



UNIVERSITY OF INNSBRUCK

MASTERTHESIS

---

# Social Weaver

A Platform for Weaving Web 2.0 Features into Web-based Applications

---

*Author:*  
Viktor PEKAR

*Supervisor:*  
Dr. Michael FELDERER

*Collaboration:*  
Dr. Dirk DRAHEIM

July 19, 2013

# Contents

<b>1</b>	<b>Introduction</b>	<b>6</b>
<b>2</b>	<b>Contribution</b>	<b>7</b>
<b>3</b>	<b>Social Weaver - Abstract Domain Level</b>	<b>10</b>
3.1	The WHY-WHAT-WHO-Model . . . . .	10
3.2	Domain Analysis . . . . .	12
3.3	Problem Description . . . . .	12
3.4	What is Social Weaver . . . . .	12
3.5	Requirements for Social Weaver . . . . .	13
3.5.1	Browser Plugin . . . . .	14
3.5.2	Server Application . . . . .	15
3.5.3	Social Weaver - Script Support . . . . .	16
<b>4</b>	<b>Social Weaver - Concrete Domain Level</b>	<b>19</b>
4.1	Social Weaver - Firefox Plugin . . . . .	19
4.2	Requirements for the Plugin . . . . .	24
4.2.1	Display Management Requirement . . . . .	24
4.2.2	Managing several comment boxes without disturbing the view on original content . . . . .	27
4.2.3	Communication to server application . . . . .	29
4.3	Social Weaver - Web Service . . . . .	31
4.4	Used Technologies . . . . .	31
4.5	Web Service Architecture . . . . .	31
4.6	Requirements for the Web Service . . . . .	32
4.6.1	Offer service that receives messages from plugin-clients . .	33
4.6.2	Synchronization for requests from different user-session .	33
4.6.3	Persist updates into a database . . . . .	33
4.6.4	Keep the server application independent to weaved-into web application . . . . .	33
4.6.5	Parse incoming messages . . . . .	33
4.6.6	Create outgