can be correctly transformed now.

Despite of consistency maintenance of shared documents between multiple concurrent editors, OT provides more capabilities like an undo-functionality, a

"workspace awareness" in 2D and 3D workspaces, a locking functionality or operation notification mechanisms[29].

2.2. Transport Technologies

This section presents technologies for enabling realtime communication between client(s) and a server. From the beginning of the Hypertext Transfer Protocol (HTTP) until the time of writing this Thesis, there was a lot of development in this field. Figure 2.10 presents the milestones of this development in form of a timeline. For sure, these milestones influenced each other. All of them are based on the Transmission Control Protocol (TCP), that ensures a reliable connection between two computers. The development of Ajax and Comet for example was only possible through the development of HTTP 1.1.

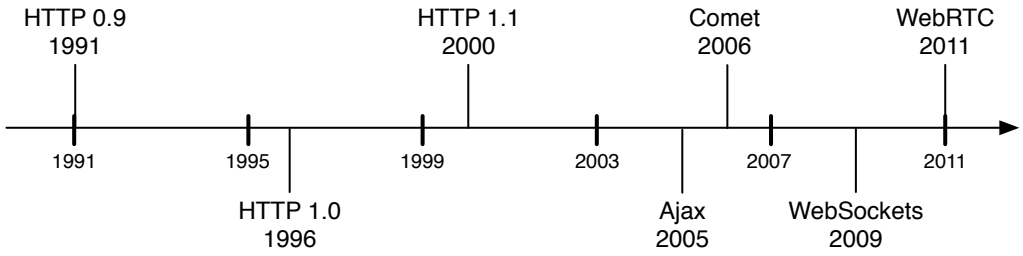


Figure 2.10.: Important Developments for realtime Web Applications

In the past, the Internet was mostly website-centric (see Chapter 1), which was also the reason for HTTP being designed as a "protocol for retrieving documents"[30]. This design decision led to two fundamental characteristics for its first specification 1.0 (version 0.9 can be seen as a prototype version of HTTP[31]). As a website visitor is interested in a specific document, she sends a request to the server, therefore, the communication is always started by the client and not by the server. For just delivering documents, the server does not need to know any meta information about the user, requesting the document, therefore the second important characteristic was, that the server does not save

[29] cf. [Su11]
[30] cf. [MC08, p. 2]
[31] cf. [GT02, p. 16]

any state about its clients[32]. A typical communication process with HTTP can be seen in Figure 2.11.

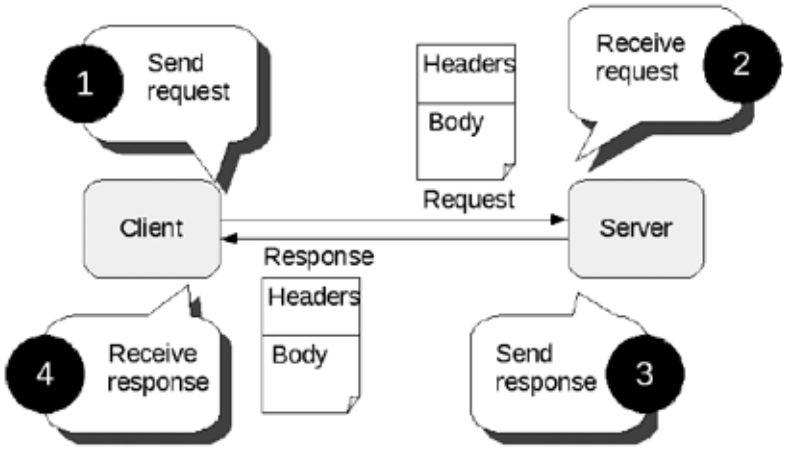


Figure 2.11.: HTTP Communication process[33]

The client sends a request to the server which then sends the respective response without saving any information. So, the communication is client-initiated and stateless[34].

As the web changed from a document-centric to a user-centric model (see Chapter 1), this paradigm was not fitting any more. There was a need for communications being initiated from the server and not only from the client[35]. An important improvement for further developments were the "Persistent Connections" of HTTP's specification 1.1, which is the current version of HTTP[36]. This connection type allows, to keep the underlying TCP/IP connection open after an HTTP transaction is completed, for a reuse of this connection[37]. This paved the way for Ajax, which came out in 2005. Ajax was mostly coined by Jesse James Garrett who described in an article a way, to send requests to a webserver in the background, without a full refresh of the opened webpage[38]. This was a very important step for enabling a lot of new experiences within the web. Now, it was possible to create dynamic webpages with immediate

[32]cf. [MC08, p. 2]
[33]cf. [MC08, p. 3]
[34]cf. [Th99, p. 4]
[35]cf. [MC08, p. 3]
[36]cf. [GT02, p. 16]
[37]cf. [GT02, p. 91]
[38]cf. [MC08, p. 1-2]

user response, like feedback messages or content, which is loaded on demand, or an automatic appearance or removal of whole parts of the website[39]. Although Ajax made big improvements for a realtime user experience within the web, it still followed the pull interaction model which means, that the browser pulls the content from the server, so, the communication between them is still initiated by the client [40]. This model is not sufficient for realtime scenarios, for example a progress bar, where you see the current percentage of already uploaded megabytes of an image, which is sent to an online web album. An even more important example would be a monitoring tool, which sends alerts if a specific temperature is reached and an immediate reaction is needed. Within these scenarios, the server has new information, which must be propagated to the client(s) and here sometimes it is not fast enough, to wait for the next request done by the client. Therefore more advanced techniques are needed, where the server initiates a connection to a client to send new information at any time.

Within this chapter, some of these techniques required for realtime scenarios are described. Figure 2.10 illustrates a timeline with all the presented technologies and their year of appearance.

2.2.1. Polling and Piggibacking

The most simple method to introduce a realtime user experience is the so called polling. Polling means, that the client sends requests to the server regularly in the background by using Ajax mechanisms, looking if there is new information available. The rough sequence is shown in Figure 2.12.

The first poll or request offers no new information for the client, therefore, there is just a normal response from the server. Within the second poll, the server has new information which are sent via the response. For instance, in chat application, which allows users to send messages to each other, this could be one message.
As it can be seen in Figure 2.12, the period between two polls is a tradeoff between actuality or technically spoken response-time and communication and resource effort. Keeping the time between two polls short means, that updates

[39] cf. [Po07, p. 1]
[40] cf. [Ro10, p. 4]

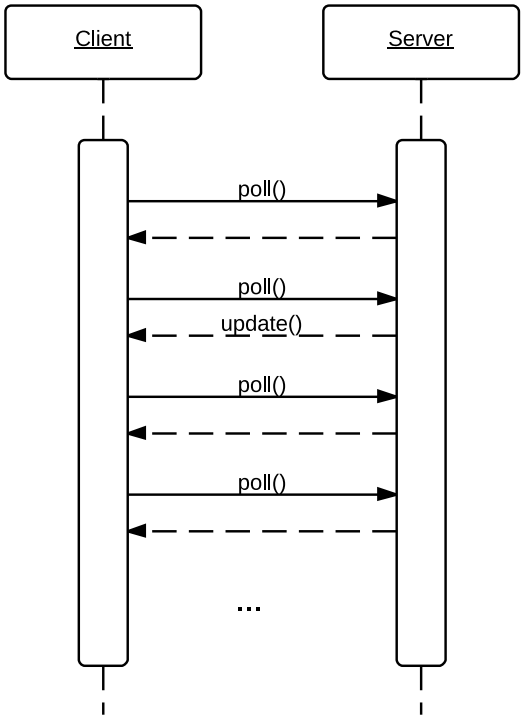


Figure 2.12.: Polling Process

like a new chat message arrive earlier at the client which leads to a better realtime experience. The disadvantage of this approach is that a lot of messages must be sent which leads to a higher load for both, the client and a server and also for the underlying network. In the example of figure 2.12, most of the exchanged messages are not needed, as there is no new information available. If, for example, the server is connected to a sensor which always has new information after a specific timeframe, this approach fits good as this timeframe can be synchronized with the clients interval for sending requests to the server.

Another simple approach is the so called "Piggybacking", which makes use of a user initiated action like *update*, *delete* or *create*. Usually, the response to such actions is just sort of an acknowledgement of receipt. When using Piggybacking, such responses are enriched with update information, also to other models not related to the currently requested one.
This method can also optimize polling. If a normal answer to a user request, not belonging to the poll message flow, is enriched with further data via piggybacking, the time to wait until the next poll is reseted. Therefore, Piggy-

backing can decrease the number of polls and make the whole communication more efficient[41].

2.2.2. Comet

Section 2.2.1 showed, that the simple approaches to enable realtime communication in web applications are still based on communications, initiated by the client. This often does not meet the requirements for realtime scenarios in terms of latency. There might be a high delay between the occurrence of a new event and the time of realization at the client. This is, what Comet is addressing. The goals of Comet are enabling the server, to send data to the clients at any time, more efficient related to speed and scalability, than the Ajax based approaches described above. The term itself describes techniques, protocols and implementations to achieve these goals and is mainly coined by Alex Russel, the creator of the DoJo framework[42].
In contrast to the typical HTTP communication process shown in figure 2.11, where the client initiates the communication, figure 2.13 shows a comet communication process.

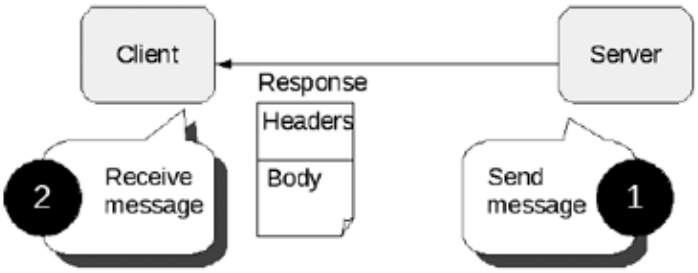


Figure 2.13.: Comet Communication Process[43]

This approach has no step, during which the client asks for updates. If the server holds new information for one or more clients, it just sends them to the respective recipients. The following sections describe different Comet technologies.

[41] cf. [MC08, p. 19-31]
[42] cf. [SA09, p. 109]
[43] cf. [MC08, p. 3]

2.2.2.1. Long Polling

Long Polling is a method, which fits to the goal of Comet, that the server can send new data to the browser at any time, although it doesn't fit directly to the comet communication process shown in figure 2.13. To enable Long Polling, the client or respectively its browser, sends a request to the server. Instead of responding directly with a concrete message which would end the connection just like it is the case with polling (see section 2.2.1), the server holds the connection open by responding either *"Transfer-Encoding: chunked"* or *"Connection: close"*. With this approach, the client waits for messages from the server, who can send the new data to the respective client, directly after it is available at any time. If the client received these new data, the connection is closed. For enabling this functionality over the whole time, the user visits the webpage, the browser opens a new connection to the server directly afterwards, so the server is again able to send new data when there is something available. This reconnection method is also used, if the connection is disrupted somehow. As mentioned above, the connection itself is still initiated by the client, but the server can send updates, directly when they occur. It also removes the unnecessary regular requests, compared to the polling method, which reduces the load for the client, the server and the underlying network.
This approach enables a realtime user experience but results in new requirements for the server. Especially traditional webservers are optimized for a fast opening and closing of connections, to handle incoming requests as fast as possible. Now, there is a huge number of parallel long-lived connections, which must be kept open. The Apache webserver for example, is optimized to handle about ten thousand parallel connections, but, for being able to realize a cost-efficient realtime enabling server-side, this one should be able to handle about fifty thousand long-lived connections, which is one requirement for a good Comet server[44].

2.2.2.2. Forever Frame

The first technique for realizing Comet was the forever frame, which relies on chunked-encoding, which is defined in HTTP 1.1's specification. Chunked-encoding divides the HTTP message body into different messages, so called

[44] cf. [SA09, p. 112-113]

chunks, which are sent independently. This can be used to transfer dynamically produced content[45] or large documents incrementally. In the context of Comet, a usually hidden iFrame refers to a chunked-encoded document, where the server can put in the new data, when necessary. Figure 2.1 shows such exemplary messages.

Listing 2.1: Example messages from the server[46]

```
<script>
parent.foreverFrame.callback("the first message");
</script>
<script>
parent.foreverFrame.callback("the second message");
</script>
```

Both are invoking a function on the client-side, with the message as parameter. The message itself can be for example "You have new messages in you inbox" and the method, which is invoked might display this message in an alert box. To avoid, that this iFrame becomes too large, the client must do sort of a garbage collection, e.g. remove DOM-nodes out if the iFrame, when they are not needed anymore like a script tag within figure 2.1 which is already fully processed.[47].

2.2.2.3. XHR Streaming

The XMLHttpRequest (XHR) is the common transport mechanism for polling and long polling and builds the foundation for all Ajax approaches. XMLHttpRequest itself was first developed by Microsoft as an ActiveX object, first called *Microsoft.XMLHTTP*, but the idea was shortly thereafter adopted by Mozilla, who developed an alternative called XMLHttpRequest. The XHR allows, to send requests to a server within the webpage fully in the background without the necessity of reloading the whole page or locking the page while waiting for the response[48].
Today, most of the current browsers like Firefox, Chrome, Safari and Internet Explorer 8 support the XHR, which can be used, to not just make requests in the background like Ajax but also to stream data from the server to a client at

[45] cf. [Th99, p. 24-26]
[46] cf. [SA09, p. 114]
[47] cf. [SA09, p. 113-115]
[48] cf. [Po07, p. 4]

any time without the necessity of a new HTPP connection for each new information. The streaming functionality can be realized via utilizing the different so called *ready states* of XHR, which are shown in Table 2.1.

Ready State	**Numeric Value**	**Description**
UNSET	0	The XMLHttpRequest object has been created
OPENED	1	The open method has been successfully invoked, which means among other things, that a valid HTTP method and a valid and resolvable url was set. Within this state, other HTTP request headers can be set and the request can be initiated, by invoking the send method
HEADERS_ RECEIVED	2	All redirects have been followed, if there were any and all HTTP response headers have been received. Some parts of the response are available now.
LOADING	3	The entity body of the response is being received
DONE	4	The whole data is completely transferred to the client or an error occurred during the transfer

Table 2.1.: Ready States of the XMLHttpRequest Object[49]

Within state 3, the client can access data, which is received before the whole response is sent and the HTTP connection is closed, which would be indicated by ready state 4. So, with using especially state 3, the server can send data at any time to the client, only within one HTTP connection. The downside of this approach for the server is almost the same as it is with long polling, but there are new difficulties for the client. As the client stores all data received, if too many messages are sent over one single HTTP connection, the browser on the client-side can get into memory issues. To avoid this, it is not recommendable to use just one single HTTP connection for all messages, but to close the current and open another one after a certain number of messages received from the server, just as long-polling is doing it for every message. Another difficulty for the client is, to get the complete messages out of the stream, which must be realized with a parser. For an easy utilization of XHR streaming, the client can take advantage out of the *onreadystatechanged*-events, which are fired always, if the ready state changed[50].

[49] cf. [W3]
[50] cf. [SA09, p. 115-116]

2.2.2.4. JSONP Polling

Sometimes it is important, to send requests to another server (on another domain) as the webpage was delivered from. For example it can be the case, that the website and other static content is just delivered by a normal HTTP server, and the realtime functionality is realized via a different comet server on a different domain. Another typical use case is the usage of third party services. Accessing a different domain than the website was delivered from refers to the *cross-domain* problem.
Polling, long-polling and XHR streaming are typically built on top of the XMLHttpRequest, therefore, the cross-domain ability of these methods highly depends on the cross-domain ability of the XHR, which normally doesn't allow cross-domain access. When using the forever frame method, and with some workarounds also with the XHR streaming, at least a cross-subdomain access can be realized, but still, this is no cross-domain. The technique which refers to this requirement is called *JSONP polling*. JSONP polling relies on the simple polling approach described in section 2.2.1, but instead of using the XMLHttpRequest object, it makes use of <script>-tags, which are appended to the document for each new request. If such a tag appears within the document, the browser loads and parses it. With JSONP, the response of loading this script is surrounded by a function, which can be called and returns the the loaded data. The script tags are normally used for loading scripts from any server. With JSONP polling, this characteristic can be utilized to making cross-domain requests[51] [52].

2.2.3. Flashsockets

Flash is a technology of Adobe Systems Incorporated, for producing animations, multimedia and interactive content for different devices[53]. For making Flash runnable on a client, a special software must be installed called FlashPlayer. Flash can be used alone but in combination with ActionScript, a programming language developed especially for Flash, running within the FlashPlayer on the client, more complex scenarios can be implemented like drawing programmatically, loading data or respond to user events.

[51] cf. [MC08, p. 40]
[52] cf. [SA09, p. 116-118]
[53] cf. [Ad12]

One advanced functionality, which Flash brings in combination with ActionScript are sockets that allow a bidirectional communication between the client and the server. Nevertheless, this technology is proprietary and the respective FlashPlayer must be installed on every machine, that want to use this communication mechanism[54].

2.2.4. HTML5 Websockets

"HTML is the publishing language of the World Wide Web."[55] The first version of HTML was developed in 1993, followed by three versions in the same year while the next to last version (4.01) was developed in 1999. In 2004, when the term "Web 2.0" became popular, a small group of people founded the "Web Hypertext Application Working Group" to develop a new HTML standard for enabling more powerful web applications. Despite of a lot of new features and APIs like Canvas, scalable vector graphics, audio and video or geolocation, one API was integrated in the new HTML5 specification which is quite important for enabling realtime communication in web applications, the HTML5 WebSocket API. HTML5 follows a plugin-free paradigm, meaning, that all features must be usable natively and without any plugin as it is the case for example, when using flash (see 2.2.3).
WebSockets realize a full-duplex connection, meaning the client, which is normally a web application and the websocket-server can send data at the same time to each other. WebSockets are lower-level networking interfaces that provide a bidirectional stream which arrive in the same order as it was sent just like TCP. In contrast to TCP, WebSockets connect to URIs instead of hosts and ports.

An example about how WebSockets can be used is shown in Listing 2.2.

Listing 2.2: An usage example for WebSockets[56]

```
1 var myWs = new WebSocket("ws://www.websockets.org");
2 myWs.onopen = function(evt) { alert("Connection open ..."); };
3 myWs.onmessage = function(evt) { alert( "Received Message: " + evt.data); };
4 myWs.onclose = function(evt) { alert("Connection closed.");
5 myWs.send("Hello WebSockets!");
6 myWs.close();
```

[54] cf. [Br10, p. 3, 561-570]
[55] cf. [W312]

WebSockets can be opened at the client side via JavaScript. In line one, the WebSocket is created and the communication is built to the given URI. From line two to line four, the evented nature of WebSockets can be seen quite good. They all add a listener to a special event("open", "message", "close"), where each time the listener is a function which is called, when the respective event is fired. To send a message just the "send" method has to be invoked as it can be seen in line five. The last line within the listing closes the connection.

Listing 2.2 showed an easy example of a client-server communication-process. It can be seen, that one of the other goals of HTML5, the "Simplification" was achieved for the WebSocket API. Doing such an easy communication example would have taken much more lines of code in the Comet approaches, described in Section 2.2.2.
Despite of the easiness of use, WebSockets have some other advantages over the previously introduced approaches. First, the connection is only built once in contrast to polling, which needs for each request a new connection or also long-polling or XHR streaming, which must build a new connection after a certain number of messages, therefore, there are a lot less connection handshakes needed. Furthermore, WebSockets offer a better latency behavior, which can be best seen compared to polling. Assume, the server wants to send some data to the client with a latency between the client and the server of 50ms as it is shown in Figure 2.14.

The upper timeline shows the communication process realized with polling. This is a half-duplex connection, therefore, the client must send a request for each new data, which makes the whole communication slower because of the latency, compared to the WebSocket approach. The client sends a request, which takes 50ms. Assume the server sends his data directly, after the request has received, which takes another 50ms until the data arrive at the client and he can send another request. So, we have a round-trip time of 100ms until the client sends another request. With WebSockets, we have a full-duplex connection. The server can send new data at any time. There is still the latency of 50ms, but as there is no request needed, the WebSocket round-trip only takes 50ms for one single message[58]. At last, the communication doesn't work on top of HTTP which leads to a reduced amount of data which has to

[56] cf. [Ka13]
[57] cf. [PL11, p. 169]
[58] cf. [PL11, p. 1, 5, 6, 161-169]

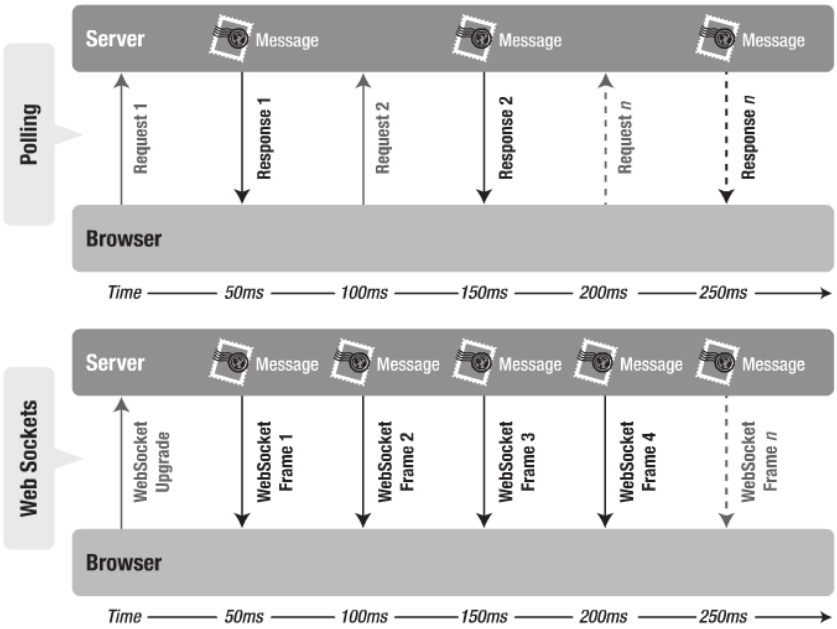


Figure 2.14.: WebSockets vs. Polling[57]

go through the network, as the unneeded HTTP-header information are not sent with messages over WebSockets.

2.2.5. Summary

In this chapter, the technical foundations for enabling realtime communication for web applications were introduced. It became apparent, that the first simple approaches shown in Section 2.2.1 were at least a step in right direction, but neither efficient nor enabling a good realtime user experience. The enhancements with long-polling, forever-frame or XHR streaming increased the efficiency and reactivity of web applications but there are still downsides e.g. the garbage collection, meaning, that the client has to take care of its memory usage. With JSONP-polling, a viable solution for cross domain Ajax was introduced, which has still the disadvantages mentioned just now.
Socket connections provide a better alternative for enabling realtime communication. They realize full-duplex connections and compared to the other methods the best realtime user experience. Flash sockets depend on the installation of the Flash Player plugin, which has some disadvantages however its distribution is quite high. HTML5 WebSockets are the newest development in this field. They have a lot of advantages over the other introduced methods

and run natively, without any plugin. The downside here is, that they are not available for all browsers currently.

2.3. Realtime Web Applications - A Market Analysis

This section gives an overview about currently available web-based collaboration tools. First, each tool is presented briefly whereas Section 2.3.6 compares their features and gives a summary.

2.3.1. Google Docs

Google Docs is part of Google Apps which is a "productivity suit" of Google Inc. It is realized by cloud-based web applications that enable users to access their data computer independently via a internet browser. Whereas Google Apps focuses on business and educational customers, there are a couple of applications also available for personal use, which applies for Google Docs, too.
Google Docs is a text-processing web application whereas Figure 2.15 shows an exemplary document created and edited with Google Docs.

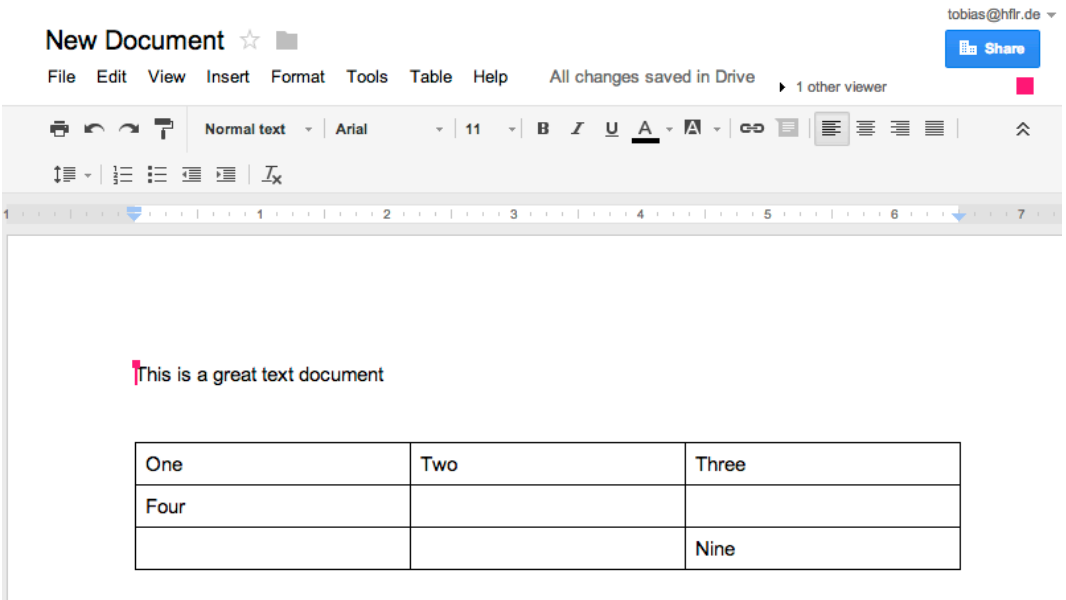


Figure 2.15.: Screenshot of Google Docs

Despite of rich-text-editing it offers functionalities like the embedding of tables, images or mathematical equations. Furthermore, it enables Google users to work collaboratively on the same document[59]. There are two people editing the document concurrently within the figure. The purple color marks the position of the cursor of the second user. Google Docs also provides a revision history which enables the user to restore different version of the document. Such a version is created regular automatically.

2.3.2. Etherpad (lite)

Etherpad lite is an "Open Source online editor providing collaborative editing in really real-time"[60]. It is the advancement of Etherpad and was fully re-written in Node.js[61]. Figure 2.16 shows an exemplary document within Etherpad lite.

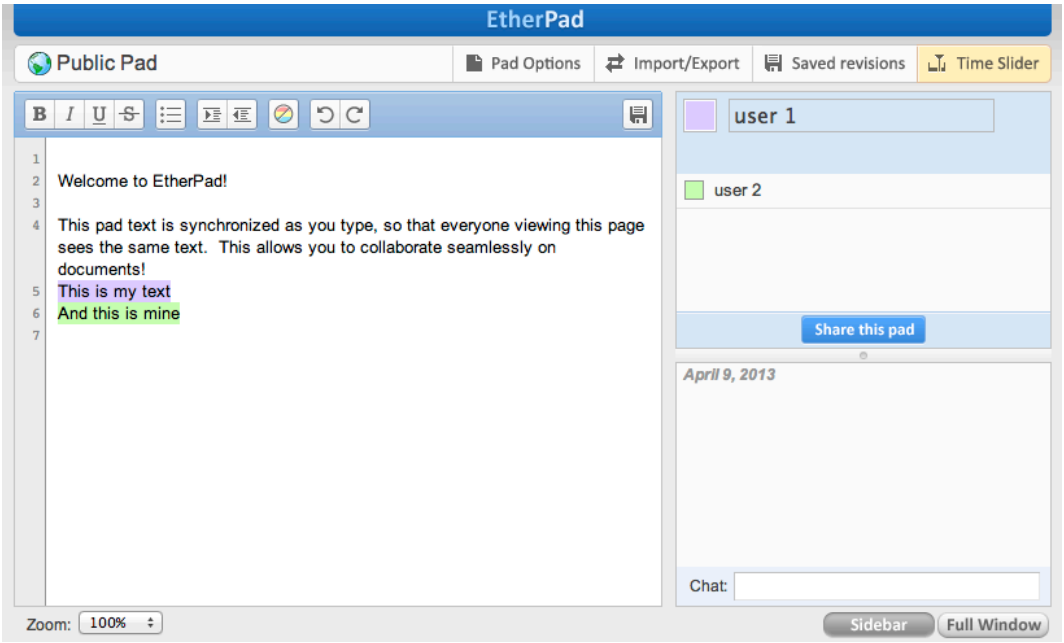


Figure 2.16.: Screenshot of Etherpad lite

Again, there are two authors editing concurrently within the document. Etherpad does not show cursor positions, instead, the text of each user is highlighted

[59] cf. [Go13]
[60] [Th13]
[61] cf. [Th13]

with a unique color for each user. It provides basic text formatting capabilities like making it bold, italic, underlined or strikes trough. For reviewing purposes, it provides a so called "Time Slider" which enables the user to go through all versions. Each time a user does a change on the document such a version is created.

2.3.3. ShowDocument

ShowDocument is a "Web Meeting and Document Sharing" web application. It offers various apps for working collaboratively like a shared online whiteboard, voice, video and text chat functionalities, screen-sharing capabilities and a collaborative editor for documents[62]. The last mentioned app is shown in Figure 2.17.

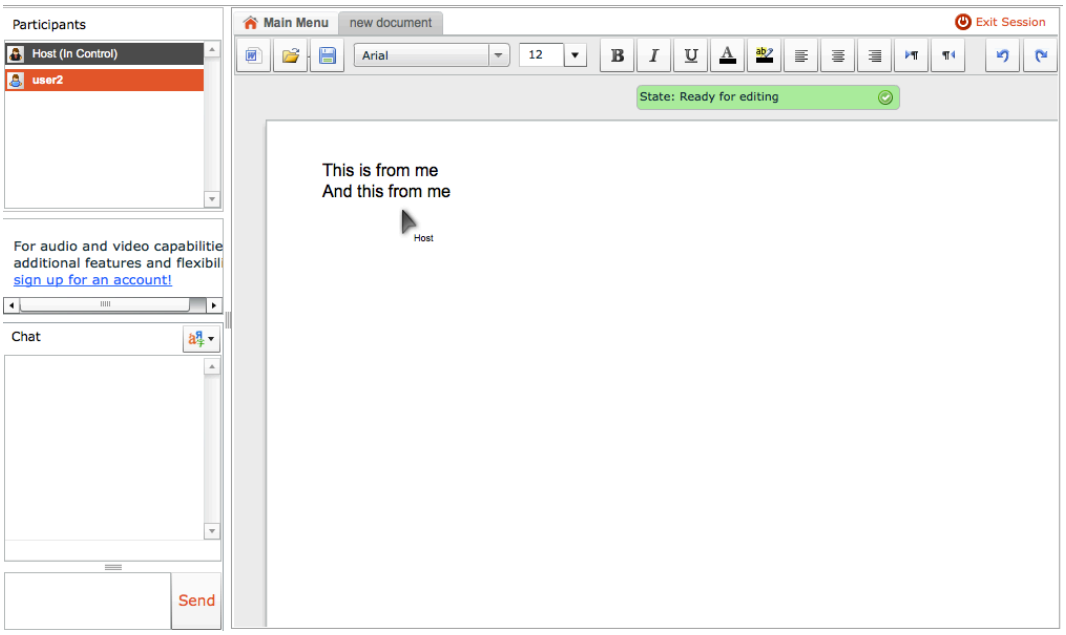


Figure 2.17.: Screenshot of ShowDocument

Again, there are two users editing the document. There is neither a cursor of the users visible nor is there a highlighting to mark which text was edited by whom. In contrast to the other tools, it provides a mouse-cursor synchronization. The text formatting includes the usage of different fonts and colors or the styling as bold, italic oder underlined.

[62] cf. [HB11]

2.3.4. Zoho

Zoho offers a lot of web applications for collaboration, business and productivity. Collaboration applications are for example are the chat-, meeting-, projects-, wiki-, mail or docs-app whereas we are interested in the last one. It enables their user to create and work collaboratively on document,s spreadsheets and presentations[63]. The respective word processor is shown in Figure 2.18.

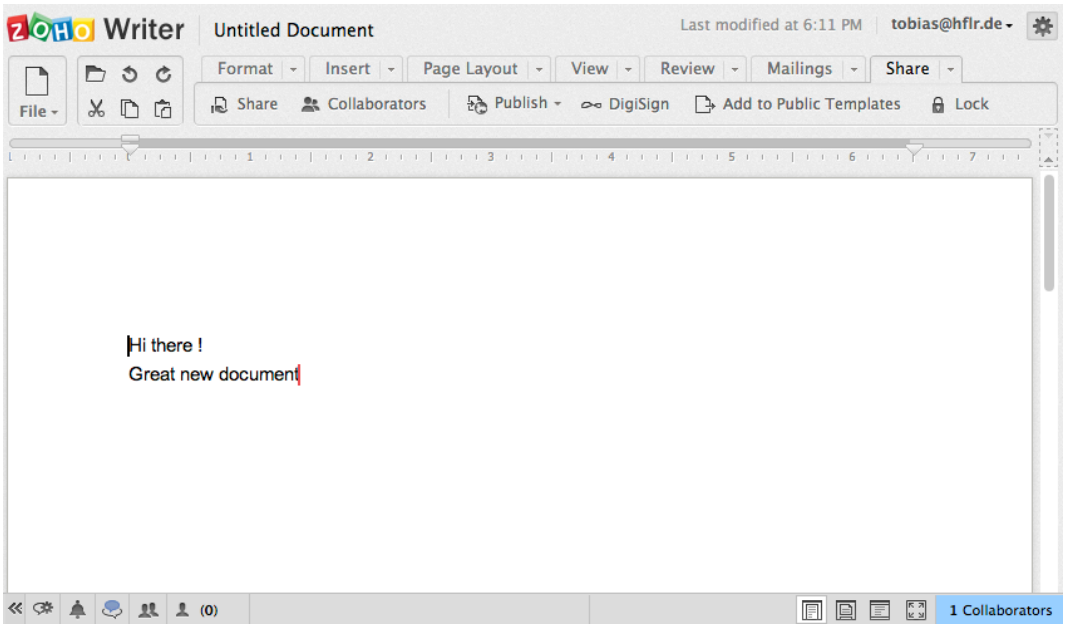


Figure 2.18.: Screenshot of Zoho

Within the figure, two users are editing the document collaboratively. As it can be seen, the cursors are marked with different colors. Zoho's word processor offer a variety of different text formatting capabilities and elements that can be integrated into the document like images, tables or mathematical equations.

2.3.5. Web IDEs

In the field of collaboration tools, there are a lot of collaborative IDEs. They focus on the development of source code for different programming languages and therefore enable mechanisms for a collaborative creation of source code.

[63] cf. [Zo13]

Examples are ShareLaTeX[64], Squad[65] or cloud9[66]. An example can be seen within Figure 2.19 which shows the cloud9 IDE.

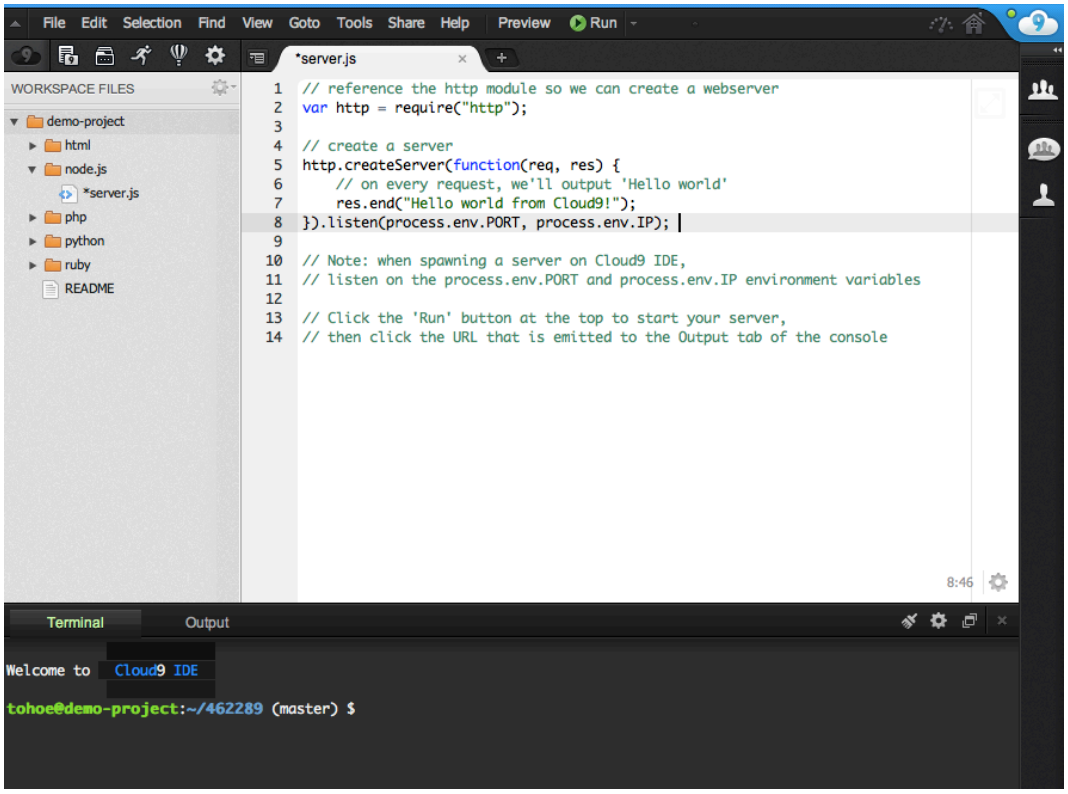


Figure 2.19.: Screenshot of cloud9 IDE

The all have in common, that they support collaborative editing, but do not support any rich text content or the embedding of tables or images which is clear for source code. They actually do support syntax highlighting for various programming languages and event further mechanisms like an automatic deployment of the code.

2.3.6. Feature Comparison and Summary

This section summarises the key findings about the above presented web collaboration tools. Table 2.2 shows metrics and how well they are supported by the respective tools. The Web IDEs are kept out of this comparison, because of their special focus on source code.

[64] [Sh13]
[65] [Co13]
[66] [Cl12]

	Google Docs	**Etherpad lite**	**Show-Documents**	**Zoho**
basic text formatting	●	◔	◑	●
advanced elements (tables, images)	●	○	○	●
cursors visible	●	◕	○	●
versioning	◕	●	○	◑

Table 2.2.: Summary about the collaboration tools and their features

Etherpad lite offers the least basic text formatting capabilities which are styling the text bold, italic, underlined our striked through. Furthermore, a colour and text indention can be set. ShowDocuments enhances this a bit by providing multiple fonts.Google Docs and Zoho in contrast provide a rich palette of text formatting capabilities. Additionally, they enable the integration of tables, links, images. Zoho is even able to automatically generate table of contents. ShowDocuments is the only tool, that does not display a cursor at all. Google Docs and Zoho show a coloured cursor of each user, whereas Etherpad highlights the text of each one.
The last criteria is the versioning. ShowDocuments does nod provide any versioning whereas Zoho offers the manually creation of different ones. GoogleDocs creates automatically versions in a regular basis. Etherpad lite has the most accurate versioning system. Each change, that is done within the document automatically creates a new version of it.

2.4. Node.js

Node.js is a JavaScript interpreter. It was developed by Ryan Dahl and first presented at JSConf EU in November 2009[67]. The most important thing about it is, that it offers a way of running JavaScript code outside of the browser. To do so it extends Google's JavaScript interpreter V8 with bindings to low-level APIs. This enables for example the working with processes, access to files or

[67][Da11]

the usage of network sockets with JavaScript.

Almost every binding of Node.js is asynchronous therefore Node.js never blocks. It is based on event handlers and follows an event-driven approach, just as JavaScript itself which leads to a good scalability and an effective handling of high loads[68]. In contrast to traditional programming, where the program does for example a database query and blocks until the answer is provided, Node.js registers a callback function that is invoked, when the database result is ready. In the meantime, Node.js can do other things. This behavior is realized with an event loop which does two functions in a continuous loop: the detection of events and the triggering of event handlers. This event loops runs in one thread within one process which results in the situation, that alway one event handler can run at the same time. Furthermore, this handler is not interrupted, it runs until it's completion[69]. A consequence of this is, that there synchronization between parallel executed code needed as this simply never happens.

Meanwhile, Node.js is sponsored by Joyent, a company specialized on cloud computing and realtime web applications[70]. Furthermore it gained acceptance in industry as it is used by companies like eBay[71], LinkedIn[72] or Yahoo[73].

2.5. Tricia

Tricia is an open-source Java platform developed by the chair for Software Engineering for Business Information Systems of the Technical University of Munich and commercialized by the infoAsset AG[74]. It is used for implementing enterprise web information systems and social software solutions like wikis, blogs, file shares and social networks[75]. Furthermore it offers hybrid wiki functionalities that extend Tricia's wiki component. Hybrid wikis enrich classic

[68] cf. [Fl11, p. 296]
[69] cf. [Te12, p. 16-17]
[70] cf. [Jo13a]
[71] cf. [EB11]
[72] cf. [Ch12]
[73] cf. [FR11]
[74] cf. [in12]
[75] cf. [BMN10, p. 1]

wiki software with a subset of semantic wiki features that allow to combine the textual content of a classic wiki with structured data[76].

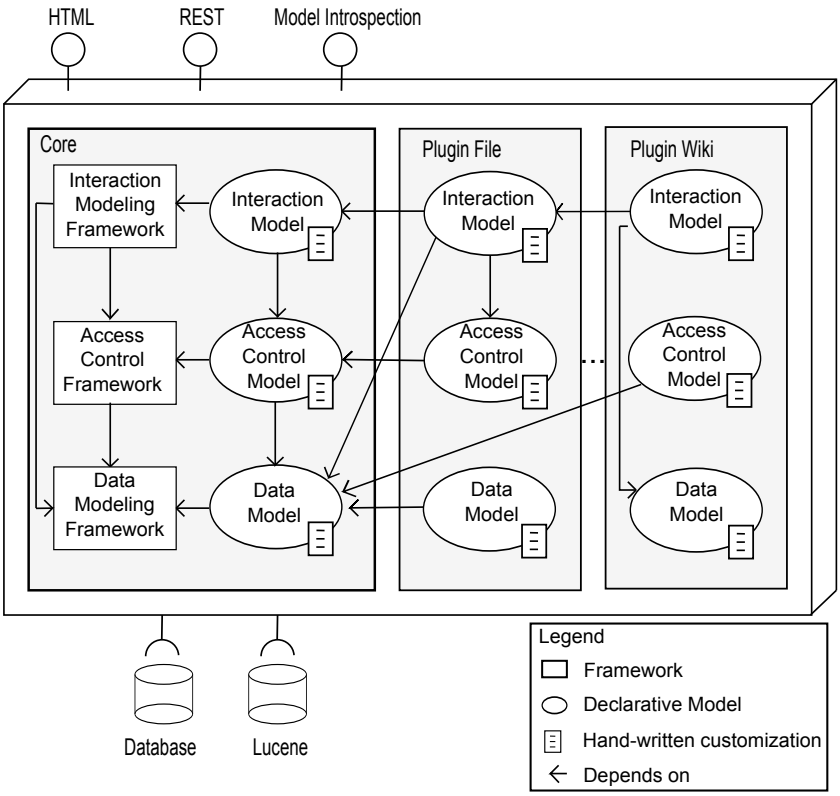


Figure 2.20.: Architecture of a typical Web Application implemented in Tricia[77]

A typical web application implemented on the Tricia platform can be seen in Figure 2.20. It consists of the so called core and plugins. The core is provided by the Tricia development team and offers functionalities which are needed by a majority of applications for example user profiles, groups, memberships, login or registration mechanisms. It is layered in frameworks for data modeling, access control and user interaction whereas each framework has its own underlying model. The plugins, used in the example in Figure 2.20 are "File" and "Wiki", whereas the "Wiki" plugin depends on the "File" plugin.

[76] cf. [MNS11, p. 1-2]
[77] cf. [BMN10, p. 3]

There are multiple ways to interact with Tricia as a user, realized by different interfaces. There is the possibility to use it via the web browser over HTTP(S), within a desktop application over SMB or WebDAV[78].

[78] cf. [in12]

3. On the Role of Server Side JavaScript in Organisations

JavaScript has an increasing popularity, as evidenced by being by far the most popular language at GitHub[79] and its usage for the majority of modern websites[80]. Additionally [O'12] did a survey, which analyses the GitHub and StackOverflow programming language popularity. JavaScript was ranked there at the first position, just before Java and PHP. Given the importance of JavaScript for Web development, especially at the client-side, there is also an emerging trend for using JavaScript for the full web stack, as new developments like Node.js or NoSQL-databases are showing. However, there is only little known about JavaScript's use and reputation in organizations, for both, the client- and the server-side.
This survey, conducted within the scope of this Thesis, tries to fill this gap.

3.1. Survey Related Work

There has been several studies conducted comparing dynamic type systems to static type systems. Parts of them directly relate to this survey, as JavaScript is a dynamically typed language.
[PT98] performed an experiment, where experienced programmers solved short programming tasks in both, a type-checking and a non-type-checking language. They figured out, that using type checking of interfaces, leads in a lot of cases to improved productivity and software quality and can result in a decreasing need of resources while the development. Also [Ga77] identified a positive influence of static type systems on development time. [SH11] refers to two unpublished works that showed, that a static type system leads to a faster usage of undocumented APIs. Furthermore, the identification and fixing of

[79] cf. [Gi13]
[80] cf. [Fl11, p. 1]

type errors within a software is easier. This finding is supported by [Kl12], who showed within an experiment, that a static type system decreases development time of programming tasks requiring fixing type errors. However, [SH11] mentioned, that there is no difference for semantic errors, compared to dynamically typed languages.

[Ha10b] conducted an experiment where 49 subjects were divided in two groups and had to program a simple Java parser, whereas one group used a statically and one used a dynamically typed language. In contrast to the findings above, this experiment showed, that statically typed languages do not have an obvious benefit over dynamically typed languages under all circumstances. [SH11] even showed within an experiment, that dynamically typed languages positively influences development time. This finding is supported by [Ha10a]. He did an experiment, where development time and the quality of the programmed code was measured for two groups. Both developed the same piece of software but one group used a statically typed language, whereas the other group used a dynamically typed one. It turned out, that there was no significant difference between these groups related to the resulting code quality but the development time, was significantly lower for the group, using the dynamically typed languages. It is noticeable, that newer publications tend to upgrade dynamic typed languages in contrast to older publications, that evaluate static typing having a positive impact on software development.

Sun Labs used JavaScript as a "Real Programming Language" for developing a web programming environment with approximately ten thousand lines of code. In [MT07], they expressed their experiences for using JavaScript as a general purpose programming language. In summary, their result was, that JavaScript is a powerful and expressive language, suitable for developing real applications and even system-software. They suggest an incremental and evolutionary development approach for the effective use of JavaScript, which is more familiar to developers grown-up with dynamically typed languages, than those, who are more used to programming languages with a static type system. They also claimed, that JavaScript's permissiveness and error tolerance has a negative impact on application development and debugging.

3.2. Survey Design

Given the limited literature and research about the assessment of client- and server side JavaScript in organizations, an exploratory survey across multiple enterprises and industries has been performed. The aim is, to gain an overview about organization's use and assessment about JavaScript, Node.js and their future. In order to evaluate our own experiences and statements found in literature, we formulated five research hypotheses to evaluate these observations.

[Kl12] identified within a survey, that the time, needed to fix type errors within a software, is decreased when using a programming language with a dynamic type system. In addition they mentioned an often upcoming statement that tool support for statically typed languages is easier achievable. This led us to the first Hypothesis:

Hypothesis 1 *JavaScript developers are not satisfied with the current tool support*

When using JavaScript as a general purpose programming-language, [MT07] identified different syntactical issues like redundancy or accidental syntax errors, which has major consequences especially because of JavaScript's error tolerance. Another indicator for syntactical problems of JavaScript might be the current development of languages like CoffeeScript, that try to "expose the good parts of JavaScript in a simple way"[81] and introduce a new language which compiles to JavaScript. The study of [O'12] even listed CoffeeScript as the 19th popular language. Therefore, we formulated the second hypothesis.

Hypothesis 2 *Developers do not like the syntax of JavaScript*

The maintainability of software refers to the simplicity for modifying it. An important aspect of maintainability is the analyzability, which refers among others to the locating of errors. As different researchers describe (see Section 3.1), this is problematic especially for dynamically typed languages. Also the development environment is an important factor for maintainability[82]. The tool support greatly contributes to this, which is not optimal for dynamic languages as described above, which led us to the next hypothesis:

[81] cf. [Fi12]
[82] cf. [Sp06, p. 325, 451]

Hypothesis 3 *JavaScript code is hard to maintain*

As described in the introduction, JavaScript has a quite high popularity at the moment, shown by different studies and by the fact, that JavaScript is used for the majority of modern websites. As we expect, that this trend will continue, we claim the fourth hypothesis:

Hypothesis 4 *The general importance JavaScript will rise in future*

Node.js currently attracts a lot attention. Big companies like eBay[83], LinkedIn[84] or Yahoo[85] build tools with it. Microsoft writes a special language for "application-scale JavaScript" with Node.js support[86] and enriches its web development tool Webmatrix2 with enhanced Node.js functionality[87]. Additionally, Microsoft supports Node applications on their platform-as-a-service product Windows Azure[88]. This wide industry support made us stating the last hypothesis.

Hypothesis 5 *Node.js is suitable for enterprise applications*

To evaluate our hypotheses and to gain a broader knowledge of organizations view on JavaScript and Node.js, we compiled an online questionnaire. It was divided into three sections. The first asking questions about JavaScript, the second focusing on Node.js and the third for getting knowledge about the participant and its company. Additionally, we provided a few open text fields in each of these sections to collect general moods about these technologies. After designing the questionnaire, a pretest was conducted with two researchers and four employees of the PENTASYS AG, who were not involved in the preparation. The suggestions and feedback, which were produced within the pretest were included in the questionnaire and the final version was published as an online survey, available for 21 days. We have send invitations for participating in the survey to over 1000 industry partners of the chair for Software Engineering for Business Information Systems of the Technical University of Munich. Additionally, the survey invitation was published on Xing, LinkedIn, the Node.js mailing list, a Google+ circle with 242 people included, mostly interested in JavaScript and the web forums jswelt.de, webdeveloper.com, python-

[83] cf. [EB11]
[84] cf. [Ch12]
[85] cf. [FR11]
[86] cf. [Mi12]
[87] cf. [Mi13a]
[88] cf. [Mi13b]

forum.com and ruby-portal.de.
We received 123 answers in total, whereat 23 participants (∼19%) have not completed the full questionnaire and were excluded from the evaluation. 37% of the participants were working in Germany, 22% in the USA, 5% in the United Kingdom , 3% in Italy and Austria and 2% in Australia. 10% did not gave the country, where they are working from currently. Each of the rest of the participants (21%) was working in separate countries all over the world. Table 3.1 illustrates the distribution of industry sectors of the participants.

Industry Sector	**n**	**% of all**
IT Products and Services	32	32%
IT Consulting	13	13%
Telecommunications	6	6%
Finance	5	5%
Education	4	4%
Production and Manufacturing	3	3%
Automotive	2	2%
Transport / Logistics	2	2%
Games	2	2%
Measurement Devices	2	2%
Health	1	1%
Management Consulting	1	1%
Public Service	1	1%
Open Source	1	1%
Other	10	10%

Table 3.1.: Survey Design - Organizations by industry sector

IT Products and Services was the largest sector with 32%, followed by IT Consulting with 13% and Telecommunication with 6%. Within the "Other" there was, despite of others, IT security, e-commerce and science mentioned. Table 3.2 shows the distribution of job titles of the participants.

The majority of participants were Software Engineers, with an amount of 47%. The second largest group of job titles was built by IT Architects with 10%, followed by Enterprise Architects with 7%. Within the "Other", there was despite of others a Head of Application Security, a Head of Research & Development and a trainee. The Consultants, with an amount of 3% were advised to answer the questions related to a single, concrete project, they have worked in.
In addition, we asked the participants for the typical kind of projects within

Job Title	**n**	**% of all**
Software Engineer	47	47%
IT Architect	10	10%
Enterprise Architect	7	7%
CXO	5	5%
Project Manager	4	4%
IT Operations	3	3%
Consultant	3	3%
Business Architect	1	1%
Other	8	8%

Table 3.2.: Survey Design - Participants by job title

their organization, whereat multiple answers were allowed. The given answers are shown in Table 3.3.

Kind of project	**n**	**% of all**
Web Development	79	79%
Cloud Computing	35	35%
Desktop Applications	26	26%
Embedded	13	13%
Security Applications	10	10%
Mobile Applications	4	4%
Other	3	3%

Table 3.3.: Survey Design - Typical kind of projects within the organizations

The majority of projects were within the field of Web Development with an amount of 79%. 35% of the projects were related to Cloud Computing, followed by the group of Desktop Applications with 26%. 13% of the organizations typically had embedded projects and 10% build Security Applications. The smallest group of projects with 4% was related to Mobile Applications. 3% of the projects had another focus, where Monitoring, Backend processing and Simulation was mentioned. Among all participants, there were 5%, who had never used JavaScript and have no experience with it. These were excluded from the JavaScript questions.

3.3. Survey Results

This section presents the results of the survey and is divided in three parts. First we present the results about JavaScript in Section 3.3.1, followed by Node.js specific answers, presented in Section 3.3.2. Finally, the results about the questions related to the participant's companies are presented in Section 3.3.3.

3.3.1. Results about JavaScript

For getting an overview about the volume of JavaScript code, produced in an average project, the participants were asked, how much of the code in their projects is written in JavaScript in average, measured in lines of code. They were able to give the answers in ranges of ten percentages or to answer that all or none JavaScript was used. Figure 3.1 illustrates the results in form of a bar chart.

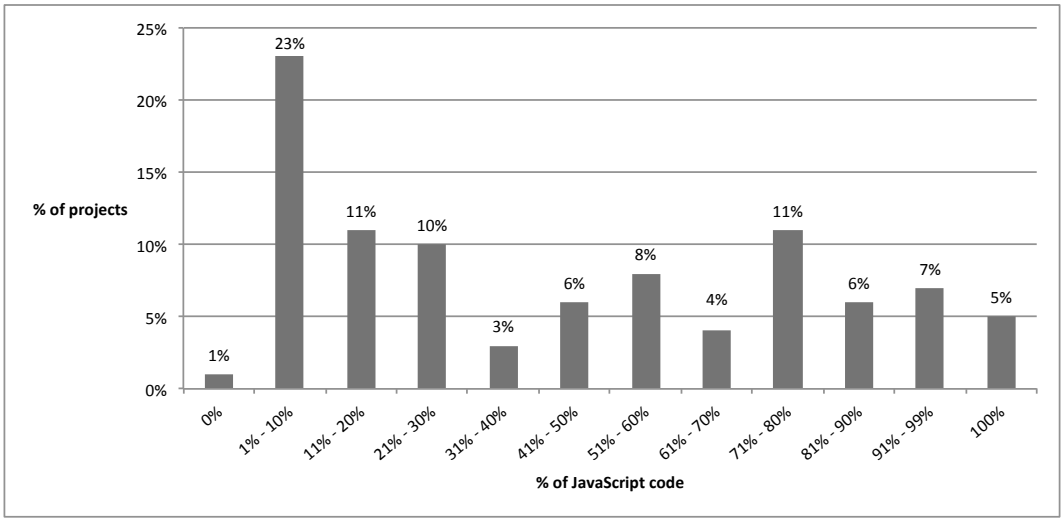


Figure 3.1.: Survey Results - The average volume of code in JavaScript

There were only 1% of projects, where none JavaScript was used. The biggest group of projects had 1% to 10% of the code, written in JavaScript. Taking the arithmetic average of each group, there is an average percentage of JavaScript code within projects of ∼44%. On the very right there are the projects shown, which are programmed fully in JavaScript, with an amount of 5%.

Table 3.4 shows the editors, used for JavaScript programming, whereat multiple answers were allowed.

Editor	**n**	**% of all**
Sublime Text	38	38%
Eclipse	30	30%
Vim	28	28%
Notepad++	14	14%
Visual Studio	14	14%
TextMate	9	9%
Aptana	8	8%
NetBeans	8	8%
Emacs	7	7%
IntelliJ	7	7%
WebStorm	7	7%
Komodo	5	5%
Other	14	14%

Table 3.4.: Survey Results - The usage of editors for JavaScript

The most popular editor for all participants was Sublime Text, with a usage of 38%, followed by Eclipse (30%) and Vim (28%). The Cloud9 IDE was mentioned two times and is included in the 14% of "Other". For analyzing Hypothesis 1 (JavaScript developers are not satisfied with the current tool support), we asked, whether the participants are fully satisfied with their editors. About 53.7% of them agreed, so it seems, that the hypothesis cannot be confirmed. In contrast, there were only ∼30.5% who were not fully satisfied, whereat ∼15.8% gave no answer to this question. All those participants, who were not fully satisfied were asked to give the features they are missing in their editors. In total, there were 25 answers. Most of them criticized missing code completion support, debugging and refactoring capabilities. It was also stated, that the syntax highlighting is not sufficient and that there is a need, for better code navigation functionalities.

For comparing the satisfaction between the different editors, we calculated the coefficient between the count of unsatisfied and satisfied answers for each editor, i.e. a higher value means, that there are more people satisfied with an editor than unsatisfied. A value lower than one means, that there were more unsatisfied people than satisfied ones. The highest values had IntlliJ(5), WebStorm (5), Komodo(4) and Emacs(4). The lowest values were calculated for Eclipse (∼0.85), Notepad++(∼0.83) and NetBeans(0.75). WebStorm is a

special IDE for JavaScript[89] and IntelliJ offers special Web Development capabilities for JavaScript and CoffeeScript[90], which applies for Komodo too[91]. This might was a reason for the high satisfaction with these editors. Emacs is an extensible and customizable text editor[92]. Developers using Emacs maybe adapt it for their needs and extend it with special JavaScript functionalities, which may lead to this high satisfaction. Eclipse has a quite low satisfaction, in fact, there are more unsatisfied people than satisfied ones. The reason for this might be, that people programming in Java with Eclipse use it for JavaScript, too. They maybe expect all the features they have for Java also for JavaScript in Eclipse's standard configuration, which is normally not the case. The same might apply for NetBeans.

For getting an overview about the impressions of JavaScript we provided nine statements, where the participants were asked to judge on a Likert scale. Figure 3.2 shows the judgement about the first statement related to JavaScript's syntax.

24.2% strongly agreed to "I like the syntax of JavaScript" and 41.1% agreed which makes in sum an approval of the statement of 65.3%. Therefore, Hypothesis 2 could not be confirmed. In contrast, there were only ∼30%, who did not like the syntax. For analyzing the current development of other languages, compiling to JavaScript, like CoffeeScript or Typescript, we asked some questions about them. CoffeeScript seems to be the more popular language. 88.4% of the participants, have already heard about it, where Typescript was only known by 54.7%. Although, CoffeeScript is widely known, it was not used a lot within the projects of the participants. ∼80% of the projects does not use it at all, ∼8% use it for less than the half but at least for 1% of their JavaScript code and ∼5% for more than the half but less than for all. There were about 7% of the projects using CoffeeScript for all their JavaScript code. In addition, the participants were asked to give some reasons for using CoffeeScript. In the eyes of the participants, the major advantage of using it was a better syntax, but being almost one to one equivalent to JavaScript at the same time. It was mentioned, too, that it provides a better object-oriented model than JavaScript and is more like Ruby or Python, which makes it easier to learn

[89] cf. [Je13b]
[90] cf. [Je13a]
[91] cf. [Ac13]
[92] cf. [Fr13]

for programmers, having experiences with these or other object oriented languages. It was mentioned additionally, that it makes the code more readable, easier to maintain, shorter and more expressive for some statements. While we received 14 answers, describing the reasons for using CoffeeScript, there were 58 reasons given for not using CoffeeScript. A lot of the participants just read about it or did not have the time to test it yet. The biggest statement against CoffeeScript was, that it adds another layer in between. This leads to a fragmentation of the community, inconsistent coding styles, less tool support, adds another build step and makes it harder to find good developers. It was mentioned by eleven participants, that they prefer the JavaScript syntax as it is and that they do not see any advantage of using CoffeeScript. It was also stated, that it is just useless. Some said that it is easier to write, but harder to read than JavaScript.
Another more often occurred opinion was, that it is harder to debug, and that it is important to debug in the same language as you write the code. It was also stated, that it is too much effort to learn, too new or that it just does not fit into the project. Two participants did not like the concept of code generation and stated, that CoffeeScript is obscure. The participants, not using CoffeeScript at the moment, were asked whether they are planning to use it in future. Three of them (∼4.5%) are planning to do so, in contrast there were 50 participants (∼74.6%) who do not, whereas the other 20.9% gave no answer.
As already mentioned, TypeScript was less known than CoffeeScript, even though ∼54.7% of the participants already heard of it. There was one person, using TypeScript for 1% to 49% of its code. It was stated, that it is "easier for complex projects". 42 reasons were given why TypeScript was not used yet. A majority said, that there was no time, or that they just played around with it a bit. Another widely stated argument was, that it is too new or not stable enough. It was mentioned a lot, that there is no need for types in JavaScript, in contrast, the dynamic typing is one of its strengths that would be eliminated by TypeScript. Six people did not like the fact, that it is developed by Microsoft and others complained, that there is no IDE supported despite of VisualStudio, which is only available for Windows. Additionally, there were similar statements as about CoffeeScript: There is no advantage or no need, it adds another layer and build step and that pure JavaScript is better respectively it is important to know JavaScript well. In contrast to CoffeeScript, there were six people, who are planning to use TypeScript in future and 33

who does not. The reasons for using it in future were mostly a positive impact of types especially for documenting and IDE support. One participant mentioned, that he will use it only, if there will be a good IDE support for Linux and Mac operating systems, too.

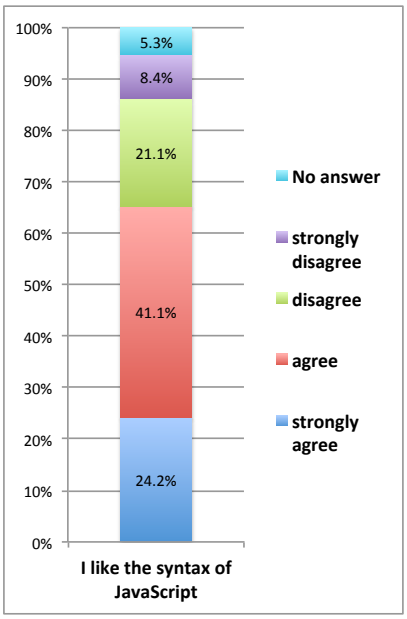


Figure 3.2.: Survey Results - Judgements about JavaScript Syntax

In sum, both, CoffeeScript and TypeScript are already well-known. Even though, they are not used a lot as the participants just like JavaScript itself or they think that it is better to write it as it is, without adding a new build step or layer in between. This also indicates, that Hypothesis 2 was not correct. Additionally, it was a lot mentioned, that there is a need of improvement for the tool support for both, CoffeeScript and TypeScript.

For analyzing Hypothesis 3, we made a closer look to the maintainability of JavaScript code. Figure 3.3 shows another four judgements of statements about JavaScript related to maintenance in a pillar chart.

The first statement was directly related to maintenance and the judgement is shown at the very left within the figure. 8.4% of the participants strongly and 43.3% agreed, that JavaScript code is hard to maintain which makes in sum an agreement of over 50%. Only 44.2% disagreed on this, which gives a first indicator for Hypothesis 3 being wrong. To go deeper in the maintenance topic, we did further statements where the participants were asked to judge about. As mentioned in Section 3.2, [Sp06] stated, that the four attributes of maintainability of the ISO software engineering product quality standard corresponds to four phases of a maintenance change. These are analyzability, changeability, stability and testability. Analyzability refers to locating the cause of an error and locating the parts of a software, that has to be modified for implementing new features. The easiness of these tasks highly refer to readability of code, which we tried to analyze for JavaScript with the judgement about the statement "JavaScript code is easy to read". The result is presented in the pillar second

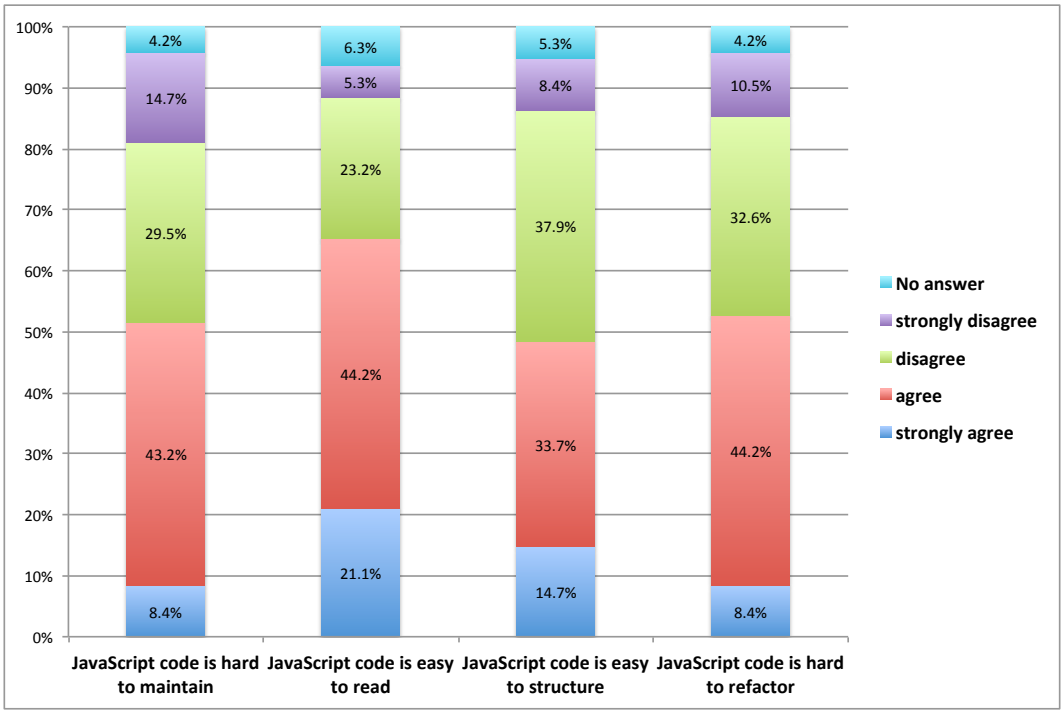


Figure 3.3.: Survey Results - Statements about JavaScript related to maintenance

from left within Figure 3.3. 65.3% of the participants agreed to this including 21.1%, who even strongly agreed. On the other hand, there were 28.5%, who disagreed, including 5.3% who strongly disagreed. An important part of analyzability that also influences the readability is the visual structure of code. 14.7% of the participants strongly agreed and 33.7% agreed to the statement "JavaScript code is easy to structure". In sum, there is an approval of this statement of 48.4% of the participants, while there are 46.3% of the participants, who disagree on this. Even with weighting the strong statements more, there is no distinct opinion about the easiness of structuring JavaScript code. Nevertheless, the agreement prevails slightly so, there is a positive influence on readability, which has a positive impact on JavaScript's maintainability.

Despite analyzability the changeability influences the maintainability, too. Changeability refers to the easiness of implementing specified modifications[93]. For analyzing this, we stated "JavaScript code is hard to refactor". The result is not distinct, too, although about 52.6% of the participants agreed, whereas 42.6% disagreed, so, the agreement prevails slightly which positively influences the maintainability of JavaScript.

[93] cf. [Sp06, p. 403]

For improving the software quality, it is important to have tools and frameworks, supporting the testing, which has an impact on maintainability too. The survey showed, that ∼42.1% were using testing frameworks for their JavaScript code and ∼46.3% did not. The most popular testing frameworks are listed in Table 3.5 where the participants were asked to name the frameworks they are using. In total, there were 69 answers given.

Framework	**n**	**% of the used frameworks**
Jasmine	15	∼21.7%
Mocha	13	∼18.8%
QUnit	7	∼10.1%
Vows	5	∼7.2%
Nodeunit	3	∼4.3%
JSUnit	3	∼4.3%
Sinon.JS	3	∼4.3%
Phantom.JS	3	∼4.3%
Expect.js	2	∼2.9%
YUI Test	2	∼2.9%
Chai	2	∼2.9%
Other	11	∼15.9%

Table 3.5.: Survey Results - JavaScript testing frameworks

Jasmine was used by ∼21.7% of the participants, using a testing framework and builds the top of the usage of testing frameworks, followed by Mocha (∼18.8%) and QUnit (∼10.1%).

After all, Hypothesis 3 seems to be not correct. It was shown, that the majority of the participants were satisfied with their tool support. They also liked the syntax of JavaScript and also agreed to a good readability and in general to a good maintainability of it. This is supported by a wide usage of testing frameworks for JavaScript.

To show whether there is a correlation between the application of a testing framework and judgments about the maintainability of JavaScript we applied a Pearson's χ^2 test on those values. The calculated value of $p = 0.008$ confirms a statistically high significance between these two values, i.e. the usage of a JavaScript testing framework positively influences the maintainability.

Figure 3.4 shows the remaining three statements related to the general impression about JavaScript. Participants, who not gave an answer were excluded

from this figure to make it more readable. This makes the pillars not comparable to each other, as they are based on a different underlying set.

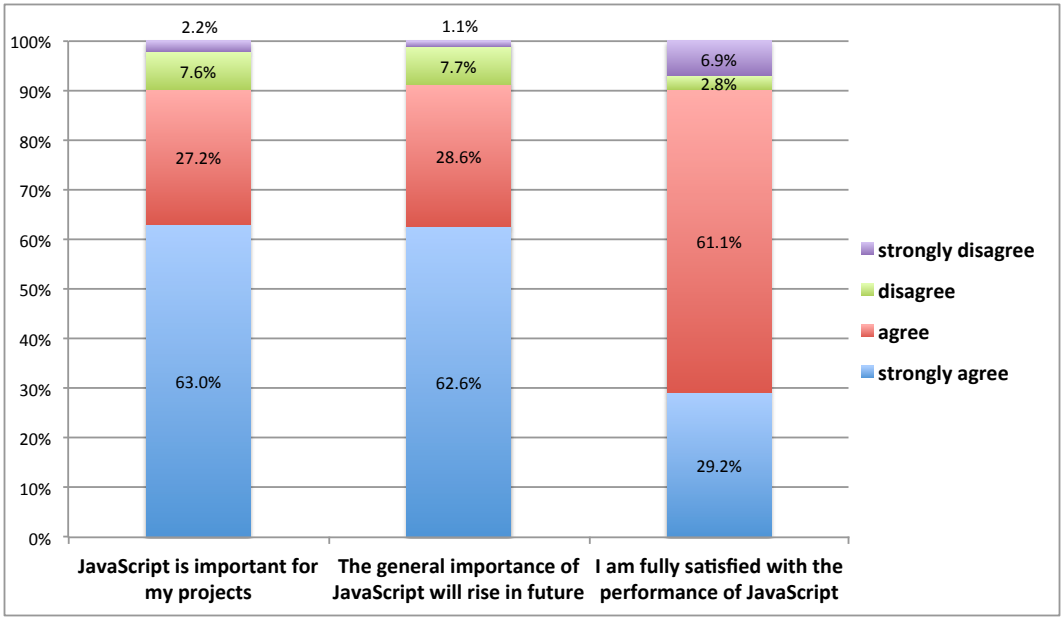


Figure 3.4.: Survey Results - General statements about JavaScript

As it is shown at the very left in Figure 3.4, JavaScript is important for the majority of the participants. About 90% agreed to this in total whereas 63% even strongly agreed to this statement. The picture is almost the same for the future importance of JavaScript, where also about 90% agreed, that the general importance of it will rise whereas 62.6% of the participants even strongly agreed to this. This confirms our Hypothesis 4 (The general importance JavaScript will rise in future). The third pillar shows, that the majority is satisfied with the performance of JavaScript, which more relates to the respective interpreters.

After all, we gave the participants the possibility, to give some general opinions and statements about JavaScript. Some selected cites out of the 16 statements can be found in Section A.2.

3.3.2. Results about Node.js

Node.js was widely known by the participants. 88% of them have already heard about it, mostly via internet news (∼54.6%), whereas "Hackernews" was mentioned explicitly a few times. Despite of that, twelve people heard about it the first time over friends and ten in business.

About 68.2% of the participants, which is an amount of 60 people, actively develop with Node.js or it is used in one of their projects. Our next question asked the context of the projects, whereas multiple answers were allowed. Table 3.6 lists the respective results.

Project context	**n**	**% of projects**
Private	45	45%
Business	43	43%
Open Source Project	21	21%
University/Research	2	2%

Table 3.6.: Survey Results - Context of the Node.js projects

45% of the projects were done privately and 43% for business. 21% of the projects were open-source projects and 2% of the projects were related to research. As only 60 people answered the question, the number of projects (111) was quite high. In average every participant had almost two projects in parallel. Next we asked for the number of people, working in these projects. The result is illustrated within Table 3.7.

# of projects	**% of projects**	**# of people**
22	41.5%	1
12	22.6%	3
6	11.3%	2
4	7.5%	5
3	5.7%	4
3	5.7%	10
1	1.9%	8
1	1.9%	15
1	1.9%	80

Table 3.7.: Survey Results - Number of people within projects

On the majority of projects, there was only one person working. 22.6% had three people working on so it seems, that the projects tend to be smaller. The biggest project had 80 employees, so there was a quite big project, too, that uses Node.js. In average there were 14.2 people working on one project.
Subsequently, the participants were asked to give the starting date of their projects. Figure 3.5 shows the number of projects started per month of year.

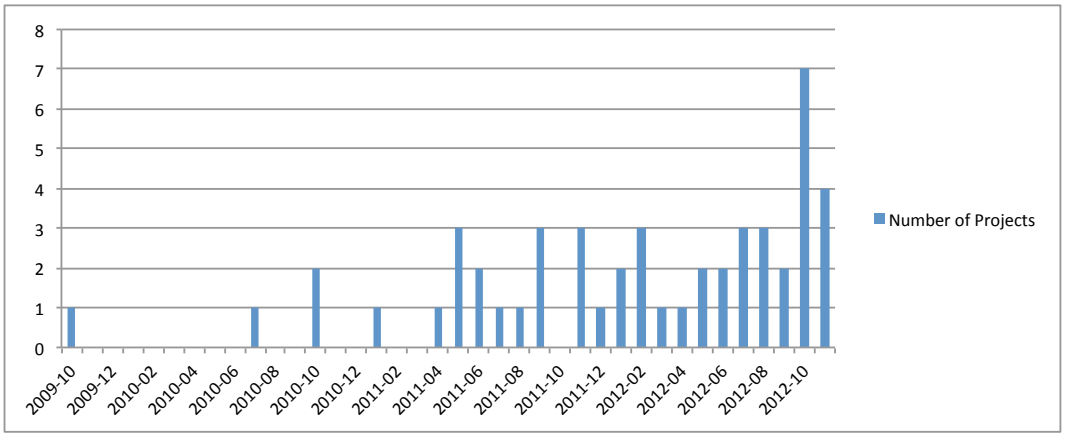


Figure 3.5.: Survey Results - Beginning of the projects

The oldest project already started in October 2009. Node.js came out a bit later as it was presented first in November 2009 at JSConf EU[94]. Therefore, this date might be wrong. Nevertheless, the majority of projects was quite young. 50% of them started after March 2012. Measured to the 31.12.2012 the average project age is about 349 days. At next, we were interested in the kind of projects the participants are developing. Table 3.8 shows the respective kinds with their amount.

Kind of Project	**n**	**# of projects**
Web Application	52	~61.9%
Command Line Tool	11	~13.1%
Framework	10	~11.9%
Server	3	~3.6%
Embedded	2	~2.4%
Browser Game	1	~1.2%
Desktop Application	1	~1.2%
Telephony IVR	1	~1.2%
Other	3	~3.6%

Table 3.8.: Survey Results - Kind of projects

It turned out, that the majority of projects are Web Applications with an amount of about 61.9%. As the gap to the subsequent kinds is quite distinct, Web Applications seem to be the common use case for Node.js. Despite of that, there is a variety of other project kinds. About 13.1% of the projects were Command Line Tools and about 11.9% of the participants projects build

[94] cf. [Da11]

frameworks. There were also ∼3.6% who build servers and ∼2.4% who worked in the area of embedded devices. One participant also built a Desktop Application and there was one, building a telephony interactive voice response with Node.js. 38 participants also provided a description of their projects. Most of them were related to web applications focussing on prototyping, games, conferencing, network analysis, monitoring and realtime functionalities. One project helps to work with asynchronous code by providing a flow control library and another one controls remotely the arduino platform.

For analyzing the reasons for Node.js being used, we asked for the reasons for choosing Node.js for these projects, where we received answers from 49 participants. The biggest reason for choosing Node is performance and simplicity. A lot of the participants felt already comfortable with JavaScript and therefore used it also on the server side. It was often stated, that this was a good fit for web applications, as the client was programmed in JavaScript and the databases were using it, too. It was also mentioned, that it is good, if there is no "phase-shift" when switching from programming the client in JavaScript to the server side. In this context it was also said, that there is a possibility of code reusability on the client and the server side. Big reasons for choosing Node were also its capabilities related to realtime web applications. A few people liked its evented and non-blocking nature and evidenced Node a good scalability and concurrency behavior. The low time to market of Node seems to attract especially startups, too. It was also mentioned, that node was chosen just out of curiosity, because it is open-source, its fun and it makes low-level HTTP usage easy. The easy integration of C/C++ libraries, the strong community, the handling of high volume of requests and the easiness to maintain was also given as reasons for choosing Node.

About 35.7% of the participants not using Node, considered to use it for one of their future projects but decided against because of several reasons. One was, that it is hard to find experienced staff and that there is lack of enterprise experience and enterprise support, which is analyzed in the following, too. One participant mentioned a personal risk when choosing Node ("You won't get fired for choosing J2EE but NodeJS"). The others found better fitting technologies, haven't got the time to learn it or found too many bugs. About 64.3% of the people not using Node did not consider to use it at all. For this we received nine reasons, whereas the biggest was, that there is no need for a new technology or that there is no customer requirement. Four participants did not

like JavaScript and two mentioned, that they felt comfortable with Python and had no need for using Node. 25% of the participants, who currently did not use it, were planning to use it in future, whereas about 64.3% did not.

For getting an overview of used frameworks on top of Node and to achieve a tendency of typical use cases for it, we asked which frameworks the participants use. About 18.3% of the participants used plain Node.js and ∼81.7% made use of frameworks, which are shown in Table 3.9.

Framework	**n**	**# of participants**
Express	40	40%
socket.io	5	5%
connect	4	4%
flatiron	2	2%
Other	17	17%

Table 3.9.: Survey Results - Usage of frameworks on top of Node.js

The high usage of express, a web application framework for node[95] (40% of the projects), connect, a middleware framework[96] (4% of the projects) and flatiron, a web application framework, too[97] (2% of the projects) confirms the typical use case of web applications for Node. The realtime capabilities of Node were mentioned a lot within the questionnaire, which is also shown by socket.io being used by 5% of the projects, which is a framework for "realtime apps" for Node[98]. Despite of them, there were 17 other frameworks mentioned, but participants stated, that they use a lot of more frameworks. These other frameworks include despite of others web application frameworks like Derby or Meteor, realtime frameworks like SocketStream or nowJS and object modeling frameworks like mongoose.

For getting an assessment about Node.js and its future, we provided three statements, where the participants were asked to judge on a Likert scale. The results of the judgements are illustrated in Figure 3.6.

As it can be seen, Node.js was very important for the projects of the participants. 47.7% of the them strongly agreed to the provided statement related

[95] cf. [Ho12]
[96] cf. [Se12]
[97] cf. [No12]
[98] cf. [Ra12b]

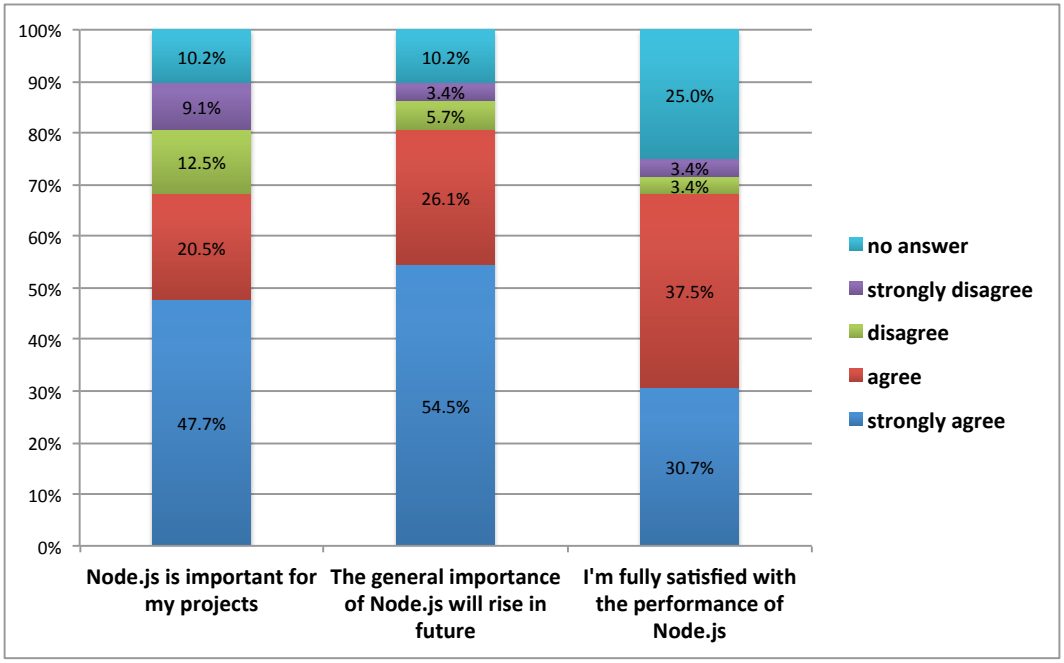


Figure 3.6.: Survey Results - Statements about Node.js

to the project importance and 20.5% agreed. From the participants point of view, the general importance of Node.js will rise in future. More than the half of them strongly agreed to this, which is a quite distinct result. Performance was one of the main reasons for choosing it for the projects. In the next question we explicitly asked to judge this. One fourth did not give an answer here, so, there were many people not sure what to judge. For the participants, who gave an answer, the result is quite clear. Here only 6.8% of them disagreed to the statement "I'm fully satisfied with the performance of Node.js", the rest agreed to this.

In order to analyze our Hypothesis 5, "Node.js is suitable for enterprise applications", we asked the participants about their opinion about its enterprise readiness. About 64.8% of them agreed to our hypothesis. The most occurring reasons for the enterprise readiness of Node.js was its scalability, performance, stability and a short time to market respectively a shorter development time. The participants emphasized its simplicity and mentioned a lot, that it is positive to have one single language for both, the client and the server side, which makes developers able to switch and also simplifies the finding of new developers. A few participants also said, that they do not see any reason, why it should not be enterprise ready. From the participants point of view, Node.js had a good and large community, strong core developers and provides a lot of

libraries and frameworks accessible over the node package manager. It was also reasoned, that Node is responsive, open, extensively tested, efficient, clean and that its easy to build modules and easy deployable in the cloud, which makes it enterprise ready. It was mentioned, too, that enterprise readiness is more related to a good development process, than to the language itself. Another participant referred to LinkedIn as an enterprise, already using Node.js productively. 13.6% of the participants stated, that Node.js is not enterprise ready. Reasons for this, given by the participants were missing stability and a lack of enterprise experience and support. One participant mentioned, that it is hard to find experienced staff for Node.js. The maintainability was criticized, too. Despite of that, it was stated, that it is hard for libraries to keep up-to-date to the quick evolution of Node and that there is no good way to handle multiple Node.js instances. Another reason was, that legacy code is mostly written in other language, which makes Node.js not to a good fit. Additionally, one participant mentioned, that there is no connectivity to enterprise databases. Despite of these it was claimed, that JavaScript is not safe and that open-source libraries have proven to be unstable. It was criticized, too, that evented IO complicated everything.
This brings us to the next question, which asks, what functionalities the participants are missing currently. It was mentioned that there is a lack of threads and that the usage of multiple cores of a CPU is not supported well. Additionally, one participant missed communication features for distributed Node.js processes. Database related features were mentioned three times. The participants wished an ORM like Active Record, a graph database engine, a generic database connection and a native HBase client to be implemented. It was also stated, that authentication and authorization features are missing at the moment. In addition there was a SSH and an incoming mail server implementation, a higher level file system API, OpenCI support and a reference content management system implementation stated. At last, the participants were not satisfied with the current debugging functionalities and wishes, to have a graphical debugger included in one major IDE. Furthermore, they claimed a better behavior driven development style testing framework.

At last we gave the participants the possibility, to state general comments about Node.js. Some selected ones can be found in Section A.3.

3.3.3. Background Information about the Companies

Within the third part of the questionnaire, we asked a few questions about the companies, the participants are working in. Some of the results were already presented in Section 3.2, whereas the remaining ones are presented here.
Within this third part, we asked the participants how many employees their companies have, where we received 73 answers in total. It turned out, that there was an average number of employees of 2495. In this context, we analyzed, which participants actively develop with Node.js or where it is used in one of their project in a business context. The number of companies, where this applied was 32. Calculating the average number of employees here gives an amount of ∼1100, so companies using Node.js tend to be smaller.
A similar tendency can be seen with the year of foundation of the companies. The average year from 68 answers was ∼1986. Taking only the companies, that use Node.js, the average year of foundation is ∼2004, therefore it seems, that younger companies use Node.js rather than older ones.
At last we asked, if the participants think, that the company they are working in is an early adopter of new technologies. 32% agreed to this, 26% even strongly agreed, 26% disagreed and 5% strongly disagreed. So, the majority of people, taking part in the survey are working in companies that are early adopters. Taking again the companies, that use Node.js, there were ∼34.2% of the participant who would strongly agree to the statement, that their company is an early adopter, 43.0% agree, whereas there are ∼19.5% who disagree and ∼2.4% who strongly disagree to this statement. It turned out, that companies using Node.js tend to be early adopters of new technologies.

3.4. Survey Conclusion

The survey showed, that JavaScript is used a lot in general. More than the half of the participants were satisfied with its current tool support, albeit, there are some features missing, like debugging, code completion or code navigation. Most of the participants liked the syntax of JavaScript and do not use the newly developed languages TypeScript or CoffeeScript. It was shown, that the majority of the participants is satisfied with JavaScript's maintainability.
About 42% of the participants are using testing frameworks for their JavaScript code. It turned out, that there is a statistically high significance between

the application of testing frameworks and the judgement about JavaScript's maintainability i.e. the usage of a JavaScript testing framework positively influences it.
In general, JavaScript has a very high importance for the participant's projects and it was shown, that its importance will rise in future.

Node,js is already widely known. Projects, that makes use of it, tend to be smaller and younger. The typical use case for it are web applications, which was shown by the participant's projects and the usage of typical web application frameworks on top of Node.js. More than the half of the participants said, that Node.js' importance will rise in future. Furthermore, a majority confirmed the enterprise readiness of it.

4. Design of the Collaborative Editor

This chapter introduces the design of a collaborative editor that enables different forms of collaboration and realtime capabilities. To do so, it follows a bottom-up approach, starting with a high-level consideration of the system followed by describing each part in detail. Chapter 5 continues by describing a prototypical implementation of the presented design.
Section 4.1 shows the use cases, that are realized by the editor. Section 4.2 gives a brief introduction to the realtime architecture, the collaborative editor follows. These two sections build the basis for an architectural overview, presented in Section 4.3, which is followed by the description of the client-design in Section 4.4 and the server-design in Section 4.5.

4.1. Use Cases

This section presents the use cases for the collaborative editor prototype. They are grouped by their central topic into user centric use cases (Section 4.1.1), document centric use cases (Section 4.1.2) and dashboard centric ones (Section 4.1.3). This grouping supports the understanding of the respective contexts. Figure 4.1 illustrates all of them within a use case diagram. The subsequent sections present all use cases with a detailed description.

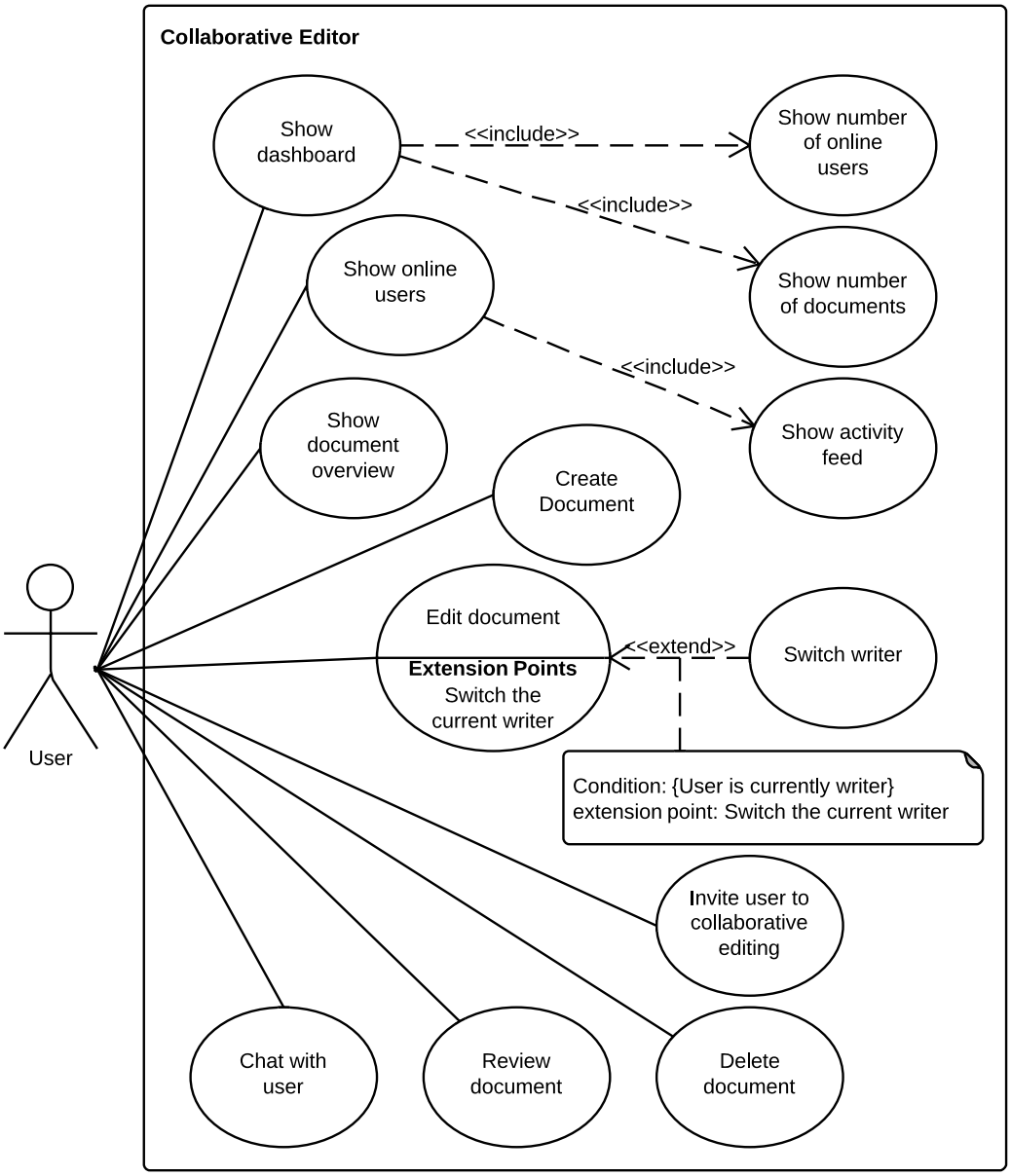


Figure 4.1.: Use Cases of the collaborative editor

4.1.1. User-centric use cases

4.1.1.1. Show online users

Description Commonly, there are multiple users active on the collaborative editor. For getting an overview, the user interface provides a list of all users who are currently online.

Pre-condition The user is logged in.

Basic Flow of Events

1. The user logs in
2. The user goes to the collaborative editor start page
3. The user sees the online collaborators

Post-condition All users, who are currently online, are shown in the user interface.

4.1.1.2. Invite user to collaborative editing

Description This use case enables the user to invite other users to work collaboratively on a document.

Pre-condition The user is logged in; there is at least one other user online within the collaborative editor; there is at least one document.

Basic Flow of Events

1. The user logs in
2. The user goes to the collaborative editor start page
3. The user sees a list of all online collaborators
4. The user clicks on one of these users to open the context-menu
5. The user chooses "Invite to collaborative work"
6. A list of all available documents is presented to the user
7. The user selects one of these documents and clicks on the "Invite and open" button
8. The user gets automatically forwarded to the chosen document
9. The chosen collaborator receives a chat message that she was invited to work collaboratively on a document. The chat message contains a link to the document.

Post-condition The inviting user sent the invitation and looks at the chosen document. The collaborator received the invitation in form of a chat message including a link to the document.

4.1.1.3. Chat with user

Description One important collaboration feature of the editor is the chat functionality, which enables users, to exchange text messages

Pre-condition The user is logged in; there is at least one other user within the collaborative editor.

Basic Flow of Events

1. The user logs in
2. The user goes to the collaborative editor start page
3. The user sees a list of all online collaborators
4. The user clicks on one of these users to open the context-menu
5. The user chooses "Talk to"
6. A chat window gets opened automatically
7. The user types a new message and sends it

Post-condition The chat-partner was notified about the new message and is able to answer

4.1.1.4. Show activity feed

Description The use case "Show online users" already displays all users, who are currently online. This use case additionally provides an activity feed for each user indicating her current activity e.g. "Is currently editing Document 5".

Pre-condition The user is logged in; there is at least one other user online.

Basic Flow of Events

1. The user logs in
2. The user goes to the collaborative editor start page
3. The user can see the current activity of the other users

Post-condition The activity feed shows the current activity of another user

4.1.2. Document-centric use cases

4.1.2.1. Show document overview

Description This use case presents all the currently available documents, including their names, the date of the last modification and the actions, that can be executed on this document like review, edit or delete.

Pre-condition The user is logged in.

Basic Flow of Events

1. The user logs in
2. The user goes to the collaborative editor start page
3. The user chooses the document overview within the user interface
4. All the documents are presented

Post-condition All currently available documents are presented to the user.

4.1.2.2. Create document

Description This use case enables the user to create new documents.

Pre-condition The user is logged in; the use case "Show document overview" was executed.

Basic Flow of Events

1. The user clicks on the the respective button for adding a new document within the document overview screen
2. The user types in the name for the new document and optionally adds some tags to it
3. The user confirms the creation

Post-condition The new document was created successfully; the document is automatically visible for all other online users.

4.1.2.3. Edit document

Description This use case enables the user to edit an existing document. There is always one user able to actively write on the document at the same time. All subsequent users, opening the same document respectively executing this use case, are automatically set as reader.

Pre-condition The user is logged in; the use case "Show document overview" was executed; the document was created.

Basic Flow of Events

1. The user clicks on the the respective button for editing an existing document within the document overview screen
2. The document gets opened and is ready for collaborative editing

Post-condition Document was opened and is ready for collaborative editing.

4.1.2.4. Switch writer

Description There is always one user able to actively write on a document at the same time. All other users are automatically set as readers. This use case enables the current writer, to hand on this role to another user.

Pre-condition The user is logged in; the use case "Edit document" was executed; the user is currently the writer.

Basic Flow of Events

1. The user clicks on the the drop down menu "Select other writer" to choose another user who has currently the same document opened
2. The user clicks on the "Go"- button
3. The user is set automatically as reader, whereas the chosen user is automatically set as writer

Post-condition The one user, initiating the switch is now set as reader. The chosen user is now set as writer.

4.1.2.5. Review document

Description This use case enables the user to see the changes which were done since the creation of the document per user.

Pre-condition The user is logged in; the use case "Show document overview" was executed, the document was created.

Basic Flow of Events

1. The user clicks on the the respective button for reviewing an existing document
2. The user gets the first state of the document presented, just after its creation
3. The user interface offers the possibility to walk through all changes done by the different users until the current state of the document

Post-condition -

4.1.2.6. Delete document

Description This use case enables the user to delete existing documents.

Pre-condition The user is logged in; the use case "Show document overview" was executed; the document was created.

Basic Flow of Events

1. The user clicks on the the respective button for deleting an existing document
2. The user can confirm or abort the action

Post-condition The chosen document was deleted successfully. The removal of the document is automatically propagated to all other users.

4.1.3. Dashboard-centric use cases

4.1.3.1. Show dashboard

Description The dashboard presents an overview of important figures including the number of currently available documents and the number of online users. Theses figures are automatically updated, if they change.

Pre-condition The user is logged in.

Basic Flow of Events

1. The user logs in
2. The user goes to the collaborative editor start page
3. The dashboard is shown

Post-condition All elements within the dashboard are presented to the user.

4.1.3.2. Show number of documents

Description This number provides an overview about the amount of documents available within the editor.

Pre-condition The user is logged in; the use case "Show dashboard" was executed.

Basic Flow of Events

1. The dashboard shows the number of documents

Post-condition The number of documents is presented within the dashboard.

4.1.3.3. Show number of online users

Description Commonly, there are multiple users active on the collaborative editor at the same time. For getting an overview, the number of currently online users is presented to the user.

Pre-condition The user is logged in; the use case "Show dashboard" was executed.

Basic Flow of Events

1. The dashboard shows the number of currently online users

Post-condition The number of currently online users is presented within the dashboard

4.2. The Realtime Architecture

In [Ma11], Alex MacCaw describes an architecture for realizing realtime web applications. As a pre requirement, the application's client side should be modeled following the Model-View-Controller-pattern that divides it into three parts. The model only holds data. It does not know anything about the controller or the views, it just keeps the pure data. The view is the part, the user directly interacts with. It should also be decoupled from the controller and the model and it has no business logic inside. It is just responsible for presenting data and enabling user interaction. The last part of this architectural pattern is the controller. It holds the business logic and can be seen as the connector between models and views. If a user acts with the view and clicks on a button an event is fired that is catched by the controller that does the logic that needs to be done for processing this event properly.
This brings us the the next important characteristic of a realtime architecture: it is event-driven. Typically, this means that it is driven by user interactions which is in our context, for example, the creation of a new document or just the appearance of a new user. All these actions result in events that may need to be propagated through the system and sent to the other clients. [Ma11] states two important things to think of during the creation of a realtime application. The first one is to decide which models need to be realtime which leads to the decision which data need to be propagated trough the system. The second one is to decide, who needs to be notified about these changes. This is mostly realized with the publish-subscribe pattern which is also used within the collaborative editor in a special way. Here, a client subscribes to different channels that enables the server to notify groups of clients respectively these ones, that subscribed to a single channel. If only one user is present in a channel, only the one user gets notified, which can be seen as direct message from the server to a single client. If multiple users subscribed to one single channel, a server is able to send messages to a group of clients. Both methods are used

within the collaborative editor.
This brief introduction to the realtime architecture will lead the whole design and development of the collaborative editor, presented in the following.

4.3. Architectural Overview

A major requirement for the collaborative editor prototype is the integration into Tricia, that was introduced within Section 2.5. Given additionally the design goal of a realtime architecture, presented in Section 4.2 the architecture presented in Figure 4.2 was developed.

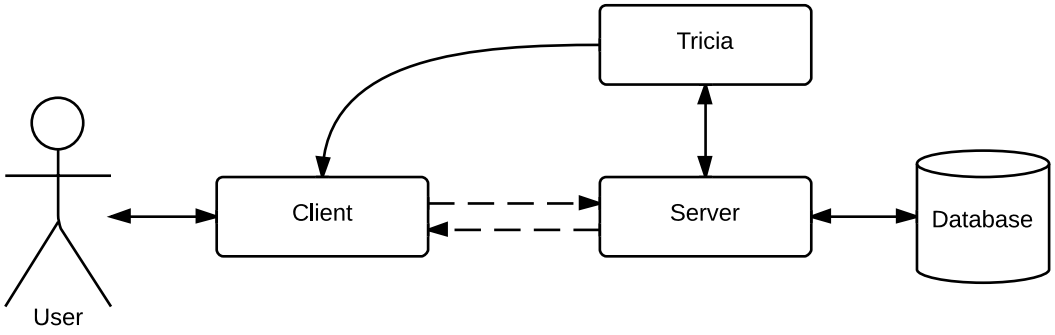


Figure 4.2.: Fundamental architecture of the collaborative editor

At the top, there is Tricia shown, where the collaborative editor is integrated. This integration offers the possibilities that there is no user authentication or authorization management needed within the prototype as this can be realized by the access control framework of Tricia as described in Section 2.5. For getting access to the collaborative editor, a user needs to login successfully into Tricia. Afterwards, Tricia delivers the whole client code via its HTML interface to the user who then connects herself to the server. In addition, Tricia delivers information about the logged in user to the client of the prototype and offers an interface that delivers the documents, stored within. These ones are collected by the collaborative editor and saved in an own database and to make them accessible for the connected clients. The separation between Tricia and a self-developed server leads to clear responsibilities. Tricia only delivers the client code, offers authorization and authentication functionalities and provides user information and the initial documents. The collaborative editor itself is realized within the client and the server independently. The

whole communication between those two parties is asynchronous to realize the event-driven nature of the system, described within Section 4.2.

4.4. Design of the Client

After the description of the realtime architecture and the architectural overview of the collaborative editor, this section introduces the client design in detail. To do so it first presents the general kind of the client in Section 4.4.1. Afterwards, the division into different components is introduced in Section 4.4.2, that also describes each component in detail.

4.4.1. Rich Internet Applications

The client of the collaborative editor is realized as a "Rich Internet Application" (RIA). RIAs follow the goal of "adding new capabilities to the conventional hypertext-based Web"[99] which leads to an improvement of the data, business logic, communication and presentation of web applications. The collaborative editor mostly runs on the client itself, which can result in quicker responses and optimized communication costs. Furthermore, there is no need for an additional usage of Tricia. The client code is delivered once whereas the following communication can be fully realized between the client and the self-written server. Typical for RIAs is also the possibility of bidirectional communication between client and server which means that both parties can initiate a communication at any time. This optimizes the communication in terms of latency[100] and is important for enabling the push messages from the server to the client, described in Section 4.2.

4.4.2. Client Components

For better structuring and for clear responsibilities, the client is divided in different components, which are shown in Figure 4.3.

[99] [FRSF10, p. 10]
[100] [FRSF10, p. 10]

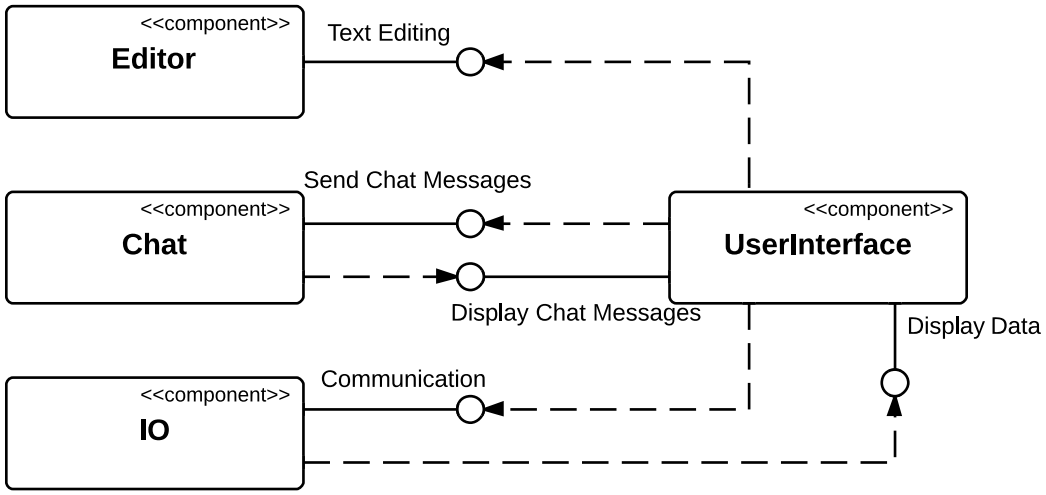


Figure 4.3.: Components of the client

The `UserInterface`-component is the central component of the client. It is the part the user directly interacts with and it manages the displaying of incoming data from either the `Chat`- or `IO`-component. Furthermore it catches events resulting from user interactions and sends them to these two components if they has to be propagated to the server or to other clients. The components `Editor`, `Chat` and `IO` directly communicate with the server. The `Editor`-component realizes the integration of an advanced text editor and manages the live-editing by applying operational transformation functionalities. The `Chat`-component holds a connection to the chat component of the server and passes new messages from the server to the client and the other way around. The fourth component `IO` manages despite of other things the data exchange of documents or user information.
The following sections present each component in detail.

4.4.2.1. User Interface Component

Wireframe

The design of the user interface of the collaborative editor is not a major part of this Thesis although, being familiar with it supports the comprehension of the following sections. To do so, Figure 4.4 presents a mock-up of the user interface for getting a first idea on how it looks like.

It consists of three main parts: a header, a navigation bar at the left and an area at the right for displaying content. The header shows the logo of the

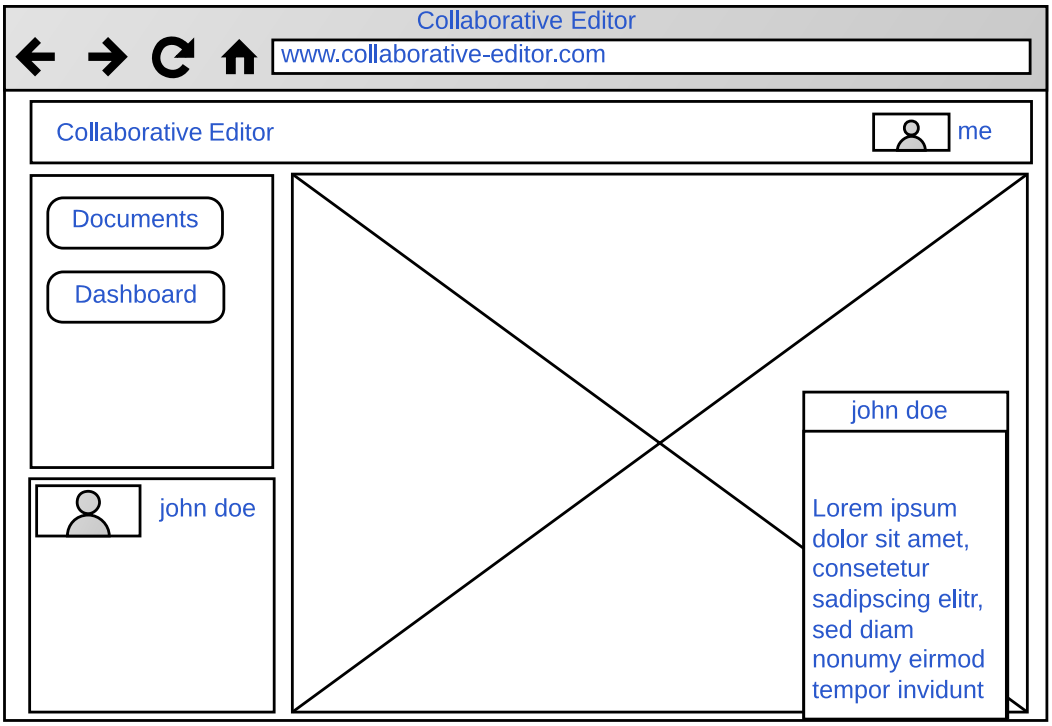


Figure 4.4.: Static Wireframe of the User Interface

collaborative editor and the name of the user who is currently logged in with her browser. This is a static area where no changes are done during the whole lifecycle of the client. The navigation bar consists of two buttons that enable the navigation through the prototype. This area is a static, too. The third area displays the content e.g. the dashboard, the document overview or the document that is edited currently so, this is changing dynamically. To see all users, that are currently available online and to add the possibility of choosing one for chatting or inviting there is another area at the left bottom that shows all online users. This is automatically updated if a new user appears. If a user decides to chat with another one a window pops up which is shown at the right bottom. This contains the whole conversation. Furthermore, incoming invitations are shown here. A detailed description of the client respectively the states of the content area is presented in Figure 4.5.

After loading the web page, the dashboard is shown within the content area and illustrates important figures about the collaborative editor as described in the use cases within Section 4.1.3. The buttons within the navigation bar enable the user to switch between the dashboard and the document overview (the `Documents`-state), whereas both are shown in the content area. The docu-

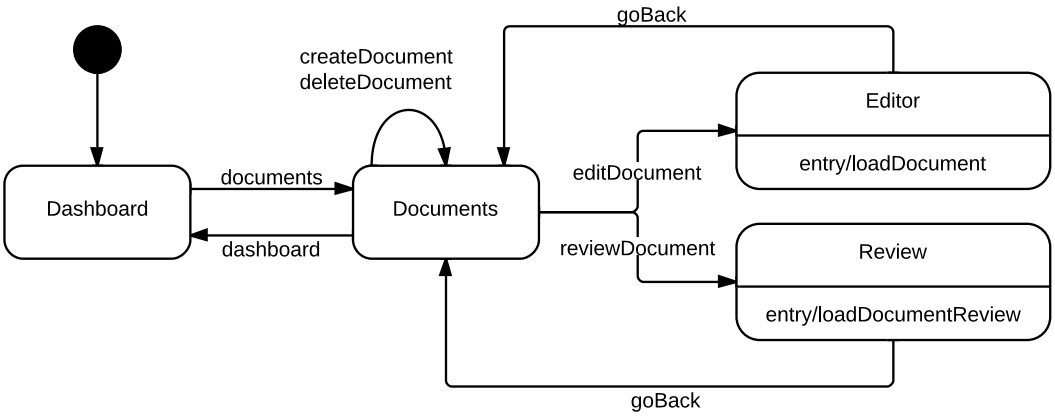


Figure 4.5.: The lifecycle of the client

ment overview presents all documents, that are currently available for editing. It offers also the possibility to create new or to delete existing documents. Furthermore this view enables the user to choose one single document for editing or reviewing purposes. When entering the `Editor`-state, the respective document gets loaded from the server and shown in a text editor that makes the editing of this document possible. The `Review`-state loads the document in all different versions from the server and enables users to comprehend changes that were done by different users since the creation of the document.

The `UserInterface`-component follows the Model-View-Controller-pattern as proposed within Section 4.2. As already described, this divides it into three parts. The model that holds the needed data, the view which is actually the presentation layer accessed directly by the user and the controller which is the interaction layer that connects the model and the view[101]. Each of these parts of the `UserInterface`-component is presented in the following.

The Model

The model holds all data needed by the user interface. Figure 4.6 shows the model, related to documents, in an entity relationship diagram in UML notation.

The `Document`-entity has the three attributes `Name`, `LastModified` and `NumberOfPeopleEditing`. This data is needed for the document overview, therefore, the `Document`-entity represents the meta information about a single document. The real document content, the rich text, is still on the server and gets loaded

[101] cf. [Ma11, p. 2-5]

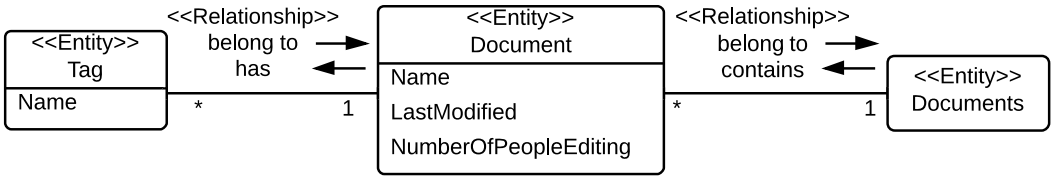


Figure 4.6.: ERD for Documents

on demand. One document can have multiple tags describing it, which is represented by the `Tag`-entity. All documents are kept within the `Documents`-entity.

Another entity within the model is the `User`-entity which is needed for multiple purposes. It is illustrated in Figure 4.7.

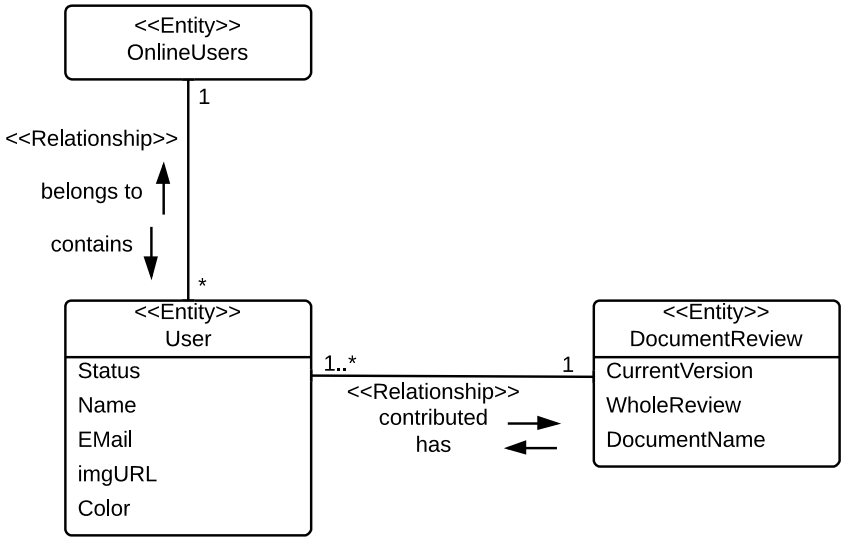


Figure 4.7.: ERD for Users

Every instance of the `User`-entity represents a user within the system. All these instances are kept in the `OnlineUsers`-entity which act as data source for the view, that shows all online users. Such a `User`-entity is also used to identify a user uniquely for chatting or inviting to a collaborative work. The `User`-entity has multiple attributes. `Name` is only used for displaying which applies for `imgURL`, too. This is an URL to an image which is shown next to the name within the user interface. The `EMail`-attribute keeps the email-address of a user and identifies each one uniquely. The status informs the other users about the activities, the user is currently performing. This fields is used for the activity feed. In addition, there is a unique color for each user

which is set by the server. It is used within the collaborative work to give every user's cursor one special color to see the place within the document, the user is currently working on.
Another purpose of the `User`-entity is for contributing to a document review. The first user who creates a document builds automatically the first version of it. Each time, another user manipulates the document another review is created, so, all modifications done by one user are grouped together to one version. This enables other users to always track the changes that were done for each participating user separately.

For enabling chat conversations within the collaborative editor, the `ChatConversaion`-entitiy is needed, which is shown in Figure 4.8.

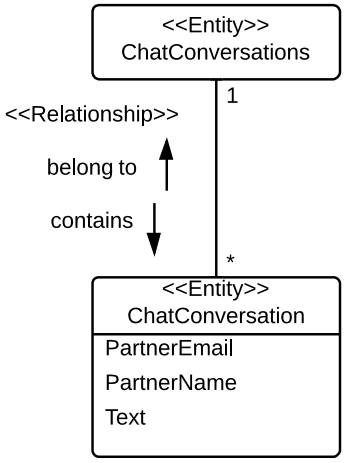


Figure 4.8.: ERD for Chat-Conversations

A `ChatConversation`-entity has the attributes `ParnterEmail`, `PartnerName` and `Text`. The partner email address is needed for identification. The combination of this address with the one of the user, the instance is based on, uniquely identifies a pair of two persons and therefore also a chat conversation. The `PartnerName`-attribute is only needed for displaying again. The real conversation text that both partners write and that is shown within the user interface is saved within the `Text`-attribute in form of a HTML-formatted string.
Another model, that is needed directly at the beginning of the user interaction is the Dashboard. Currently, this has only two attributes, `NumberOfDocuments` and `OnlineUsers`.

The Views

The views are based on the design of the user interface, shown at the beginning of this section. The following views are used within the collaborative editor:

application-view The `application`-view is the main view that gets loaded if a user visits the collaborative editor. It consists of other views loaded dynamically on demand, depending on the client state as described in

Figure 4.5. Therefore, it builds a frame for all views within the user interface.

dashboard-view The `dashboard`-view is the first view loaded into the `application`-view. It presents important figures like the number of online users or the number of documents, currently available within the collaborative editor.

onlineUsers-view Just as the upper two the `onlineUsers`-view gets loaded at the beginning. It is positioned at the left and shows dynamically all users that are currently online.

documents-view This view presents all available documents with their meta information. Furthermore, it enables the user to create or delete a document and to navigate to other views to edit or to review a document. It corresponds to the `Documents`-state, presented within Figure 4.5.

editor-view After choosing one concrete document for editing within the document overview (`documents`-view), the `editor`-view is shown that corresponds to the `Editor`-state within Figure 4.5. It shows the document itself within an editor that enables its editing.

documentViewers-view When editing a document which refers to the `Editor`-state this view is shown, too. It presents all the users, that are currently editing the same document.

chatWindow-view The `chatWindow`-view appears on demand, either when a user starts to chat with another one or if a chat message comes in. Furthermore, it is responsible for presenting invitations to a collaborative work.

review-view The `review`-view corresponds to the `review`-state within Figure 4.5. It presents one single document in all its versions that has been produced since its creation.

The Controllers

The controllers connect the model with the views and realize the business logic. There are several controllers that realize the client-side functionalities of the collaborative editor. Each one is presented in the following.

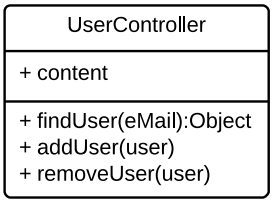


Figure 4.9.: User-Controller

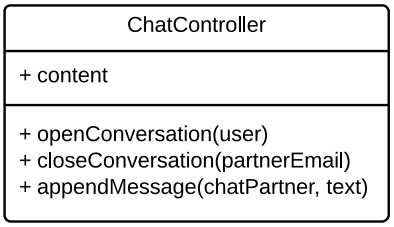


Figure 4.10.: Chat-Controller

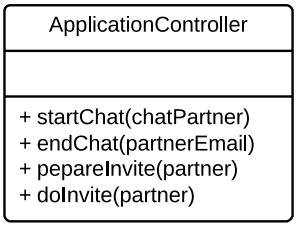


Figure 4.11.: Application-Controller

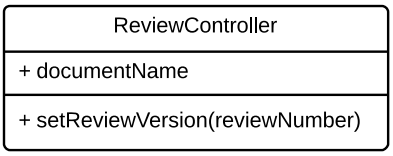


Figure 4.12.: Review-Controller

The `UserController`, presented in Figure 4.9 is responsible for managing the online users. This controller also builds the basis for the `onlineUsers`-view. It has an attribute called `content` which holds all the available online users. Therefore, it also realizes the `Documents`-entity of the model. It also enables common operations like the adding of new users or the removal of users that left the collaborative editor. Additionally, it offers an operation that enables the searching for users by their eMail-address which is used for identifying one user. The `ChatController` (Figure 4.10) enables the chat functionalities. To do so it offers three operations. The first one creates a new conversation with another user that is given via parameter. In a typical workflow, the `appendMessage` operation would be called multiple times afterwards. Each time an other user sends a new chat-message, this operation is called, to show up the new message within the `chatWindow`-view. The last operation is called if the user wants to end the conversation and it finally triggers the closing of the `chatWindow`-view. The `ApplicationController` shown in Figure 4.11 manages the `application`-view. Its four operations manage the events, thrown by the view. If a user clicks on another user and decides to start chatting with her, the first operation is called which forwards the request to the `ChatController` which applies for the second operation, too. The `prepareInvite`-operation triggers the opening of a new window that enables the user to select the document, she wants to edit collaboratively. If she chose the document, the fourth operation is called that does the invitation by first, sending an invitation to the

respective partner and second, by forwarding the user to the chosen document.

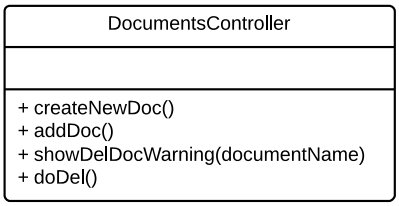


Figure 4.13.: Documents-Controller

The `ReviewController` (Figure 4.12) manages the `review`-view and is responsible for one single document-review. The respective attribute `documentName` refers to this single document. The operation `setReviewVersionNumber` triggers the view to show the single versions of the document. The `DocumentsController` manages the `documents`-view and the documents itself by offering operations that create or delete documents(`addDoc`, `doDel`). It also triggers the view to show up dialogues to confirm the creation or deletion (`createNewDoc`, `showDelDocWarning`). The `DocumentsController` itself is shown in Figure 4.13.

4.4.2.2. The Chat- and IO-Component

The `Chat`- and `IO`-component are responsible for managing the emitting and receiving of events through the system. On the one hand they send upcoming events from the client-side to the server, on the other hand they build the interface for receiving events from the server.
Although, both components have similar tasks they are divided into two different components to separate them by their functionalities and to increase the modularity. In the following, all events that are received or sent by the client are presented. For shorter descriptions, S stands for the server, C for the client and *cb* for a callback function. Table 4.1 describes the events the clients are listening for within the `IO`-component.

Event	**Parameter**	**Description**
who are you	-	S wants the identity of C.
set color palette	palette	S sets the colour palette of C.
set color	colour	S sets the unique colour for C.
newDoc	document	A new document was created.

availableTags	availableTags	S sets the available document-tags for C.
a new user appeared	user	A new client was registered S.
a user left	user	A client left the collaborative editor.
delDoc	doc	A document was deleted.
new tag	obj	A new tag was added to a document. `obj` contains both, the document name and the new tag.
remove tag	obj	A tag was removed from a document. `obj` contains both, the document name and the removed tag.
new document editor	user	C is editing a document. Another client opens the same document and needs to be added to the list of current collaborators.
editor left the document	user	C is editing a document. Another client, that was also collaborating at the same document, left. This one needs to be removed from the list of current collaborators.
initial documents sent	-	S sent all available documents to C.
set as writer	-	C is set as writer for the currently opened document.
set as reader	-	C is set as reader for the currently opened document.

Table 4.1.: Incoming events to the clients within the IO component

As already mentioned, the `IO`-component is also responsible for emitting new events that occur within the `UserInterface`-component. Table 4.2 shows all outgoing events that need to be distributed to the server or other clients.

Event	**Parameter**	**Description**
check in	user	Send the own identity to S .

new status	status	C has a new status that needs to be distributed for the activity feed.
storeAnd-Distribute-Document	document, cb	C created a new document. `cb` is called if the server received the event, which triggers the creation locally.
delDocument	docName, cb	C deleted a document. `cb` is called if the server received the event, which triggers the deletion locally.
new tag	docName, tag, cb	C added a new tag to a document. `cb` is called if the server received the event, which triggers the addition locally.
remove tag	docName, user, cb	C removed a tag from a document. `cb` is called if the server received the event, which triggers the deletion locally.
new editor joined	docName, user	C started to edit a document. It sends its own identity as a new editor.
editor left the document	docName, user	C left the edit-screen of a document. Tt sends its own identity to distribute the leaving.
create review	docName, cb	C wants to review a document. `cb` is called if S processed the review with it as parameter.
choose another writer	email, cb	C wants to hand over the writer-role to another client, identified by the email address. `cb` is called, when the new writer was notified.

Table 4.2.: Emitted events by the IO component

The purpose of the `Chat`-component is the similar, it emits and receives events. Here, within the scope of chatting. There are only two events in total, manage by the `Chat`-component. The `‘‘send a new message’’`-event is emitted for sending a new message. It that takes as parameters the respective chat partner and the text. Furthermore, the `Chat`-component listens for the event `‘‘there`

`is a new message''` which is used for a new incoming message. It has the same parameters: the chat partner and the text that was written by the partner.

4.4.2.3. The Editor-Component

A central part of this Thesis is the realization of a collaborative editor which enables users to collaboratively work on the same document. All users, that have the same document opened at the same time, can follow the edits of the single writer in realtime. The role of the writer can also by handed over to other users. To do so, the basic idea follows the concepts described within Section 4.2, too. This means, that the whole process of editing a document is event-driven. The part of the editor the user is interacting with is a rich text editor, that enables the user to not just only insert and edit text, but also include images, links, tables or formatted text. The content of the document that is created with this rich text editor is represented in form of the Hypertext Markup Language (HTML) so basically, every feature that is supported by HTML and the rich text editor itself is possible within the collaborative editor, too.

The process of collaborative writing starts with the user, typing something into the rich text editor. This one changes the underlying HTML document each time, the user does changes within the editor. The HTML text is a direct representation of the rich text created by the user within the editor. This text is now synchronised followed the principles of operational transformation, described in Section 2.1.2. Each time, the underlying HTML text changes, the client analyses the changes that were done and submits operations that represents these changes to the server. The server actually applies these operations to its local copy of the document and distributes them to the other clients. This process is illustrated in Figure 4.14.

This proceeding happens at the user, that is currently the writer. For readers, the process is the other way around. They receive the operations, that represent a content change from the current writer from the server. These ones are applied to the underlying HTML text which updates the content of the editor. Afterwards, the user can see the changes of the current writer.
In this form, operational transformation is not doing any conflict resolution.

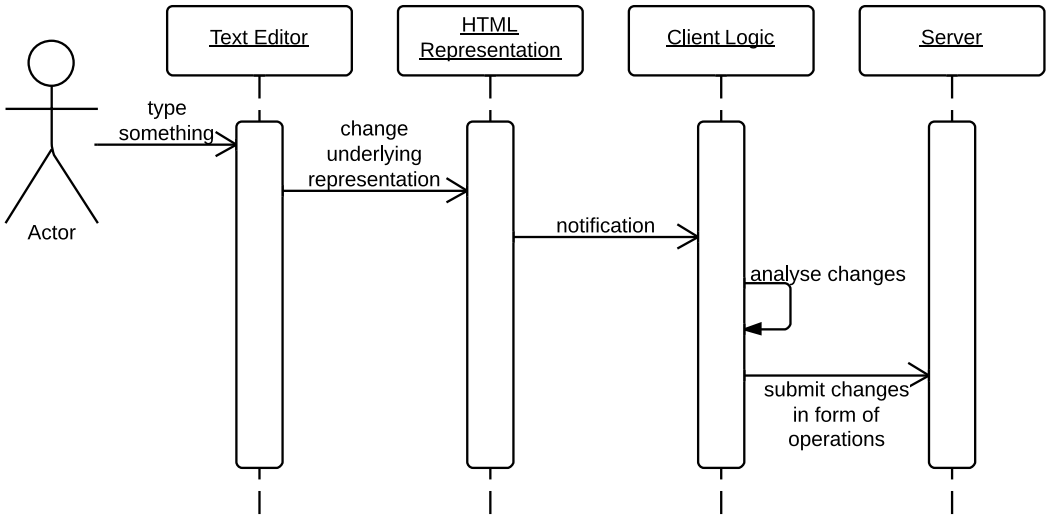


Figure 4.14.: Basic Process of collaborative editing

In fact, there are no conflicts as there is only one writer at any time. Nevertheless, the usage of operational transformation has major advantages. On the one side, the transmitted operations only represent the changes that were done to the underlying document. This keeps the data that needs to be exchanged between client and server quite small. On the other side, having operations as fundamental representation for document changes, makes versioning and reviewing possible.

4.5. Design of the Server

The server is one of the three main parts of the architecture of the collaborative editor, presented in Section 4.3. It mainly realizes the management of clients, the distribution of events through the system and the operational transformation backend. As these tasks differ, the server is divided into different components, too. Figure 4.15 presents an overview of all different components of the server.

The `Database`-component offers functionalities for persisting and reading data. Although, this is not a major task, there are still some important information that need to be stored persistently like the documents and their operations. The `DocumentManager` realizes all functionalities, needed for the document management like the creation and deletion or the addition or removal of tags.

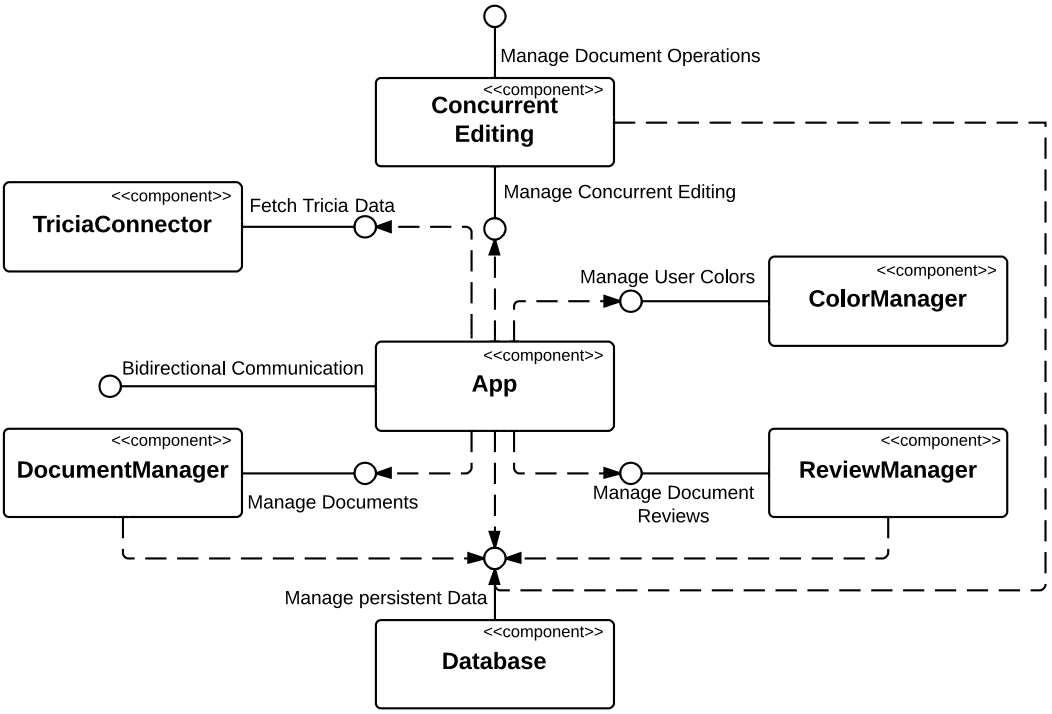


Figure 4.15.: Components of the server

The `ConcurrentEditing`-component manages all clients, working on documents collaboratively and realizes the operational transformation capabilities described in Section 2.1.2. To do so it offers a bidirectional communication channel that enables users to receive the operations, done by other users, or to send their own ones. This component is the counterpart of the `Editor`-component of the client and can be seen as the "server" within Figure 4.14. As already described, the `Database`-component stores all operations that were applied on a document. This enables the `ReviewManager`-component to create reviews for every document on demand. This is done by starting with the first version of the document and applying the single operations step by step.
The `TriciaConnector`-component offers a connection to Tricia and realizes the import of the currently available documents. The allocation and changing of colors for different users is managed by the `ColorManager`. In the middle of Figure 4.3 the `App` component is shown which is the main part of the server. It connects all the other components and manages their integration. Furthermore it offers a bidirectional communication channel which is used by the client(s) to receive or send realtime data. It can be seen as the counterpart to the `Chat`- and the `IO`-component of the client.

4.6. Initial Client-Server Communication

Based on the fundamental architecture presented in Section 4.3, the client is delivered from Tricia once. Afterwards the communication between the client and the server is autonomously. Nevertheless, there are multiple steps to do, to integrate the client into the whole system properly. This setup phase exchanges initial data to make it usable for its user and establishes the awareness of other clients. The whole procedure is presented within this section.
After the client was delivered from Tricia, each one connects itself immediately to the server. Figure 4.16 shows the respective flow of asynchronous messages of the involved parties to setup a new client.

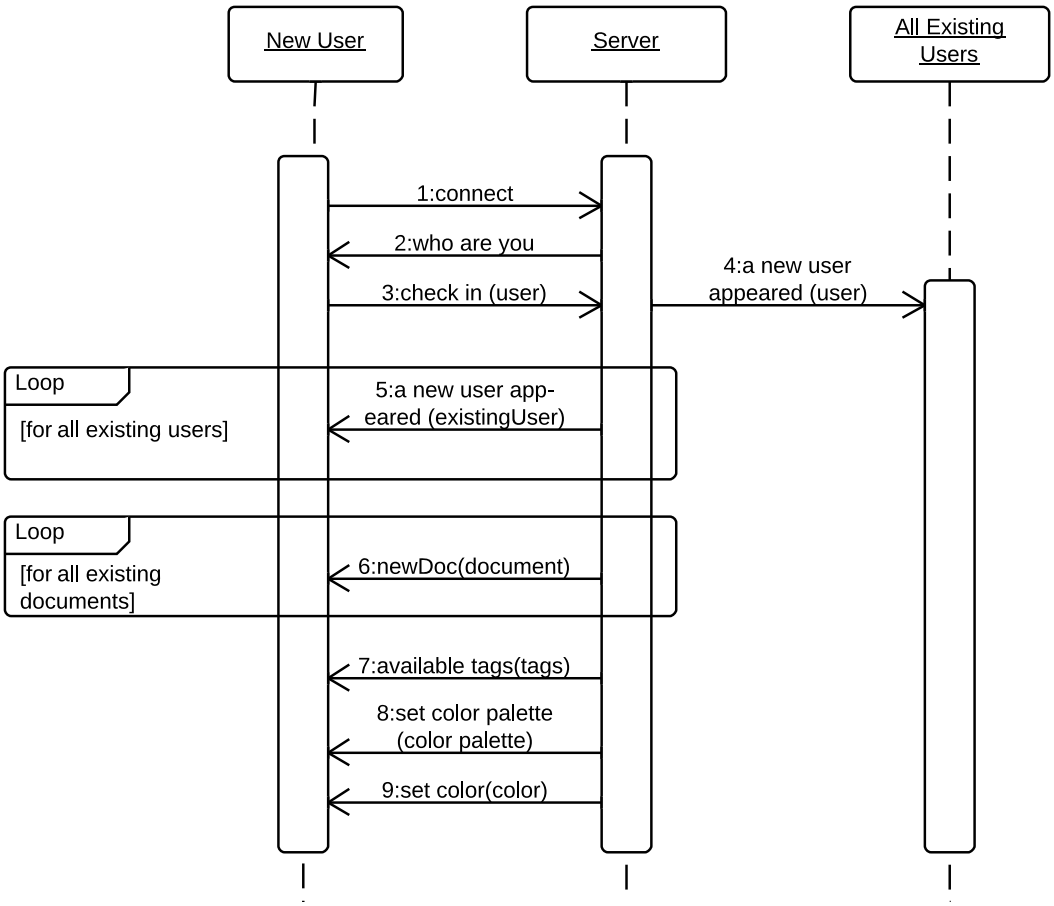


Figure 4.16.: Sequence Diagram for the integration of new users

After the client is loaded, which is after a successfully authentication at Tricia, it connects itself to the server. This is shown in message number one in Figure 4.16. The server emits a `'who are you'`-event to the client, which answers with his own identity afterwards (message three). This identity was injected

into the client by Tricia. If the server received the identity of the new user, it gets distributed to all other currently existing users with event number four `‘‘a new user appeared’’`. Subsequently, the new user receives all existing users (message number five), the metadata of all currently available documents (message number six), the available tags (message number seven), the initial color palette (message number eight) and a unique color (message number nine) sent.
After this initial synchronization, the client is fully integrated into the system and usable by the respective user.

4.7. Conclusion

This chapter presented a design for enabling realtime collaboration in the context of a realtime collaboration tool. There were different use cases described, that are supported by this design. The basis for this is built with the realtime architecture. Its main characteristic is that it is driven by user interactions and therefore by events. Furthermore it claims the application of the Model-View-Controller pattern on the client-side and the Publish-Subscribe pattern for the communication with the server. This makes it easy to distribute the events through the system and to adept the models of each client to the impact of them.
The collaborative editor itself is realized as a so-called Rich Internet Application and is delivered by Tricia. This chapter introduced the architecture of the client and the server part of this tool and made a brief introduction to its user interface. As the whole system is event-driven, the respective events were presented, that are used for communicating between the clients and the server.

5. Prototypical Implementation of the Collaborative Editor

This section gives an inside into the implementation of the collaborative editor and presents interesting extracts. The implementation itself follows the design, presented in Section 4. Section 5.1 introduces third-party libraries that are used by both, the client- and the server-side. Subsequently, Section 5.2 presents some details concerning the client implementation followed by Section 5.4, which does the same for the server.

5.1. Shared Third-Party Libraries

Within the collaborative editor, two major third party libraries are used. These are deployed in both, the client and the server side. Therefore, they are described their own sections. Section 5.1.1 described socket.io, Section 5.1.2 presents share.js.

5.1.1. socket.io

Socket.io is a Node.js framework that intends to enable realtime web applications for every browser[102]. To do so, it abstracts multiple different transport protocols and offers one WebSocket-like API to access them. The underlying transport protocol is chosen depending on the user's browser. Supported are WebSockets, Adobe Flash Socket, AJAX long polling, AJAX multipart streaming, Forever Iframe and JSONP Polling that were already introduced in Section 2.2. This enables socket.io to choose faster transport protocols like WebSockets for modern browsers and to support older browsers at the same time by choosing older transport technologies[103]. All supported web browsers

[102] cf. [Ra12b]
[103] cf. [Ra12a]

are shown in the appendix in Section B.1.

Technically, the realtime communication is realized by listening for events on the one side and emitting events on the other side. This approach fits ideally into the realtime architecture presented in Section 4.2. For illustrating on how this is implemented with socket.io, Listing 5.1 shows some lines of code from the server. This code realizes the first four messages presented in the sequence diagram in Figure 4.16.

Listing 5.1: socket.io example from the server

```
1 socketio.sockets.on('connection', function (socket) {
2   socket.emit('who are you');
3   socket.on('check in', function(user) {
4     socket.broadcast.emit('a new user appeared', user)
5     //........
6   }
7   //........
8 }
```

In the first line, the server listens for initial connections from clients, which is represented by the `connection`-event. If such an event occurs, the anonymous function within line one is called. This gets a socket as argument passed, which must be used for further communication with this client. The sending of a new event to this newly connected client is done via the `emit`-function of the socket object, which can be seen in line two. Here message number two of Figure 4.16 is shown. Afterwards the newly connected user must be propagated to all other clients to make her visible in their lists of online users. This is done via the `broadcast`-function within line four which is also the implementation of message number four of the sequence diagram.

Another important functionality of socket.io are rooms. Rooms enable the grouping of clients which means that events can be emitted to a special number of clients, respectively these ones within a room. It can be seen as the implementation of the channels, mentioned in Section 4.2. In the collaborative editor, rooms are used for grouping the clients that edit or view the same document. Listing 5.2 shows a code snippet out of this context.

Listing 5.2: Code snippet that shows the usage of rooms

```
1 socketio.sockets.in(documentName + '_editors').emit('new document editor', user);
2 socket.join(obj.docName + '_editors');
```

If a new user opens a document, all users that currently have the document opened are notified about the new participant. This is done via a special group that is created for each document. The name of this group is the concatenation of the document name and the string "_editors". This notification is done in Listing 5.2 in line one. If this group does not already exist, it is created automatically. Afterwards, the new user joins the group for the editors of this document, too, to get also updates about users that join this group or leave it. The joining procedure is illustrated in line two with the "join"-function of the socket object, that belongs to the new user.

For a clearer separation of the messages, the socket.io connection is multiplexed. This enables the division of the single connection into multiple ones, which is done in socket.io with different namespaces. Within the collaborative editor, the default namespace and a namespace called "chat" is used. The second one is used for chatting, whereas the default namespace handles all other events. These two namespaces are used as interfaces for the `Chat-` and the `IO` component of the client, described within Section 4.4.

5.1.2. share.js

Share.js enables the concurrent editing of documents by providing operational transformation functionalities. To do so, it realizes multiple functionalities. On the one hand it offers an interface for the clients, which allows the sending and receiving of new operations. On the other hand it builds the system for the operational transformation mechanisms.
Share.js realizes its interface for connecting clients via so called "wire protocols", whereas two different ones are offered. One is a RESTful web protocol that enables the user to fetch the document, delete a document and submit new operations in a RESTful way via HTTP-requests and HTTP-methods. Further descriptions on this can be found in [Ge12d]. The collaborative editor uses the second wire protocol, the "Streaming Protocol". This offers all functionalities but the permanent deletion of documents as the RESTful protocol. In addition it allows the receiving of newly arrived operations as soon as they

occur[104], which is very important for fitting into the event-driven nature of the realtime architecture of the collaborative editor.
The streaming protocol is based on technologies presented in Section 2.2 and uses a JSON message format for the communication. It is realized with the two other libraries socket.io (see Section 5.1.1) or BrowserChannel whereas BrowserChannel is currently the standard. Therefore this one is used in the collaborative editor, too. From a functionality point of view both are comparable. The BrowserChannel library calls itself even "google's version of socket.io"[105] as it is developed by the former Google Wave developer Joseph Gentle and was used in the GMail chat[106].
A typical communication of a client and a server using the streaming protocol is illustrated in Listing 5.3.

Listing 5.3: Client-Server communication with the streaming protocol[107]

```
1 S: {auth:'90b657dc1498061fb7b974740c21395d'}
2 C: {doc:'holiday', open:true, create:true, type:'text', snapshot:null}
3 S: {doc:'holiday', open:true, create:true, v:0, meta:{creator:'Sam', ctime
     :1327379131999}}
4 C: {v:0, op:[{i:'Hi!', p:0}]}
5 S: {v:0}
6 C: {v:1, op:[{i:' there', p:2}]}
7 S: {v:1, op:[{i:'Oh, ', p:0}], meta:{...}}
8 S: {v:2}
```

At the beginning of a communication the client connects to the server. The server manages each client with a session ID which is generated at the server and sent to the client within line one. Afterwards, the client opens a document called "holiday" with the type "text", meaning, that a pure text is going to be edited concurrently. Share.js currently supports two document types, text and JSON, whereas the JSON type is unstable at the moment and may change at any time[108]. If the document does not exist yet, it should be created automatically by the server, which is indicated by `create:true` . If it does exist, the client receives a snapshot of the current version. Within line three, the server sends its response to the client. `v:0` indicates that the current document version is 0, meaning, that is was newly created by the server. This is

[104] cf. [Ge12d]
[105] cf. [Ge12b]
[106] cf. [Ge12b]
[106] cf. [Ge12d]
[108] cf. [Ge12c]

also the reason why there is no snapshot included in the answer. In addition, the server sends some meta information, like the creator and the creation time of the document. This was the initialization phase, the real document editing follows from line four. Here the client sends a new operation that inserts the string "Hi!" (`i:"Hi!"`) at position zero (`p:0`). This operation is done on document version 0, which is also confirmed by the server at line five. The current document version was incremented now. At line six, the client sends a new insert operation on document version one. This is followed by a message from the server, meaning, that there was another client submitting an operation on document version one, that inserted "Oh, " at the beginning of the document (position zero). In the last line, the server confirmed the receiving and application of the operation. It was applied on the document in version two, therefore the current document version is three. To do so, the server transformed the operation from `op:[{i:' there', p:2}]` to `op:[{i:' there', p:5}]` to include the changes of the operation submitted by the other client. The final content of the document in version three is now "Oh, Hi there!".
Listing 5.4 shows the case, where another client connects to the server, after all steps above were done.

Listing 5.4: Client-Server communication with the streaming protocol - A new Client connects[109]

```
1 C: {doc:'holiday', snapshot:null}
2 S: {doc:'holiday', v:3, type:'text', snapshot:'Oh, Hi there!', meta:{...}}
```

Again, the client opens the document and states, that she wants to receive a snapshot of the current document in line one. The answer from the server within line two differs now from that within Listing 5.3. Now, the document version is three and the current snapshot of it is delivered to the client.

These were some typical messages exchanged by the client and the server to use share.js and to edit a plain text document concurrently. This form of communication happens a lot when a document is edited collaboratively. The one writer continuously sends new operations to the server which applies them to his local copy. These operations get additionally distributed to all clients, that apply them to their copy, too.

[108] cf. [Ge12d]

5.2. Prototypical Implementation of the Client

This section presents some details about the client implementation. Section 5.2.1 introduces all modules and libraries and describes the building of modules in detail. Section 5.2.2 introduces the implementation of the user interface, followed by some implementation details about the `Editor`-component and the editor overview in the following sections.

5.2.1. Modularisation and libraries

The implementation of the client has a big code basis and makes use of a lot of third party libraries for enabling special functionalities. To reduce the complexity, the code is divided into different modules. Figure 5.1 shows all used libraries and self-written modules whereas all third party libraries are briefly explained in the appendix in Section B.2.

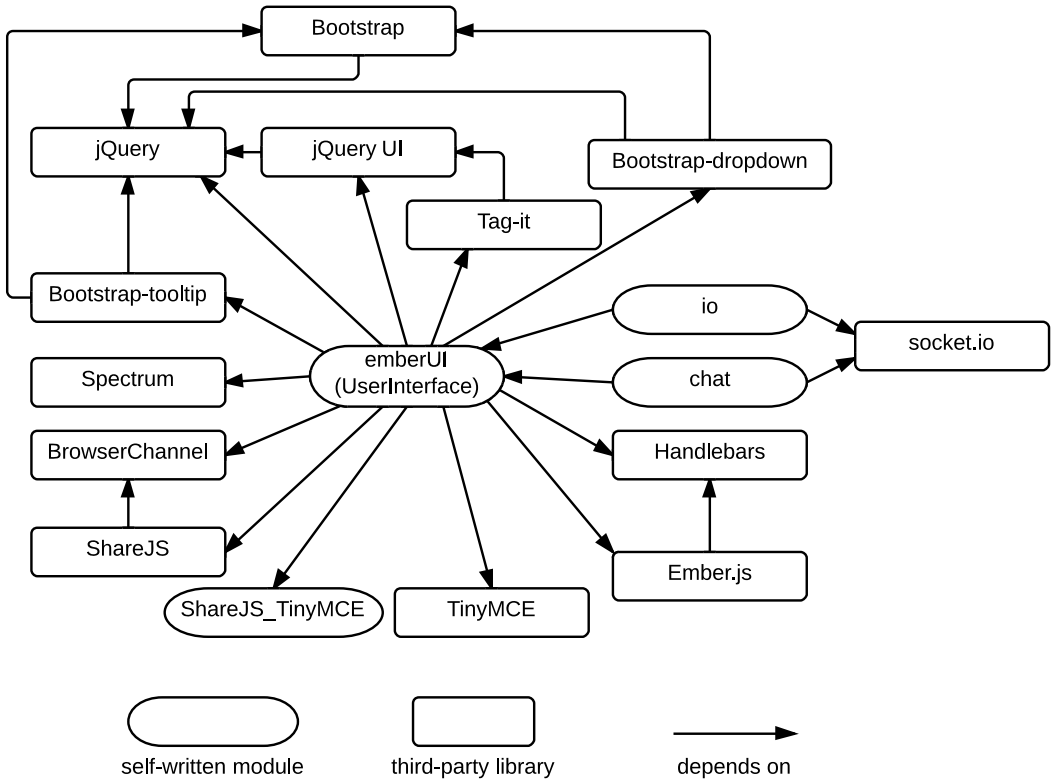


Figure 5.1.: Modules and libraries with their dependencies

As it is shown, there are 17 different modules respectively libraries, with different dependencies according to their specification. This can easily lead to

an unmanageable complexity. For reducing this complexity, the framework RequreJS is used. "RequireJS is a JavaScript file and module loader"[110], which uses the Asynchronous Module Definition (AMD) API for JavaScript. This specifies a way on how to define modules, their dependencies and how they can be asynchronously loaded[111]. Such a loaded module is not within the global namespace as it would be the case when using the normal script tags, for example. Instead, it is a scoped object that can define its dependencies and use them via a handle. A module itself can also offer methods by returning them within the `define`-function. Another positive effect of modules being not in the global namespace is that different versions of a library or a module can be loaded within the same page without conflicts.
Listing 5.5 shows an extract of the `emberUI` module to show how a module is defined. This module is the implementation of the `UserInterface`-component.

Listing 5.5: Exemplary definition of the emberUI module

```
1 define( [ "jquery", "shareJs_tinymce", "emberjs", /*...*/ "sharejs" ],
2   function($, shareJsPlugin) {
3   //...
4   return {
5     removeUser : function(user) {
6       CollaborativeEditor.userController.removeUser(user);
7     },
8   //...
9 });
```

A module definition always starts with the `define`-function. Its first argument is an array of strings that defines the modules, the new one depends on. Each of these strings uniquely identifies a module which is described later in detail. The second argument is the so called "definition-function" that will be automatically invoked to define the module after all dependencies are loaded. The definition function gets passed the dependencies in the same order, as they were defined in the array before. The names of these parameters are now the handles that allow the access to them within the new module. In line two, only two parameters are passed, as they are directly accessed within the definition-function. The other modules defined as dependencies are simply loaded. The last step within the definition function is the returning of an object that makes data and methods available for other modules. Line five shows for example

[110] [Bu11b]
[111] [Bu12]

the `removeUser`-function, which is one method within this object. It changes the model within the `emberUI`-module and is later called by the `IO` component if a user leaves the system.
To use this modular ecosystem an additional JavaScript file needs to be added that manages and configures all modules. In the collaborative editor this file called `main.js`. In fact, this is the only JavaScript file despite of the RequireJS library that is loaded via a script tag within the main html file. Listing 5.6 shows an extract out of this file.

Listing 5.6: RequireJS configuration

```
1  require.config({
2    paths: {
3      //...
4      jquery_ui:      '../tagit/jquery-ui-1.8.24.custom.min',
5      shareJs_tinymce: 'shareJs_tinymce',
6      emberUI:     'emberUI',
7      io:        'io',
8      chat:        'chat'
9    },
10
11 shim: {
12   'sharejs': {
13     deps: ['bcsocket'],
14   },
15   //...
16   'emberjs': {
17     deps: ['handlebars']
18   }
19 }
20 });
21
22 require(["jquery", "emberUI", "io", "chat"], function($, ui) {
23   $(function() {
24     ui.init();
25     //...
26   });
27 });
```

All libraries and modules reside in an own file. At first, all these modules and libraries get a string id assigned which is shown from line four to line eight. All those referenced files are JavaScript files, but as RequireJS expects that filetype anyway, the extension is omitted.

Unfortunately, not all modules are defined the way described in Listing 5.5. In fact most files are not, especially third party libraries. Therefore, there is no dependency declaration available for them and they would be loaded in a random order which would lead to an undefined behavior where the app would sometimes work if the order was correct, but most of the times not. Still, to work with them the dependency declaration can be caught up by using "shim config" which is done within the listing from line 11 to line 18. ember.js for example depends on handlebars, which is therefore always loaded first.
All the dependencies presented in Figure 5.1 are modeled via the above two ways. The self-written modules were already written the AMD-way, all other library dependencies are configure via shim.
In line 22, the `require` method is invoked. Here the main app logic starts. It loads the four modules described within the string array and in the background all needed dependencies. In the function, passed as the second argument, the `ui.init()`-function is called. From now on, the user interface is visible for the user.

5.2.2. User Interface with Twitter Bootstrap and Ember.js

The layout of the user interface is built with Twitter Bootstrap, a front-end framework for web development[112]. To meet the requirements stated in Section 4.4.2.1, the so called "Fluid layout" was used which is on the one hand structured as defined and on the other hand a responsive layout that scales automatically depending on the screen size. The collaborative editor additionally uses the lot of components of Twitter Bootstrap like buttons, dropdowns or its tooltip functionalities. In the end, Twitter provides a lot of CSS and JavaScript that was included into the collaborative editor for having a nice looking user interface. Furthermore, the user interface of Tricia will use Twitter Bootstrap in future which enables an easy adjustment of the user interface of the collaborative editor to fit into the design of Tricia.

As presented as part of the realtime architecture in Section 4.2 and in the design section of the collaborative editor 4.4.2.1, the user interface component is following the Model-View-Controller-pattern. For its implementation, the

[112][Tw13]

framework Ember.js is used. Basically, Ember.js follows five core concepts that support this pattern. The first concepts are templates. Templates describe the user interface via utilizing the Handlebars tempting language. Despite of HTML, such a templates can contain expressions, that directly map to controller or model data and automatically updates them, if the respective sources change. Furthermore they can contain so called outlets that act as placeholders for other templates. This functionality is used for the content area within the collaborative editor. It shows the dashboard at the beginning and changes its view or template during the client lifecycle for displaying an overview of documents, the editor page or the page that shows the reviews. Additionally, a template can contain views, that handles interactions.
The second core concept are views that are located within a template and handle, as already described, user events like clicks. Controllers, that are the third core concept, are responsible for storing the application state. They are also often the link between the models and the templates. This means that they fetch data out of the model and pass it to the template directly or translate it to a representation, the template expects. The fourth concept are the models, that actually store persistent state which is the data, needed by the application. The last core concept is the router. It manages the application state by mapping the URLs with the needed templates and models. It also updates the URLs when the user navigates through the application and goes through different states[113]. The available states were already described in Figure 4.5.

The usage of these concepts is described via code snippets in the following. Listing 5.7 shows an extract out of the `application`-template.

Listing 5.7: The application-template

```
<script type="text/x-handlebars" data-template-name="application">
  <div id="wrap">
    <!--Lot of Twitter Bootstrap div-tags for the top navigation bar ->
      <p class="navbar-text pull-right">Logged in as {{CollaborativeEditor.
          ownIdentidy.user.name}}</p>
    <!--closing div-tags ->

    <div class="container-fluid">
      <div class="row-fluid">
        <div class="span3">
```

[113] cf. [Ti13a]

```
10          <div class="well sidebar-nav">
11            <ul class="nav nav-list">
12              <li class="nav-header">Sidebar</li>
13              <li id="sb_dashboard" class="active sb">
14                {{#linkTo "dashboard"}}Dashboard{{/linkTo}}
15              </li>
16              <!-- Link to "Documents" -->
17            </ul>
18          </div>
19          {{view CollaborativeEditor.DocumentViewersView}}
20          <div class="well sidebar-nav">
21            <ul class="nav nav-list">
22              <li class="nav-header">Users</li>
23              {{#each item in CollaborativeEditor.userController.content}}
24                {{#view CollaborativeEditor.OnlineUserView userBinding="item"
                      }}
25                <!-- Shows the user information via e.g. {{item.name}} -->
26                <div {{bindAttr id="item.htmlId"}} class="dropdown-menu">
27                  <ul>
28                    <li><a tabindex="-1" href="#" {{action startChat item}}>
                          Talk to </a></li>
29                    <li><a tabindex="-1" href="#"{{action prepareInvite item}}>
                          Invite for collaborative work</a></li>
30                  </ul>
31                </div>
32                <!-- ... -->
33                {{/view}}
34              {{/each}}
35            </ul>
36          </div>
37        </div>
38        <div class="span9" id="container">
39          {{outlet}}
40        </div>
41    <!-- ... -->
42  </script>
```

The application template is the main template. When Ember.js is loaded initially, it looks for this template automatically and shows it. First of all, the template is embedded in a script tag with the type "text/x-handlebars" which shows, that the handlebars tempting language is used. There are a lot of div tags for using Twitter Bootstrap. Most important are the span-classes that describe different elements within the user interface. `span3` (line nine) for ex-

ample defines the content of the left frame of the page, where the navigation and the list of online users is embedded. `span9` (line 38) is the main area where the changing content is displayed. Within this template there is a lot usage of handlebars tags. In line four, there is an expression that directly links to the model, which contains the user identity. The link refers to the name, which means that this handlebar expression is automatically replaced by the value behind the name attribute of the model. In line 14, there is the `linkTo` helper shown. It gets replaced by a `<a>`- tag. The `linkTo` relates to document states, so if the link "Dashboard" is clicked, the router transforms the application state to the state "dashboard". In line 19, another view gets embedded, the `DocumentViewersView`, where all document viewers are shown. This relates to another template file which is included on demand. Line 23 realizes a loop that iterates through all online users. Afterwards, within line 24, a new view is created, which ends in line 33. This view is responsible for showing one user at the very left, including its name, image and current activity. It also realizes the functionality of chatting and inviting, which is shown in line 28 and 29. The `action`-tags are also replaced by `<a>`- tags, but they call a method of the application controller. If a user clicks on the link, implemented within line 28, the operation `startChat` of the application controller is called. The current item, which is the user, gets passed as argument. As already mentioned, this whole view is embedded into a loop through all users, which enables the described behavior for all users. Ember.js takes care of the updating automatically, which means, that if a new user joins, or if one leaves, which changes the model of the `userController`, the view gets automatically updated. In line 39, the `{{outlet}}`-tag is shown, which is a placeholder for other template files that get loaded into the template at this position.

Figure 5.2 shows the start page of the collaborative editor. It is directly generated out of the `application`-template, described above, with the dashboard included into the outlet. At the left, the two links "Dashboard" and "Documents" can be seen. These are shown within the template in Listing 5.7 in line 14 and 16. At the right top, the name of the currently logged in user is shown. This is stored within the "CollaborativeEditor.ownIdentidy"-object. The template directly refers to this model, which is shown within the template in line four. At the left, below the links, there is the list for all currently online collaborators with the headline "Users". As it can be seen, there is only one other user currently online.

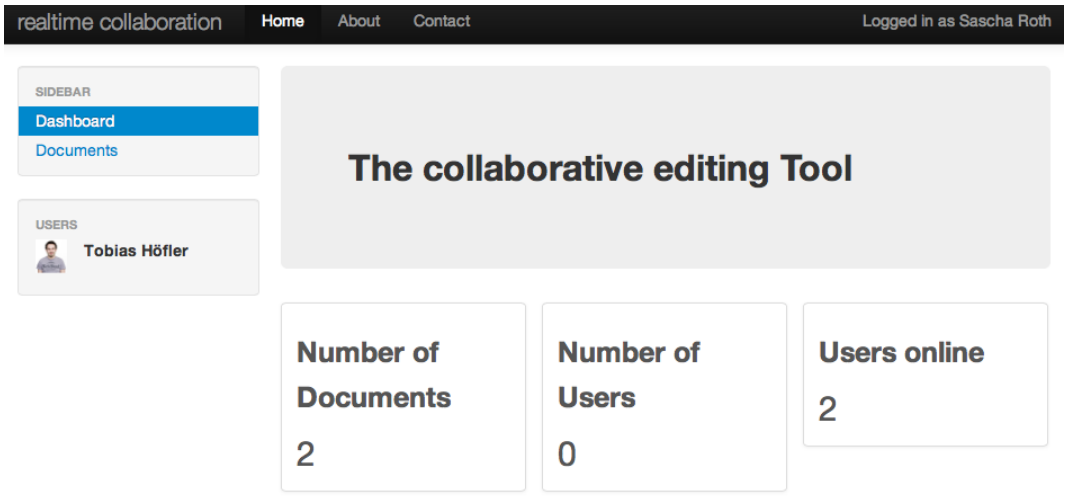


Figure 5.2.: The Collaborative Editor Start Page

In the template in Listing 5.7, this is realized from line 23 to line 34 within the loop. Furthermore, the Figure shows the dashboard, which is the first template loaded into the outlet after the start. It shows the number of documents and number of online users. The number in the middle is currently not active.

As already described, the outlet is the placeholder for the dynamic content area. It gets automatically replaced by other templates, fitting to the current application state. The managing of states is the responsibility of the router, whereas the respective routes of the router are shown in Listing 5.8. They also correlate to the UML Sate Diagram 4.5, presented in the client design section.

Listing 5.8: Router states

```
1 CollaborativeEditor.Router.map(function() {
2   this.route('dashboard');
3   this.resource('documents');
4   this.resource('editor' , { path: '/editor/:documentName' });
5   this.resource('review' , { path: '/review/:documentName' });
6 });
```

If a user starts the collaborative editor, she gets automatically redirected to the dashboard state. Ember.js automatically puts the template called "dashboard" into the outlet, which is directly related to the `DashboardController`. It also automatically matches the URL `/dashboard` to this state, if nothing else is

specified.
If a post processing needs to be done, before a state is entered, a resource can be used, which can be seen from line three to line five. An example is the review resource shown in line five. Its path is "/review/:documentName", whereas the part after the second slash is a dynamic segment, which is used for the real document name. A proper URL for the review page of the document "HelloWorld" would therefore be "/review/HelloWorld". Listing 5.9 shows the resource code for `review`-route.

Listing 5.9: Review Route with post processing

```
1 CollaborativeEditor.ReviewRoute = Ember.Route.extend({
2   model: function(params) {
3     return params.documentName;
4   },
5   setupController: function(controller, documentName) {
6     controller.set('documentName', documentName);
7   }
8 });
```

It contains two important methods, whereas Ember.js calls them hooks, as they can be used for the preprocessing. The `model`-hook is used to parse the URL into a fitting model[114]. The only part which is needed is the document name, so this is filtered out of the URL within line three and gets returned. The second hook `setupController`, can be used to set some attributes of the controller. The respective controller, which is the `ReviewController` in this case, gets passed as argument automatically. The second argument is the value returned by the `model`-hook. In line six, the attribute "documentName" of the controller is set to the value, returned by the `model`-hook, that parsed the name out of the URL. After the routing was done, the controller gets automatically loaded. This one now has resource to the dynamic document name, as it was set by the router within the preprocessing. All other resources are quite similar compared to the presented one.

Controllers are another major concept of the implementation of the user interface. To illustrate the usage with Ember.js, Listing 5.10 shows an extract out of the `ReviewController`.

[114] cf. [Ti13b]

Listing 5.10: Extract out of the Review Controller

```
1 CollaborativeEditor.ReviewController = Ember.Controller.extend({
2   documentName : "",
3   setReviewVersion : function(reviewNumber) {
4     //Build the requested version of the review by starting with the first
          document version and applying the operations continuously
5   }
6 });
```

To create a controller with Ember.js, the new one has to extend the `Ember.Controller`-class, which is shown in line one. Afterwards, special attributes and methods can be defined within the controller to make them usable for the other components. The `documentName`-attribute was already set by the router, presented in Listing 5.9. According to the design section of the client, the `ReviewController` additionally has a method called `setReviewVersion`, to set the current review version. This one is shown in line three.

The last important concept are the views. As an example, Listing 5.11 shows an extract out of the review-view implementation.

Listing 5.11: Extract out of the review-view

```
1 CollaborativeEditor.ReviewView = Ember.View.extend({
2   templateName : "review",
3   didInsertElement : function() {
4     var controller = this.get('controller');
5     //Set the document name from the controller
6     //Make use of the controller-function to set the current version of the
          review
7   }
8 });
```

For creating a Ember.js view, it has to extend the `Ember.View`-class, which is shown in line one within the listing. Every view can manually set its template which can be seen here within line two. In this case, it is not necessary, as they would be automatically matched because of their names. Also the view provides hooks. Often used within the collaborative editors views are the `didInsertElement`-hooks, that are called after the template was inserted into the DOM or the template was re-rendered[115]. In this case, the function is used to insert the document name from the controller into the template and

[115] cf. [Ti13b]

makes use of the controllers functionality of setting the chosen version of the review.

All the controllers, views and templates are implemented similar to the above presented ones, according to the design decisions, made within Section 4.4.

5.2.3. Collaborative Editor Details

The integration of an editor and the enabling of working collaboratively on a document is a major part of this Thesis. The principle technique which is used to work collaboratively is operational transformation which was presented within Section 2.1.2. Also the design decisions were already introduced in Section 4.4. This section focuses on the respective implementation.

5.2.3.1. From TinyMCE to Operations

One requirement for the development of the collaborative editor was the integration of TinyMCE, a web-based "what-you-see-is-what-you-get" (WYSIWYG) editor, implemented in JavaScript[116]. Furthermore, it was chosen to use the Node.js library share.js as operational transformation framework. To make them work together, the module `ShareJS_TinyMCE` was developed, which is presented in detail in the following. Share.js' client representation of the document is a `Document`-object[117]. It is a handle to the real document that resides at the server. The used methods of the `document`-object are presented within Table 5.1.

Method	Signature	Description
getText()		Get the textual content of the document
insert()	pos, text, [function(error, appliedOp) ...]	Submit an insert operation, whereas "text" is inserted at position "pos". The callback method is called, if the server acknowledged the operation. It furthermore provides error messages, if an error occurred.

[116] cf. [Mo12]
[117] cf. [Ge12a]

del()	pos, length, [function(error, appliedOp) ...]	Submit a delete operation, wheres a deletion from position "pos" for "length" characters was done. The callback method is called, if the server acknowledged the operation. It furthermore provides error messages, if an error occurred

Table 5.1.: Methods of the Share.js document-object

These methods build the basis for working with share.js. If the self-written module `ShareJS_TinyMCE` is loaded, the Share.js `document`-class gets extended by a function called `attach_tinymce`. Its primary task is to connect a Share.js document with a TinyMCE editor-instance. This function takes the DOM-element name of the TinyMCE instance as parameter. Additionally the email address of the user and a function, that returns the color of her. The `attach_tinymce`-function has the following two major tasks:

1. If a user with the current role as writer changes the content of a document within TinyMCE, the function first has to detect this change and afterwards find the changes to deliver a proper operation to the server via the share.js document-object. Additionally, the current cursor position has to be submitted properly on every change to make it visible for the other readers, too.

2. The readers continuously receive operations that are distributed by the server. Therefore the second major task of this function is the listening for these changes and the making of a proper adjustment of the content of TinyMCE.

The implementation of both functionalities follows the design, presented in Section 4.4.2.3. As the collaborative editor has an event-driven nature, it is also built with the usage of events.
To recognize a change within TinyMCE, meaning, that the writer inserted or deleted something, the implementation uses two events: `KeyUp` and `Click`, which are attached to the TinyMCE instance. The `KeyUp`- and `Click`-event were chosen, as they catch all cursor movement on the one side, on the other

side also recognize almost every insertion or deletion of a character. Unfortunately, elements that are included via the TinyMCE-buttons are not recognized by these events. These are for example tables or images. If these elements get inserted via the TinyMCE-buttons, they are only distributed after another `KeyUp`- or `Click`-event occurred. An Exemplary extract of the code for the `KeyUp`-event can be seen in Listing 5.12.

Listing 5.12: KeyUp-listener for recognising changes

```
1 tinyMCE.get(elementName).onKeyUp.add(function(e) {
2   setCurrentCursorPosition(doUpdate);
3 });
```

In line one, the attaching of the listener to the `KeyUp`-event can be seen, whereas the code for the `Click`-event is similar. The DOM-element name was passed to the `attach_tinymce`-function as parameter and the `get`-function returns the respective TinyMCE instance. If it emits the `KeyUp`-event, the anonymous function is called. Within, a function is called that sets the current cursor position into the document, whereas the `doUpdate`-function is given as parameter. In this example, the order of execution is important: first, the cursor position has to be set, then, the `doUpdate`-function must be called. To make this sure, the pattern of using callback-functions is quite typical for event-drive languages like JavaScript. This means, that the `doUpdate`-function is called within the `setCurrentCursorPosition`-function at the needed place.
The `setCurrentCursorPosition`-function itself inserts a special span element into the editor at the current position of the cursor. This span is styled as a colored cursor, which enables the user to see her own cursor in her own color. The embedding directly into the document also enables the distribution of the cursor to all other readers. So, they can follow live, what the writer is editing currently. If the role of the writer switches, the cursor must be positioned completely new and its color must be changed, according to the new writer's color.

Listing 5.13 shows the respective span element, that is embedded into the document and represents the writer's cursor.

Listing 5.13: Span-Element for user-specific cursors

```
1 <span id="cursor_' + authorId + '"></span>')
```

The respective `authorId` is built out of the email address of the user, whereas characters, that are not allowed as an ID for HTML-elements are replaced like the @-character.

After the insertion of the cursor at the right position, the `doUpdate`-function is invoked. Its purpose is to compare the document, which is currently within the editor with the document, that is currently at the server. As this one is only triggered, if either the `KeyUp` or the `Click` event was emitted, there is a difference in any case, at least a different cursor position.

The compare algorithm does a character-wise comparison of both documents from the beginning and from the end of both documents. This results in two numbers. The first one indicates the number of characters from the beginning, that are equal in both documents. The second one indicates the number of equal characters from the end of both documents. If the sum of these values is different to the length of the servers document a deletion was done. If the sum differs to the length of the new document an insertion was done.
The following example will explain the algorithm. The initial situation of all clients and the server version is, that the document contains the string "HelloWorld". One user now deletes the character "e", resulting in the string "HlloWorld", therefore, the version within her editor differs now from the server one. The result of comparing these versions are two numbers. From the beginning, there is one character equal, from the end, there are eight ones. The sum of theses values is nine, which differs from the length of the old document, which was ten, so a deletion was done. At the same time, it is equal to the length of new document, therefore, there is no insertion. Lets assume, the user inserted a character like an underscore, resulting in the string "Hello_World". The number that counts the same characters from the beginning is now five, the number that counts from the end is five, too. The sum of those is ten, which is equal to the servers' document length, but different, to the editors' one, so an insertion was done.
This proceeding enables the client to decide for each change, whether an insertion or a deletion was done and to submit the proper operations to the server. The algorithms complexity is $\mathcal{O}(n)$ and it was developed by Joseph Gentle, published at [Ge13].

In addition to the functions the document-object of share.js offers, it also emits events. The two events that are used by the collaborative editor are `insert` and `delete` , which are emitted if a new operation was done at the server. In addition, they provide the respective position and the text, that was added or deleted. With their help, the second functionality of the `attach_tinymce` module is implemented.
For a proper handling of these events, a special class called `OperationWrapper` was developed whose class diagram is shown in Figure 5.3.

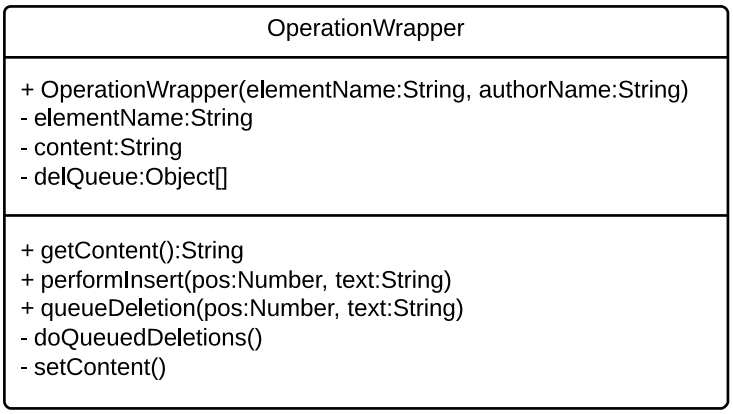


Figure 5.3.: The OperationWrapper class

The `insert`-event, emitted by the share.js document-object provides the new text and its position. It is directly forwarded to the OperationWrappers function `performInsert()`. The deletion is passed to `queueDeletion()`. The OperationWrapper then manages the application of the operations to the content of TinyMCE. As the name of the function already implies, the deletions are queued, whereas insertions are applied immediately. The reason for this is TinyMCE's internal behavior of automatically normalizing its underlying HTML as this one is used by the client, to apply the incoming operations.
A single change of the writer can lead to multiple insertions or deletions. If these changes would be applied directly it can lead to a malformed HTML which would then be normalized by TinyMCE. Afterwards, different users would have different HTML code basis depending on their current role of reader or writer. The relative positions would not be the same anymore which would make the application of OT impossible. Therefore it is important to apply operations that result out of one change together before setting it into TinyMCE. To reach this behavior, the `OperationWrapper` first loads the HTML content out of TinyMCE into the `content`-attribute. If only an insertion needs to be

done, the operation is applied immediately to `content` which is then set into TinyMCE. If a deletion arrives it might belongs to only one change with a corresponding insertion. Therefore, the deletion is queued. If no operations arrives within a certain timeframe, it gets applied, too. If another insertion comes in, all deletions are applied first and afterwards the insertion. All these changes are done on the `content`-attribute first, before setting it into TinyMCE.
This proceeding assures, that each collaborator always has the the same document as every other.

5.2.3.2. The Editor in Action

Figure 5.4 shows a screenshot of the editor.

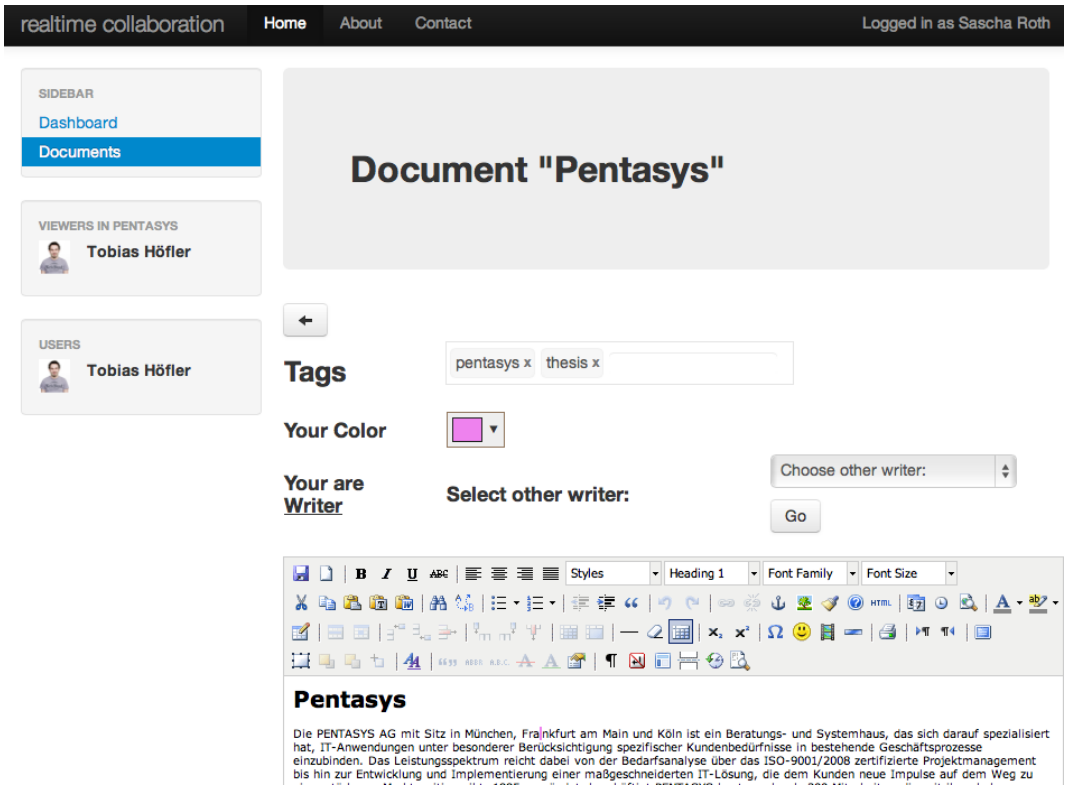


Figure 5.4.: The Editor in Action

The document “Pentasys” is currently opened and there are two writers working collaboratively. At the left within “Viewers in Pentasys”, all other users are shown, that have currently the same document open. If the current writer leaves the document via the back-button, another writer is chosen automatically. So, the collaborators are “Sascha Roth”, who is currently logged in at

the client shown in the screenshot, the other is "Tobias Höfler". As it is shown, "Sascha Roth" is currently the writer, which means also, that "Tobias Höfler" is currently set as reader. This can be changed via the drop down menu, shown in the middle at the right. At the bottom, TinyMCE is shown with all its menu bars to support rich text editing. The tag-bar in the middle enables the users to add and removes tags. These ones are distributed automatically to all other collaborators, looking at the document.

5.2.4. Implementation of the Editor Overview

Larger documents often do not fit into the editor without the need to scroll. This leads to a loosing orientation and overview for the readers. On the one hand they cannot realize the document as a whole, on the other hand they cannot follow the cursor of the writer anymore, as this one may be out of their current viewport. To address this issue, the editor overview provides an overview of the whole document including the images of the users to mark their current positions within the document.
To implement this feature an iFrame is created directly under the editor which contains a copy of the document within the editor. It gets automatically updated on every change. For providing the overview-character, it has 40% of the size as it would normally has. Listing 5.14 shows the respective CSS for reaching this.

Listing 5.14: CSS for the overview

```
1 #documentOverview {
2   width: 240px;
3   float : left ;
4   transform: scale (0.4,  0.4) ;
5   -ms-transform: scale(0.4, 0.4); /* IE 9 */
6   -moz-transform: scale(0.4, 0.4); /* Firefox */
7   -webkit-transform: scale(0.4, 0.4); /* Safari and Chrome */
8   -o-transform: scale(0.4, 0.4); /* Opera */
9 }
```

The major part of the CSS is the resizing, presented from line four to line eight. It shrinks to whole iFrame to 40% of the normal size. As it can be seen, there are browser specific statements needed for the realization.
Furthermore, the original cursor of the writer is replaced by her image, as the

normal cursor would not be visible anymore clearly. To implement this, the whole document is processed each time an update is done. It is scanned for the `span`-element that belongs to the cursor of the writer. With the help of the ID of the cursor element, the former cursor position can be mapped to the user images and replaced by `img`-elements that include the image of the user. Figure 5.5 shows an overview of an exemplary document.

Figure 5.5.: Exemplary Editor Overview

It is shown, that the cursor is replaced by the image of the writer. The upper left corner of the image is positioned at the former cursor position.

5.3. Integration into Tricia

Tricia is a major component of the collaborative editor. Although from an architectural point of view it resides at the server-side it is described in a separate section, as it also makes important pre-processing tasks for a proper client-code delivery and as it is intentionally separated from the self-written server. Tricia realizes multiple functionalities. First it does the authentication and authorization and delivers the client-code if the client is registered at Tricia. Furthermore, it provides user information and the documents that will be edited within the collaborative editor. The implementations of these tasks are described in the following.

For meeting the requirements of the collaborative editor, Tricia's functionalities are extended by three new so-called "Handlers". Handlers are mapped to a single URL, depending on the class-name of the respective Handler and the package, the classes are placed in. Such a handler can return special replies, whereas `SimplePage` and `JsonAnswerStation` is used. The first one returns, as the name already implies, a simple page and the second one returns a JSON formatted object. The typical workflow of the interaction of a user with Tricia is shown in Figure 5.6.

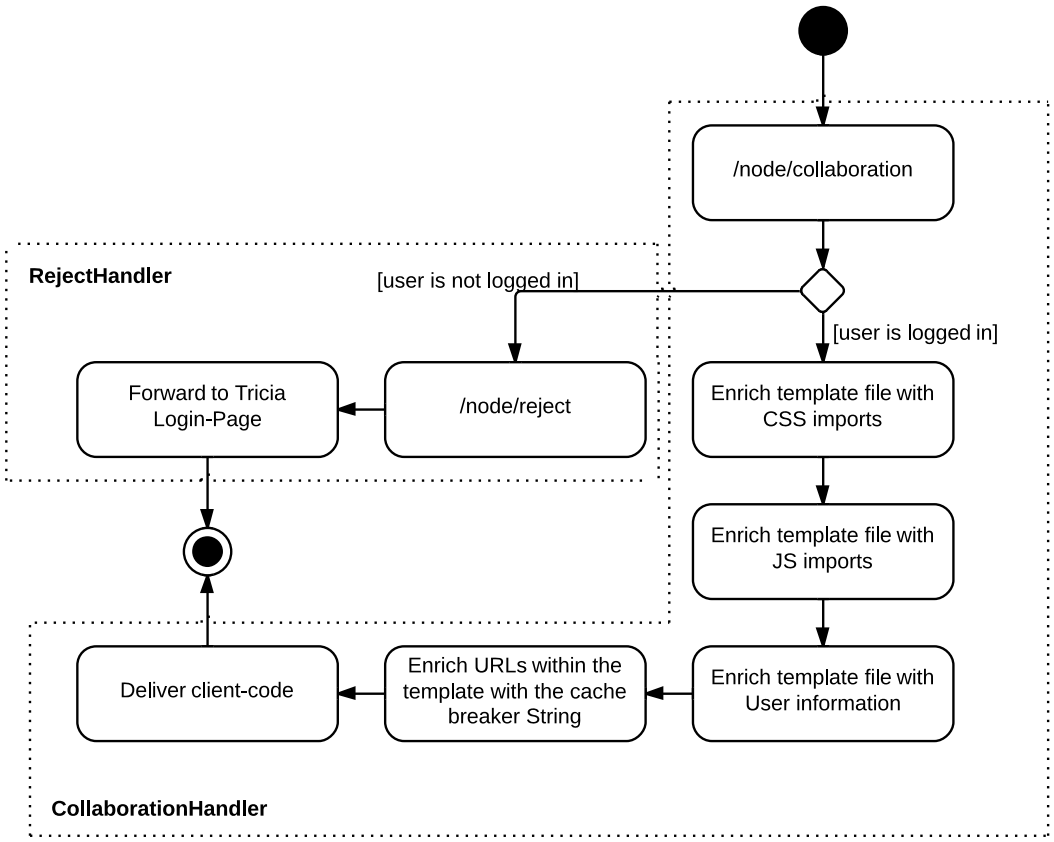


Figure 5.6.: Interaction workflow with Tricia

The `CollaborationHandler` is mapped to the URL "/node/collaboration" as it is within the package "node" and as the name begins with "Collaboration". Its first task is to check, whether a user is logged in in Tricia. If not, she gets redirected to "/node/reject", which is managed by the `RejectHandler` that shows a page with an information for the user and the possibility to go to the Tricia login page. Both handlers return a `SimplePage` that has a underlying template file which is the page that gets delivered to the user. The template file of the `RejectHandler` is a simple html page that is delivered to the user, whereas the template file of the `CollaborationHandler` is more complex. It has special placeholders that get replaced by the handler to construct a target html-file before it is delivered to the user. These placeholders stand for the CSS and JavaScript imports of the html-file. In fact, this HTML and the JavaScript files contain the whole client code. Additionally, the user information of the currently logged in one gets included. These are the name, the email address of the user and the URL to the image, the user provided in her Tricia profile. All addressable pages, CSS and JavaScript libraries in Tricia have a special

cache breaker string within the URL. This one gets replaced at last within the template file. Now, the page is ready to get delivered to the client.

In addition, there is `AllDocumentsHandler`. It returns all the documents currently available in Tricia in form of a JSON response. To do so it has the `JsonAnswerStation` return type, which automatically translates the document objects of Tricia to JSON. This is accessible via "/node/allDocuments".

5.4. Prototypical Implementation of the Server

The server part of the collaborative editor is implemented with Node.js, which was introduced within Section 2.4. This proceeds with the event-driven nature of the whole system also at the server-side. Within this section, the server implementation is described in detail. Section 5.4.1 describes the Implementation of each module in detail, including the used third-party libraries and the interfaces exposed for other modules.

5.4.1. Implementation of the Server Components

The server consists of different components as described on a high-level in Section 4.5. These components are partially based on third party libraries or are completely written in own JavaScript code. For giving a first impression about the different parts of the prototype and the usage of libraries and technologies, Figure 5.7 shows again the different server components, now with the focus on the involvement of major third party libraries and technologies. Here, one component, the `CommunicationManager`, was added. It is fully realized with socket.io and manages the communication with the clients. The introduction of this new modules leads to clearer responsibilities for the implementation.

The components `App`, `TriciaConnector`, `DocumentManager`, `ReviewManager` and `ColorManager` are, beside of the usage of the database client, completely self-written. The central `App`-component integrates all parts and makes use of functionalities of socket.io for enabling the bidirectional communication with the client(s). The `CommunicationManager` itself represents the functionalities of socket.io. The `ConcurrentEditing`-component is realized by share.js. The `Database`-component is realized with Redis.

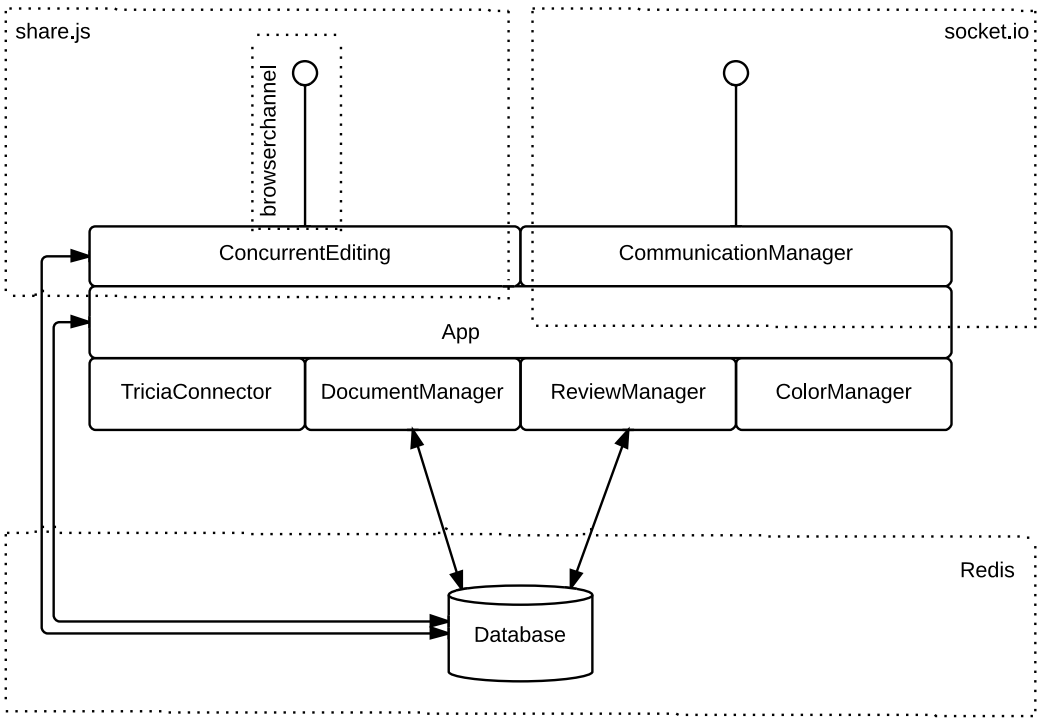


Figure 5.7.: Server Components with major third-party libraries

In the following, all parts are described in detail including the reasons for using the libraries respectively technologies.

5.4.1.1. The App Component

The `App`-component is the central part of the server-side of the collaborative editor. Listing 5.15 shows the first ten lines of the app component, that import all needed modules.

Listing 5.15: Includes Modules

```
1  var http = require('http');
2  var express = require('express');
3  var io = require('socket.io');
4  var sharejs = require('share').server;
5  var redis = require("redis");
6
7  var triciaConnector = require('./triciaConnector.js');
8  var documentManager = require('./documentManager.js');
9  var colorManager = require('./colorManager.js');
10 var reviewManager = require('./reviewManager.js');
```

The "http" module within line one is the Node.js core module for providing a HTTP server. It builds the basis for further modules that need to communicate with the client(s). The following four modules were installed into the *node_modules* folder via the node package manager NPM. Express is a special "web application framework for node" [118]. It is needed to act as a request listener for Node's HTTP server[119] which means, that incoming requests are forwarded to express. This again is needed by share.js and socket.io. Both libraries are attached to this express request listener, to listen for special requests addressed to them. This is the only reason why express is needed. The "redis"-module is the client library for the Redis database. The last four lines within Listing 5.15 include the self-written modules, that are presented in the subsequent sections.

After the import of all needed modules, the integration of the request listeners and the start of the HTTP-server, the `App`-component initiates the `TriciaConnector` to fetch all documents from Tricia and to give them to the `DocumentManager`, that stores them.
Until now, the `App`-component managed the initial start of the collaborative editor.
The second task of it is to realize the bidirectional communication with socket.io. To do so, it implements listeners for all events emitted by the clients which were already described in Table 4.2. Furthermore is listens for the `connect`- and `disconnect`-event that get emitted if a user connects or disconnects. The `connect`-event was already described in Listing 5.1. The `disconnect`-event follows a similar proceeding the other way around. If a user is disconnected

[118] cf. [Lo04]
[119] cf. [Jo13b]

the event "a user left" gets broadcasted to all connected clients to inform them about this event.
As it was already described, the bidirectional connection of socket.io is multiplexed. Listing 5.16 shows how this is implemented.

Listing 5.16: Multiplexing of the socket.io communication channel

```
1 var chat = socketio.of('/chat');
2 chat.on('connection', function (socket) {
3     //Further listeners
4   })
5 });
6 //....
7 socketio.sockets.on('connection', function (socket) {
8   //Listeners
9 });
```

The "chat"-namespace is realized within line one. It gets addressed by appending "/chat" to the connection URL at the client. The default namespace, that is realized in line seven, has no need of appending a string to the connection URL. In line three and six, all the listeners are now implemented separately. Now, the server-side is started and initialized and ready to listen for events. From this moment, the clients are able to connect and work with the server.

5.4.1.2. The Concurrent Editing Component

The concurrent editing component is fully realized with share.js. Nevertheless, there are some steps to do for its integration. Listing 5.17 shows the respective code snippet.

The options are passed to share.js via an object which is defined from line one to line ten. Line two defines the usage of the Redis database for storing the document meta data and all the operations. In line three, the cross-site access for BrowserChannel is enabled by setting the "Access-Control-Allow-Origin" header. This is needed, as the user originally received the data from Tricia, which runs on a different port than the Node.js server. More on this topic can be found at [Mo13]. At last, a special authentication method is set, if a new user connects to the server-side for getting access to the documents. Currently, there is no real authentication done, instead, this message easily provides the respective session-ID of a user an its identity, which can then be matched for

later usage. It is also shown, that all requests from clients are accepted via the function `action.accept()` within line eight.

Listing 5.17: Needed options for share.js

```
1  var options = {
2    db: {type: 'redis'},
3    browserChannel : {cors:"*"},
4    auth: function(agent, action) {
5      if(action.type =='connect') {
6        //Store the sessionID of the new user
7      }
8      action.accept();
9    }
10 };
11 sharerejs.attach(app, options);
```

At last, the share.js server-part is attached to express.js-server that is stored within the `app`-variable. The respective options described below are passed as parameter. From now on, the clients can use the share.js backend to work collaboratively on a document.

5.4.1.3. Database with Redis

There is little information needed to be stored persistently at the collaborative editor. To do so, the key-value store Redis[120] was chosen. It keeps all data in the RAM and periodically saves it to disk, which offers high performance and persistency at the same time. As there is only little data that need to be saved, the data probably won't exceed the ram size, which leads to a predictable high performance[121]. At last, there is no complex relational schema needed and a key-value store fits better to this scenario.

Although, there is no complex schema, a simple hierarchy is built with the keys, to get all needed data comfortably. Given a concrete document name keys are built as shown in Listing 5.18.

[119] cf. [Ge12d]
[120] cf. [VM03]
[121] cf. [Wa11]

Listing 5.18: Key schema of the key-value store

```
1 "docs:" + documentName + ":lastModified"
2 "docs:" + documentName + ":tags"
3 "ShareJS:ops:" + documentName
4 "ShareJS:doc:" + documentName
```

Line one presents the key for the date of the last modification for the document with the name "documentName", line two the respective tags. The modification date is a normal number value whereas the value for the tags is a list. Line three and four are keys created and managed by share.js and keep the document-meta information and the respective operations, that were submitted by the users. For getting a list of all documents and for building the needed keys, there is a set called `docs:all` that keeps all document names. Another set addressed by the key `availableTags` keeps all tags that were used in the past, which are used for the auto-complete functionality at the client-side.

5.4.1.4. Document Manager

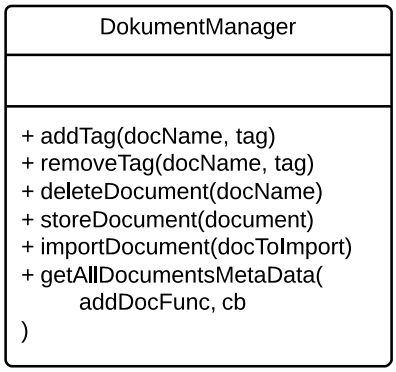


Figure 5.8.: The Document Manager

The `DocumentManager` realises multiple functionalities related to the management of documents. Figure 5.8 shows the respective class diagram. In general, all the presented methods access the database and persistently store information or read them. The above two ones are responsible to read and write tags, belonging to documents. The third and fourth method enables the server-side, to read or persistently write documents. `importDocument` is a special method for the `TriciaConnector`. It accepts as parameter a JSON-object that was created by Tricia for delivering all available documents. This method translates Tricia's representation of documents into the representation, the collaborative editor uses. The last method `getAllDocumentsMetaData` is needed for distributing the meta-data of the documents to newly connected clients. To do so, it gets two functions passed as arguments. The first one broadcasts one single doc to the

clients by emitting the `'newDoc'` event. The second one is a callback-function that is called, if all documents are sent.

5.4.1.5. Tricia Connector

The `TriciaConnector` realizes the connection to Tricia. At this stage of development, it fetches all documents within Tricia right after the start of the collaborative editor. For its implementation, it makes use of the `AllDocumentHandler` of Tricia, described in Section 5.3, that returns all available documents within Tricia in the JSON Format. Therefore, the `TriciaConnector` just needs to make a call to `/node/allDocuments`, parses the textual JSON response to build JavaScript objects and forwards them to the `DocumentManager`, described in Section 5.4.1.5. The respective code to do so can be seen in Listing 5.19.

Listing 5.19: Access Tricia to fetch all documents

```
1  var requestHeader = {
2    host: 'localhost',
3    port: 8084,
4    method: 'GET',
5    path : '/node/allDocuments'
6  }
7
8  http.request(requestHeader, function(res) {
9    res.setEncoding('utf8');
10   res.on('data', function (chunk) {
11     callback(JSON.parse(chunk).pages);
12    });
13 }).end();
```

Line one to six build an object that represents the header for the request that will be done. It specifies the target host, port, HTTP-method and URL. Afterwards, a normal HTTP-request is done, including the currently created header-object and a anonymous callback function that is called, right after the data is available with the response as parameter. In line ten, a listener for the "data-event" is registered, that will be called if the response-body of the HTTP-request arrives. This gets parsed to a JSON-object and passed to the

[121] cf. [Ge12d]

callback-function, which is a function of the `DocumentManager` to import the newly arrived documents.

5.4.1.6. Review Manager

The `ReviewManager` is responsible for creating reviews on demand. To do so, it exposes the one method `getReview(docName, cb)` whereas `docName` is the name of the document, the review is created for. `cb` is the callback function that is called with the review as parameter.
For creating a review, the method reads all operations for the respective document out of the database and groups them by the users into an array. This one again is put into another array, that has one field for each user, therefore, the index of this field is exactly the version, the user is able to choose at the client. The respective document is built afterwards at the client out of the transmitted operations, depending on the chosen version.

5.4.1.7. Color Manager

The `ColorManager` exposes three functions: `getColorPalette()`, `getColor(name)` and `changeColor(name, color)`. The first one returns the color palette in form of an array of valid color values. These ones are readable by the "Spectrum"-library used by the client for choosing colors. The second method defines a unique color for each user initially, whereas `name` is the respective name of the user. The last method is used, if a user wants to choose her color, whereas the name of the user and the chosen color is passed as parameter.

5.5. Conclusion

This chapter presented a prototypical implementation of a collaborative editor. It followed the design decisions, made in Chapter 4 and was realized with JavaScript at the client- and the server-side.
The client-side was developed, following the Model-View-Controller pattern with the Ember.js framework. Its core concepts of models, views, controllers,

templates and the router supported the implementation of the realtime architecture. All changes, done to the models, are automatically propagated to the views, which makes the application of received events easy and directly visible to the user. Furthermore, it made the integration of Twitter Bootstrap straightforward. Nevertheless, from a developer perspective, the working with Ember.js was not comfortable. Its documentation is only rudimentary and its API totally changed during the development of the collaborative editor, including fundamental concepts.
When developing a web application with JavaScript, there are commonly a lot of third-party libraries included for the realization of special tasks. This leads to a lot of dependencies between those libraries, which becomes more and more complex with an increasing number of libraries. The framework RequireJS provided an elegant and efficient way, to manage all these libraries. It offers the building of modules and definition of the respective dependencies in a comfortable way. Additionally, it is able to include libraries, that were not intentionally made for the usage with RequireJS.
The editor component of the collaborative editor tool was realized with TinyMCE. It offers a wide variety of functionalities that enable rich text editing. To enable realtime collaboration, all changes to a document done by one writer, are automatically propagated to other users that currently look at the same document. To implement this feature, an own module was developed that builds a bridge between TinyMCE and the server. To access the content within TinyMCE, it provides its own API. It is well-documented and has a rich set of features. It turned out, that when using advanced features via TinyMCE's API, there is still some browser specific behavior, which complicates the development of the client-side. This can also be seen for the implementation of the editor overview. It displays the whole document in a smaller area, to provide an overview of the document and its current changes to the user. Its implementation needed specific definitions, for each major browser.
The client is delivered initially by Tricia. This situation made it possible, to make use of features of it, like the user management, authorization and authentication. Tricia is a complex application and it needs some induction, to develop with it.

The server was fully developed with Node.js. Its event-driven nature supported the implementation of the realtime architecture. Node.js' rich set of core functionalities made the implementation of the server-side comfortable

and straightforward. For the realization of the collaborative editor the wide variety of different packages was very useful. With socket.io, Node.js provides a powerful way for distributing events to connected clients. It abstracts the underlying technologies, which makes it comfortable to use. Share.js, the second major Node.js library that was used, also provides rich set of features for collaborative editing, that was used for implementing the editor.
During the implementation, one major library needed to be replaced as it was not supported anymore by their developers. Therefore, the selection of third party libraries need to be done wisely. It is also important to realize, that a lot of libraries are developed by single persons in their private time. This makes them less usable in an enterprise environment.

6. Enhancing Collaboration

In the recent sections collaboration was enabled via the instant distribution of events through the whole system. Furthermore, a collaborative editor was presented, that enables users to follow the changes done by one single editor in realtime. Nevertheless, the collaboration can be enabled for more complex scenarios and models.

The prototype presented in Section 4 and Section 5 was based on operational transformation for having easy mechanisms for sharing changes on the document throughout the whole system and to easily save states of the document for further versioning. This approach works good for rich text when there is only one user writing at the same time. In fact, it works when there are no conflicts that need to be resolved or, in an OT-context, that need to be transformed. The normal operational transformation algorithm presented in Section 2.1.2 can enable reactive writing on a linear sequence of data, concretely plain text. It cannot be applied to structured content like HTML. Its application might lead to unbalanced tags and therefore semantically incorrect code. [Go08] even states that "HTML makes OT (Operational Transforms) difficult if not impossible".

This section describes a mechanism to enable reactive writing of rich text and introduces approaches to work collaboratively on arbitrary models. To do so, Section 6.1 describes Google Wave Operational Transformation, Section 6.2 outlines a method to work collaboratively on arbitrary models.

6.1. Google Wave Operational Transformation

Google Wave allows reactive writing on rich text. Its core technology can even be used to collaboratively work on any structured data. To do so, Google invented their own concurrency control system, which is based on operational

transformation[122] (see Section 2.1.2).
In Goole Wave, a document consists of multiple items. Such an item is a start or end tag or a single character. The gaps between these items are called positions. Figure 6.1 shows an exemplary document with the respective items and positions.

Figure 6.1.: Google Wave Items[123]

This is a major difference to the concept presented in Section 2.1.2, where all operations relate to single characters. In Google Wave, the reference unit for operations are items. In Google Wave, operations additionally use annotations that are applied to a sequence of items. These annotations contain metadata, which is actually used for text formatting[124]. A single operation in Google Wave consists of multiple actions or so called operation components. Therefore, an operation is not only a deletion or insertion of one character but instead, can change the whole document at multiple parts. The most important operation components are presented in Table 6.1.

[122] cf. [WML10]
[123] cf. [WML10]
[124] cf. [WML10]

Operation component	Description
insertCharacters	Inserts the specified string at the current index
deleteCharacters	Deletes the specified string from the current index
openElement	Creates a new XML open-tag at the current index
deleteOpenElement	Deletes the specified XML open-tag from the current index
closeElement	Closes the first currently-open tag at the current index
deleteCloseElement	Deletes the XML close-tag at the current index
annotationBoundary	Defines the changes to any annotations (starting or ending) at the current index
retain	Advances the index a specified number of items

Table 6.1.: Most important operation components in Google Wave[125]

An operation is applied in a sequence of operation components that relate to positions within the whole document, therefore, the sequential execution of the components is important. For example lets assume, there is the text `<body>Test message</body>` and we want to capitalize the 'm'. The respective operation can be seen in Listing 6.1.

Listing 6.1: Exemplary Google Wave Operation

```
1 retain(7)
2 deleteCharacters('m')
3 insertCharacters('M')
4 retain(7)
```

First, the cursor is moved to the position seven. Afterwards, the character "m" gets deleted and "M" gets inserted. At last, the rest of the document, which consists of seven items, is kept as it is. This changes the "m" to an "M". It can be seen, that an operation always covers the whole document and not just parts of it[126].

[125] cf. [Sp10]
[125] cf. [Ge12d]
[126] cf. [Sp10]

The remaining functionalities of Google Wave Operational Transformation are similar to these ones presented in Section 2.1.2, albeit all of them are more complex. The main difference between this approach and the classic approach of OT, presented in Section 2.1.2 is the reference unit of "items" instead of positions within the plain text, including the new operations presented in Table 6.1. This enables Google Wave OT to handle structure within a document in contrast to the classic approach. Nevertheless, this reference unit and the additional operations lead to much more complex transformation algorithms[127].

6.2. Collaborative Editing of arbitrary Models

It was shown, that the Google Wave Operational Transformation can enable reactive writing on rich text. Nevertheless, there are more complex models than formatted text where this algorithm does not meet the requirements for realtime collaboration. If there are for example two persons working collaboratively on the simple text "Hello World" and one decides to delete the character "d" and the other one wants to make it emphasized, there is a conflict Operational Transformation cannot resolve via transformation. When applying OT in general, the result might be, depending on the concrete algorithm, "Hello Worl". So, the last character was deleted. This can be seen as acceptable for text, as at the one side, this was the intention of one editor anyway and on the other side, the previous state can be easily recovered, if the deletion was not the intention of the whole group of concurrent editors. Nevertheless, these assumptions may not apply for any complex model.

[Br09] analyzed conflict resolution for models related to model driven engineering. In this context, the models are used for automatic code generation. Conflict resolution methods are an important step towards reactive writing as they build its basis. Joseph Gentle, the developer of share.js which is used within the collaborative editor prototype said for reason, that "Operational Transformation is like realtime git"[128]. [Br09] stated, that standard version control systems, just as Git, are not sufficient for models. They normally work via text comparison. If they are applied to the textual representation of a model, important information get lost, which makes them no valuable option

[127] cf. [Sp10]
[128] cf. [Ge11]

for models. Therefore, special version control systems for specific models has been developed, whereas all follow the approach, that in case of a conflict, one person has to merge the two versions manually to resolve the conflict.[129]
This can be problematic as the one person, doing the conflict resolution, might not reflect the intention of the whole team, which may leads to a wrong result or even to a potential risk of losing important data. In fact, there might be overlapping intentions through the whole collaborators[130]. Furthermore, especially in an early-phase of a project, there is no common understanding about the system, yet, which leads to different views of different collaborators[131].

To solve this problem, [Wi11] proposed a conflict-tolerant method to merge different versions of models. They especially took into account, that developers do not want to interrupt their work for manual conflict resolution. Instead, they want to submit their changes immediately and take care of the resolving later. The proposed approach therefore enables the developers to submit their changes at any time, even if there are conflicting states. Furthermore, it does not force the developers to resolve the conflict immediately. Instead, conflicts gets annotated, which represents a marker for later resolution, which is done during the consolidation phase.
During a consolidation phase, all conflicts within the repository are tried to be resolved collaboratively to get a common stable version of the model. To do so, [Wi11] developed a structural model about this procedure. If there is a conflicting version, there are basically two possibilities to resolve them. One way is to choose one concrete version, another way is to reject all versions and to develop a new one in a collaborative way, which is then accepted. It may also happen, that a conflict depends on another one. This needs to be handled on the concrete example[132].

As described above, conflicts arise especially in an early-project phase when there is no common view and understanding of the system, yet. Another positive result of the collaborative consolidation is to produce this common understanding of the system. Here, especially these project members has to find a solution, that produced the conflict and therefore had a different understanding of the system.

[129] cf. [Br09, p. 3]
[130] cf. [Br09, p. 2 - 3]
[131] cf. [Wi11, p. 2]
[132] cf. [Wi11, p. 18, 29-31]

6.3. Conclusion

Google Wave Operational Transformation is a valuable way for enabling reactive writing on rich text. Albeit, this leads to complex transformation algorithms and therefore a high complexity. Google itself does not improve the wave project any more. Instead, they handed it over to the open-source community. Now, it is an Apache project, whereas it "is still in its infancy" and there is currently no stable version available of it[133].

The approach presented in Section 6.2 is quite interesting for enabling reactive writing on arbitrary models. If conflicts occur, that can not be automatically resolved, an annotation is created, whereas the conflicting versions are kept and the resolving is postponed. Nevertheless, this is still an ongoing research topic, which needs further research.

[133] cf. [Th12]

7. Conclusion and Outlook

The goal of this Thesis was to analyze how realtime collaborative web applications can be enabled. Thereby, a special focus was on the assessment of server-side JavaScript for this scenario.

The survey, conducted within this Thesis showed a high satisfaction with JavaScript in general. Participants are contented with its syntax, tool support and maintainability. They additionally stated, that JavaScript is very important for their projects and that its general importance will rise in future. This builds a good basis for the future of both, client- and server-side JavaScript. Node.js, the most prominent candidate of server-side JavaScript, is already widely known whereas projects that are using Node.js tend to be smaller and younger. The survey showed, that Node.js is typically used for web applications. Furthermore, the majority of the participants confirmed Node.js' enterprise readiness.
On the contrary, the selection of third-party libraries on top of Node.js must be deliberate. Even though, they can be easily installed with Nodes' own package manager, a big amount of libraries and frameworks is developed by single persons in their private time and are immature, yet. This makes them to a potential risk within projects, especially in enterprise environments.

The realtime architecture, presented within this Thesis, is a good pattern for enabling realtime collaboration in web applications. It allows an easy way of sending events through the system and making the impact of them applicable at the client-side. As this architecture is fully event-driven, Node.js is a good fit for implementing the respective server-side. Especially in combination with socket.io, Node.js can enable realtime communication easily, which was shown with the implementation of the collaborative editor prototype.

Nevertheless, there are still open topics, that need to be evaluated in further research. There were some approaches presented within this Thesis, for collab-

orative editing of rich text and arbitrary models. The Google Wave Protocol is a valuable way for realizing reactive writing on rich text. Unfortunately, the algorithms behind are quite complex and would lead to an immense implementation effort, when it should be used in practice. Collaborative editing of arbitrary models is still an open topic. Although, the approach presented within this Thesis is quite interesting for enabling collaboration, there is still a manual merge to be done.
Furthermore, the integration of the collaborative editor into Tricia can be enhanced in future. Especially, realtime collaboration functionalities like the chat or the displaying of online users could be integrated into Tricia, to enhance its collaboration features.

Bibliography

[Ac13] ActiveState Software Inc.: *Komodo IDE 7: The Professional IDE for Python, PHP, Ruby and Perl | ActiveState.* `http://www.activestate.com/komodo-ide`. 2013. [Online; accessed 24.01.2013].

[Ad12] Adobe Systems Inc.: *Adobe Flash Professional CS6.* `http://www.adobe.com/de/products/flash.html`. 2012. [Online; accessed 12.12.2012].

[BMN10] Büchner, T.; Matthes, F.; Neubert, C.: *Data model driven implementation of web cooperation systems with Tricia.* In *Objects and Databases*. pages 70–84. Springer. 2010.

[Br09] Brosch, P.; Seidl, M.; Wieland, K.; Wimmer, M.; Langer, P.: *We can work it out: Collaborative conflict resolution in model versioning.* In *ECSCW 2009*. pages 207–214. Springer. 2009.

[Br10] Braunstein, R.: *ActionScript 3.0 Bible.* Wiley. 2 edition. 2010.

[Bu10] Bundesverband Informationswirtschaft, Telekommunikation und neue Medien e.V.: *Studie „Connected Worlds“ - Wie Lebens- und Technikwelten zusammenwachsen.* `http://www.bitkom.org/files/documents/bitkom_connected_worlds_extranet.pdf`. 2010. [Online; accessed 10.05.2013].

[Bu11a] Bundesverband Informationswirtschaft, Telekommunikation und neue Medien e.V.: *Soziale Netzwerke - Eine repräsentative Untersuchung zur Nutzung sozialer Netzwerke im Internet.* `http://www.bitkom.org/files/documents/sozialenetzwerke.pdf`. 2011. [Online; accessed 10.05.2013].

[Bu11b] Burke, J.: *RequireJS.* `http://requirejs.org/`. 2011. [Online; accessed 02.04.2013].

[Bu12] Burke, J.: *AMD*. `https://github.com/amdjs/amdjs-api/wiki/AMD`. 2012. [Online; accessed 02.04.2013].

[Ch12] Chow, J.: *Continuous Integration for Mobile*. `http://engineering.linkedin.com/testing/continuous-integration-mobile`. 2012. [Online; accessed 24.01.2013].

[Cl12] Cloud9ide: *Cloud9 IDE | Online IDE – Your code anywhere, anytime*. `https://c9.io/`. 2012. [Online; accessed 09.04.2013].

[Co13] Code Together, L.: *Squad: Collaborative Code Editor*. `https://squadedit.com/`. 2013. [Online; accessed 09.04.2013].

[Da11] Dahl, R.: *Ryan Dahl - History of Node.js*. `http://youtu.be/SAc0vQCC6UQ`. 2011. [Online; accessed 10.05.2013].

[EB11] EBay Inc.: *Announcing ql.io*. `http://www.ebaytechblog.com/2011/11/30/announcing-ql-io/`. 2011. [Online; accessed 24.01.2013].

[EG89] Ellis, C. A.; Gibbs, S. J.: *Concurrency control in groupware systems*. In *ACM SIGMOD Record*. volume 18. pages 399–407. ACM. 1989.

[Fi12] Ficarra, M.: *CoffeeScript*. `http://coffeescript.org/`. 2012. [Online; accessed 04.02.2012].

[Fl11] Flanagan, D.: *JavaScript: The Definitive Guide: Activate Your Web Pages (Definitive Guides)*. O'Reilly Media. 2011.

[FR11] Fernandez-Ruiz, B.: *Yahoo! Announces Cocktails – Shaken, Not Stirred*. `http://developer.yahoo.com/blogs/ydn/posts/2011/11/yahoo-announces-cocktails-?-shaken-not-stirred/`. 2011. [Online; accessed 24.01.2013].

[Fr13] Free Software Foundation Inc.: *GNU Emacs*. `http://www.gnu.org/software/emacs/`. 2013. [Online; accessed 24.01.2013].

[FRSF10] Fraternali, P.; Rossi, G.; Sánchez-Figueroa, F.: *Rich Internet Applications. IEEE Internet Computing*. 14(3):9–12. 2010.

[Ga77] Gannon, J.: *An experimental evaluation of data type conventions. Communications of the ACM*. 20(8):584–595. 1977.

[Ga13] Gartner Inc.: *IT Market Clock for Programming Languages, 2013*. 2013.

[Ge11] Gentle, J.: *ShareJS – Live concurrent editing in your app*. `http://sharejs.org/`. 2011. [Online; accessed 21.03.2013].

[Ge12a] Gentle, J.: *Client API*. 2012. [Online; accessed 21.03.2013].

[Ge12b] Gentle, J.: *josephg/node-browserchannel*. `https://github.com/josephg/node-browserchannel`. 2012. [Online; accessed 21.03.2013].

[Ge12c] Gentle, J.: *JSON Client API*. `https://github.com/josephg/ShareJS/wiki/JSON-Client-API`. 2012. [Online; accessed 23.03.2013].

[Ge12d] Gentle, J.: *Wire Protocol*. `https://github.com/josephg/ShareJS/wiki/Wire-Protocol`. 2012. [Online; accessed 21.03.2013].

[Ge13] Gentle, J.: *ShareJS/src/client/textarea.coffee at master - josephgShareJS - GitHub*. `https://github.com/josephg/ShareJS/blob/master/src/client/textarea.coffee`. 2013. [Online; accessed 05.04.2013].

[Gi13] GitHub Inc.: *Top Languages*. `https://github.com/languages`. 2013. [Online; accessed 22.01.2013].

[Go08] Google Inc.: *FAQ - Google Wave Federation Protobol*. `http://www.waveprotocol.org/faq#TOC-What-s-the-XML-schema-for-waves-Why-isn-t-it-HTML5-`. 2008. [Online; accessed 18.04.2013].

[Go13] Google: *Welcome to Google Apps*. `http://www.google.com/apps/index1.html`. 2013. [Online; accessed 11.05.2013].

[GT02] Gourley, D.; Totty, B.: *HTTP: The Definitive Guide*. O'Reilly Media. 2002.

[Ha10a] Hanenberg, S.: *An experiment about static and dynamic type systems: Doubts about the positive impact of static type systems on development time. ACM Sigplan Notices*. pages 22–35. 2010.

[Ha10b] Hanenberg, S.: *Doubts about the positive impact of static type systems on programming tasks in single developer projects-an empirical study. ECOOP 2010–Object-Oriented Programming*. pages 300–303. 2010.

[HB11] HBR Labs, L.: *Web Meeting and Document Sharing | ShowDocument*. `http://www.showdocument.com/`. 2011. [Online; accessed 09.04.2013].

[Ho12] Holowaychuk, T.: *Express - node.js web application framework*. `http://expressjs.com/`. 2012. [Online; accessed 21.12.2012].

[in12] infoAsset AG: *infoAsset : Home*. `http://www.infoasset.de`. 2012. [Online; accessed 13.11.2012].

[Is11] Isaac Z., Schlueter: *Node.js Digs Dirt - about Data-Intensive Real-Time Applications*. `http://video.nextconf.eu/video/1914374/nodejs-digs-dirt-about`. 2011. [Online; accessed 10.05.2013].

[Je13a] JetBrains: *IntelliJ IDEA :: Features*. `http://www.jetbrains.com/idea/features/index.html`. 2013. [Online; accessed 14.05.2012].

[Je13b] JetBrains: *The best JavaScript IDE with HTML Editor for Web development :: JetBrains WebStorm*. `http://www.jetbrains.com/webstorm/`. 2013. [Online; accessed 14.05.2012].

[Jo13a] Joyent Inc.: *High-performance Cloud Infrastructure for Real-time Web and Mobile Applications - Home - Joyent*. `http://joyent.com/`. 2013. [Online; accessed 04.03.2013].

[Jo13b] Joyent Inc.: *Node.js v0.8.21 Manual & Documentation*. `http://nodejs.org/api/http.html#http_http_createserver_requestlistener`. 2013. [Online; accessed 04.03.2013].

[Ka13] Kaazing: *About HTML5 WebSockets*. `http://www.websocket.org/aboutwebsocket.html`. 2013. [Online; accessed 14.05.2013].

[Kl12] Kleinschmager, S.; Hanenberg, S.; Robbes, R.; Tanter, É.; Stefik, A.: *Do static type systems improve the maintainability of software systems? An empirical study*. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*. pages 153–162. IEEE. 2012.

[Le13] LearnBoost Labs: *Socket.IO: the cross-browser WebSocket for realtime apps - Supported transports*. `http://socket.io/#browser-support`. 2013. [Online; accessed 04.03.2013].

[Lo04] Lowry, Paul Benjamin, Curtis, Aaron Mosiah and Lowry, M. R.: *A Taxonomy of Collaborative Writing to Improve Empirical Research, Writing Practice, and Tool Development. Journal of Business Communication (JBC).* 41:66–99. 2004.

[Ma11] MacCaw, A.: *JavaScript Web Applications.* O'Reilly Media. 2011.

[MC08] McCarthy, P.; Crane, D.: *Comet and Reverse Ajax: The Next-Generation Ajax 2.0.* 2008.

[Mi12] Microsoft: *TypeScript preview.* `http://www.typescriptlang.org/`. 2012. [Online; accessed 24.01.2013].

[Mi13a] Microsoft: *WebMatrix 2.* `http://www.microsoft.com/web/webmatrix/`. 2013. [Online; accessed 24.01.2013].

[Mi13b] Microsoft: *Windows Azure Node.js Dev Center.* `http://www.windowsazure.com/en-us/develop/nodejs/`. 2013. [Online; accessed 24.01.2013].

[MNS11] Matthes, F.; Neubert, C.; Steinhoff, A.: *Hybrid wikis: Empowering users to collaboratively structure information.* In *6th International Conference on Software and Data Technologies (ICSOFT)*. pages 250–259. 2011.

[Mo12] Moxiecode Systems AB.: *TinyMCE - Home.* `http://www.tinymce.com/`. 2012. [Online; accessed 21.10.2012].

[Mo13] Mozilla Developer Network and individual contributors: *HTTP access control (CORS).* `https://developer.mozilla.org/en-US/docs/HTTP/Access_control_CORS`. 2013. [Online; accessed 08.04.2013].

[MT07] Mikkonen, T.; Taivalsaari, A.: *Using JavaScript as a Real Programming Language Using JavaScript as a Real Programming Language.* 2007.

[No12] Nodejitsu Inc.: *Flatiron, A framework for Node.js.* `http://flatironjs.org/`. 2012. [Online; accessed 04.11.2012].

[O'12] O'Grady, S.: *The RedMonk Programming Language Rankings: September 2012 – tecosystems.* `http://redmonk.com/sogrady/2012/09/12/language-rankings-9-12/`. 2012. [Online; accessed 04.02.2012].

[PL11] Peter Lubbers, Brian Albers, F. S.: *Pro HTML5 Programming*. Apress. 2 edition. 2011.

[Po07] Powers, S.: *Adding Ajax*. O'Reilly Media. 2007.

[PT98] Prechelt, L.; Tichy, W. F.: *A controlled experiment to assess the benefits of procedure argument type checking. Software Engineering, IEEE Transactions on*. 24(4):302–312. 1998.

[Ra12a] Rauch, G.: *LearnBoost/socket.io-spec*. `https://github.com/LearnBoost/socket.io-spec`. 2012. [Online; accessed 04.03.2013].

[Ra12b] Rauch, G.: *Socket.IO: the cross-browser WebSocket for realtime apps*. `http://socket.io/`. 2012. [Online; accessed 04.03.2013].

[Ri10] Richards, G.; Lebresne, S.; Burg, B.; Vitek, J.: *An analysis of the dynamic behavior of JavaScript programs*. In *ACM Sigplan Notices*. volume 45. pages 1–12. ACM. 2010.

[Ro10] Roden, T.: *Building the realtime user experience*. O'Reilly Media. 2010.

[SA09] Souders, S.; Almaer, D.: *Even faster web sites*. O'Reilly. Beijing ; Sebastopol, CA. 1st edition. 2009.

[Se12] Sencha Labs: *Connect - High quality middleware for node.js*. `http://www.senchalabs.org/connect/`. 2012. [Online; accessed 13.11.2012].

[SH11] Stuchlik, A.; Hanenberg, S.: *Static vs. dynamic type systems: an empirical study about the relationship between type casts and development time*. In *ACM SIGPLAN Notices*. volume 47. pages 97–106. ACM. 2011.

[Sh13] ShareLaTeX: *LaTeX Editor ShareLaTeX - ShareLaTeX.com*. `https://www.sharelatex.com/`. 2013. [Online; accessed 10.05.2013].

[Sp06] Spinellis, D.: *Code Quality: The Open Source Perspective*. Addison-Wesley Professional. 2006.

[Sp10] Spiewak, D.: *Understanding and Applying Operational Transformation*. `http://www.codecommit.com/blog/java/understanding-and-applying-operational-transformation`. 2010. [Online; accessed 20.04.2013].

[Su04] Sun, D.; Xia, S.; Sun, C.; Chen, D.: *Operational transformation for collaborative word processing. Proceedings of the 2004 ACM conference on Computer supported cooperative work - CSCW '04*. 6(3):437. 2004.

[Su11] Sun, C.: *OT FAQ - Operational Transformation Frequently Asked Questions and Answers*. `http://cooffice.ntu.edu.sg/otfaq/index.php`. 2011. [Online; accessed 24.11.2012].

[SWF12] Sun, C.; Wen, H.; Fan, H.: *Operational transformation for orthogonal conflict resolution in real-time collaborative 2d editing systems. Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work - CSCW '12*. page 1391. 2012.

[Te12] Teixeira, P.: *Professional Node.js: Building Javascript Based Scalable Software*. John Wiley & Sons. 1st edition. 2012.

[Th99] The Internet Engineering Task Force: *RFC 2616 - Hypertext Transfer Protocol – HTTP/1.1*. `http://www.ietf.org/rfc/rfc2616.txt`. 1999. [Online; accessed 10.12.2012].

[Th12] The Apache Software Foundation: *Apache Wave - Welcome to Apache Wave (Incubating)*. `http://incubator.apache.org/wave/`. 2012. [Online; accessed 12.05.2013].

[Th13] The Etherpad Foundation: *Etherpad lite*. `http://etherpad.org/`. 2013. [Online; accessed 09.04.2013].

[Ti13a] Tilde Inc.: *Ember.js - Concepts*. `http://emberjs.com/guides/concepts/core-concepts/`. 2013. [Online; accessed 03.04.2013].

[Ti13b] Tilde Inc.: *Ember.js - Ember*. `http://emberjs.com/api/`. 2013. [Online; accessed 04.04.2013].

[Tw13] Twitter, I.: *Bootstrap*. `http://twitter.github.com/bootstrap/`. 2013. [Online; accessed 03.04.2013].

[VM03] VMware, Inc: *Redis*. `http://redis.io/`. 2003. [Online; accessed 27.02.2013].

[W3] W3C: *XMLHttpRequest - W3C Working Draft 6 December 2012*. `http://www.w3.org/TR/XMLHttpRequest/`. [Online; accessed 11.12.2012].

[W312] W3C: *Share: News and Opinions about HTML*. `http://www.w3.org/html/`. 2012. [Online; accessed 11.12.2012].

[Wa11] Warden, P.: *Big Data Glossary*. O'Reilly Media. 2011.

[Wi11] Wieland, K.; Langer, P.; Seidl, M.; Wimmer, M.; Kappel, G.: *Turning Conflicts into Collaboration. Computer Supported Cooperative Work (CSCW)*. pages 1–60. 2011.

[WML10] Wang, D.; Mah, A.; Lassen, S.: *Google Wave Operational Transformation*. `http://wave-protocol.googlecode.com/hg/whitepapers/operational-transform/operational-transform.html`. 2010. [Online; accessed 18.04.2013].

[Zo13] Zoho Corporation Pvt. Ltd.: *More than 7 Million users Work Online with Zoho*. `https://www.zoho.com/`. 2013.

A. Survey

A.1. Survey Question

The survey is divided into three parts. The first one asks JavaScript specific questions, the second one deals with Node.js, whereas the last part addresses the organization, the participant is working in. The questions of all three parts are presented in the following sections.

A.1.1. Java Script

1. **Are you using or have you ever been in contact to JavaScript ? (multiple answers allowed)**

   - ○ I use JavaScript in a private project
   - ○ I work in a project, where JavaScript is used
   - ○ I actively develop with JavaScript
   - ○ I've used JavaScript in the past
   - ○ I've never used JavaScript and have no experience with it
   - ○ Other: _____

2. **What programming languages are mostly used in your projects despite of JavaScript ? (multiple answers allowed)**

   - ○ C
   - ○ C++
   - ○ C#
   - ○ Java
   - ○ PHP
   - ○ HTML
   - ○ Python
   - ○ Ruby
   - ○ Perl
   - ○ Objective C
   - ○ Scala
   - ○ Haskell
   - ○ Lua
   - ○ Erlang
   - ○ Prolog
   - ○ Visual Basic
   - ○ ASP
   - ○ Go
   - ○ Shell
   - ○ Power Shell
   - ○ Other: _____

3. How much of your code is written in JavaScript (measured in lines of code) in your projects in average ?

○ 0%	○ 31% - 40%	○ 71% - 80%
○ 1% - 10%	○ 41% - 50%	○ 81% - 90%
○ 11% - 20%	○ 51% - 60%	○ 91% - 99%
○ 21% - 30%	○ 61% - 70%	○ 100%

4. Which editors/IDEs do you use to program in JavaScript ?

○ Suiblime Text	○ NetBeans	○ IntelliJ
○ Vim	○ Notepad++	
○ Emacs	○ Visual Studio	○ Aptana
○ TextMate	○ WebStorm	
○ Eclipse	○ Komodo	○ Other: _____

5. Are you fully satisfied with the features of your editors/IDEs related to JavaScript programming ?

○ Yes ○ No ○ No answer

6. Which features are you missing in your editor/IDE related to JavaScript programming ?

_____

7. Are you using any testing-frameworks for your JavaScript Code ?

○ Yes ○ No ○ No answer

8. Which testing-frameworks do you use for your JavaScript Code ?

○ QUnit	○ Buster.js	○ Sinon.JS
○ Jasmine	○ TestSwarm	○ Chutzpah
○ JSUnit	○ JsTestDriver	
○ RhinoUnit	○ Yeti	○ Other: _____

9. **What other testing-frameworks do you use ?**

Please evaluate the following statements about JavaScript and state whether you agree or disagree.

10. **JavaScript is important for my projects**
    strongly disagree ○—○—○—○ strongly agree No answer ○

11. **The general importance of JavaScript will rise in future**
    strongly agree ○—○—○—○ strongly disagree No answer ○

12. **JavaScript code is easy to structure**
    strongly agree ○—○—○—○ strongly disagree No answer ○

13. **JavaScript code is hard to maintain**
    strongly agree ○—○—○—○ strongly disagree No answer ○

14. **JavaScript code is hard to refactor**
    strongly agree ○—○—○—○ strongly disagree No answer ○

15. **JavaScript code is easy to read**
    strongly agree ○—○—○—○ strongly disagree No answer ○

16. **I'm fully satisfied with the performance of JavaScript**
    strongly agree ○—○—○—○ strongly disagree No answer ○

17. **I prefer clientside processing (instead of serverside)**
    strongly agree ○—○—○—○ strongly disagree No answer ○

18. **I like the syntax of JavaScript**
    strongly agree ○—○—○—○ strongly disagree No answer ○

19. **Have you ever heard of CoffeeScript ?**
    ○ Yes ○ No

20. **How much of your JavaScript code is written in CoffeeScript ?**

○ 0%
○ 0% - 49%
○ 50% - 99%
○ 100%

21. What are the main reasons for you using CoffeeScript ?

---

22. What are the main reasons for you to not use CoffeeScript ?

---

23. Are you planning to use CoffeeScript in future ?

○ Yes ○ No ○ No answer

24. What are the main reasons for you to use CoffeeScript in future ?

---

25. Have you ever heard of TypeScript ?

○ Yes ○ No

26. How much of your JavaScript code is written in TypeScript ?

○ 0%
○ 0% - 49%
○ 50% - 99%
○ 100%

27. What are the main reasons for you to use TypeScript ?

---

28. What are the main reasons for you to not use TypeScript ?

---

29. Are you planning to use TypeScript in future ?

○ Yes ○ No ○ No answer

30. Why are you planning to use TypeScript in future ?

---

31. Are you using any JavaScript frameworks or libraries ?

○ Yes ○ No ○ No answer

32. Which JavaScript Frameworks/libraries do you use ?

○ jQuery	○ Backbone.js	○ batman.js
○ Prototype	○ ExtJS	○ YUI
○ Cappucino	○ Angular	○ Ember
○ MooTools	○ Knockout	○ Spine
○ JavaScriptMVC	○ PureMVC	○ Others: _____

33. Which additional JavaScript frameworks/libraries do you use ?

_____

34. If you want to state any further comments or impressions about JavaScript please do so in the following.

_____

A.1.2. Node.js

35. Have you ever heard of Node.js ?

○ Yes ○ No

36. Where did you first hear about Node.js ? Please give a more detailed source in the comment-field.

○ Magazine	○ Friends	○ Conference
○ Internet News	○ Business	○ Others: _____

37. Do you actively develop with Node.js or is it used in one of your projects ?

○ Yes ○ No

38. What is the context of your project ?.

○ Private	○ Open Source Project
○ Business	○ Others: _____

39. When was the project start ?

_____

40. How many people are working in your project ?

_____

41. Whats the name of the Open Source Project ?

_____

42. What kind of project are you developing ?

○ Web Application ○ Command Line Tool ○ Desktop Application

○ Framework ○ Embedded ○ Others: _____

43. Please describe your project in a few words

_____

44. Please upload some screenshots of your project

45. What were the main reasons for choosing Node.js for your project ?

_____

46. Did you ever consider to use Node in one of your projects ?

○ Yes ○ No

47. Why did you decide against Node ?

_____

48. What were the main reasons for not considering Node at all ?

_____

49. Are you planning to use Node in future ?

○ Yes ○ No ○ No answer

50. Do you use any frameworks on top of Node.js ?

○ Yes ○ No

51. Which frameworks do you use on top of Node.js ?

○ Derby ○ TowerJS ○ nowJS ○ Others: _____

○ Meteor ○ Express ○ flatiron

○ SocketStream ○ GeddyJS ○ restify

52. Which frameworks are you using addtitionally on top of Node.js ?

_____

53. Do you use testing frameworks for your Node.js code ?

○ Yes ○ No

54. Which testing frameworks do you use ?

○ Mocha ○ Jasmine ○ Others: _____

○ Vows ○ Expresso

○ should ○ NodeUnit

55. Which additional testing frameworks are you using for your Node.js code ?

_____

Please evaluate the following statements and state whether you agree or disagree

56. Node.js is important for my projects

strongly disagree ○—○—○—○ strongly agree No answer ○

57. The general importance of Node.js will rise in future

strongly agree ○—○—○—○ strongly disagree No answer ○

58. I'm fully satisfied with the performance of Node.js

strongly agree ○—○—○—○ strongly disagree No answer ○

59. Do you think Node.js is suitable for enterprise applications ?

○ Yes ○ No ○ No answer

60. What are the reasons for Node.js being not suitable for enterprise applications in your opinion ?

---

61. What are the reasons for Node.js being suitable for enterprise applications in your opinion ?

---

62. What functionality or module are you missing currently which you wish to be implemented ?

---

63. If you want to state any further comments or impressions about Node.js please do so in the following.

A.1.3. Your Company

64. Whats the name of your organization ?

---

65. What industry does your organization belong to ?

- ○ Aerospace
- ○ Education (e.g. professor, (graduate) student)
- ○ Finance (banks, insurance companies)
- ○ Health
- ○ IT Consulting
- ○ IT Products and Services
- ○ Management Consulting
- ○ Production and manufacturing
- ○ Automotive
- ○ Public service
- ○ Telecommunications
- ○ Transport / Logistics
- ○ Open Source
- ○ Other: _____

66. What kind of project does your organization typically have ?

○ Embedded
○ Cloud Computing
○ Web development
○ Desktop Applications
○ Security Applications
○ Other: _____

67. How many employees work at your organization ?

_____

68. What is your current role within the organization ?

○ Business Architect
○ Consultant
○ CXO
○ Enterprise Architect
○ IT Architect
○ IT Operations
○ Software Engineer
○ Subject Matter Expert
○ Software Tester
○ Project Manager
○ Other: _____

69. In which country do you work currently ?

_____

70. When was your company founded (year) ?

_____

71. What is your current role within the organization ?

○ Scrum
○ V-Modell XT
○ Waterfall Model
○ Other: _____
○ V-Modell
○ Kanban

Please evaluate the following statement and state whether you agree or disagree

72. My organization is an early adopter related to new technologies
strongly disagree ○—○—○—○ strongly agree No answer ○

73. Do you want to be informed about the results of the survey ?
○ Yes ○ No

74. Would you be willing to answer further questions ?
○ Yes ○ No

75. Please provide your contact eMail address

A.2. General Statements about JavaScript

The following presents some selected cites out of the 16 statements:

The biggest thing missing from JavaScript is education. Too many bad programmers think they can use it, but have no clue.

JavaScript is important because it is the only viable option for web-based RIA's, not because it's actually a good language.

I think javascript together with html5 and css3 makes some of the common web development technologies obsolete. Especially device independent development will be more efficient. However in enterprise environment there is a need of more matured frameworks and better integration with emerging technologies like closure.

I have found JavaScript itself to be great, especially when mostly following JsLint and The Good Parts. I'm excited to see where the language is headed with many of the new features that have been proposed. I have found that dealing with browsers is much more of a pain. Many of the standardized APIs are very strange and there are many weird edge cases when different browsers work differently or haven't implemented the same features. Browsers are what made JavaScript what it is, but now they seem like a limitation. JavaScript's bad parts aren't completely getting fixed because no one wants to force people to rewrite their old code.

I think that with the evolution of Javascript on server side, and with the combination of Html5 and CSS3. Javascript is going to be lead programming language of the future.

Once you're getting used to it, you wouldn't use anything else. The asynchronous paradigm is awesome.

A.3. General Statements about Node.js

Do not build monolithic apps with Node, compose services and REST interfaces. Know what 'node debug' does. Also, watch your memory usage and if you actually need to buffer / allocate objects. These are the major pitfalls of people in many languages but Node's adopters seem to find themselves unable to research the solutions to these for some reason.

Node.js is great! There seems to be a ton of people who don't like it and I feel like the controversy might be causing others to decide not to use it. It's fine for people to have their own opinions, I just wish Node.js had a better public image. Most of the problem seems to be because people don't like using callback functions. I understand that for beginners it's not immediately clear the best way to handle non-trivial cases when several asynchronous operations are going on at once or when asynchronous operations need to be chained together, but there are many very simple ways to address these problems. There must be some better way to combat the idea that callbacks are hard.

It is great! Enjoying every minute.

Still in it's infancy, but stable enough to use in production.

Never had a development environment that strong.

After developing with node for nearly two years, anything else I use seems antiquated.

I am thinking to switch my career to : Node.js, html5, css3, Nosql etc

callback style of development is immediate a turn off, you really have to switch the way you think.

Node will likely be a good choice for certain kinds of enterprise applications once it stabilizes, though I'm still not a fan of JavaScript in general.

B. Implementation of the collaborative editor

B.1. Supported browsers by socket.io

Socket.io supports the following browsers:

B.1.1. Desktop Browsers

Browser	**Minimum Version**
Internet Explorer	5.5
Safari	3
Google Chrome	4
Firefox	3
Opera	10.61

Table B.1.: Supported Desktop Browsers by socket.io[134]

B.1.2. Mobile Browsers

Browser
iPhone Safari
iPad Safari
Android WebKit
WebOs WebKit

Table B.2.: Supported Mobile Browsers by socket.io[135]

[134] cf. [Le13]
[135] cf. [Le13]

B.2. Third party libraries used by the client

Name	Homepage	Description
Bootstrap	`http://twitter.github.io/bootstrap/`	Front-end Framework
Bootstrap-dropdown	`http://twitter.github.io/bootstrap/`	Library for Bootstrap drop down menus
Bootstrap-tooltip	`http://twitter.github.io/bootstrap/`	Library for Bootstrap tooltips
jQuery	`http://jquery.com/`	JavaScript library fur multiple functionalities like HTML document traversal and manipulation or event handling
jQuery UI	`http://jqueryui.com/`	Pre-developed user interface elements, effects, widgets or themes
Tag-it	`https://github.com/aehlke/tag-it`	JavaScript library that provides tagging functionalities and animation
Spectrum	`http://bgrins.github.io/spectrum/`	A colorpicker
Share.JS	`http://sharejs.org/`	The client-library for Node.js' live concurrent editing library
Browser-Channel	`https://github.com/josephg/node-browserchannel`	The client-library for using the BrowserChannel server
TinyMCE	`http://www.tinymce.com/`	Javascript WYSIWYG editor with rich set of features
Ember.js	`http://emberjs.com/`	Web-application framework
Handlebars	`http://handlebarsjs.com/`	Templating framework
socket.io	`http://socket.io/`	Library that enables realtime communication with the server

Table B.3.: Third party libraries used by the client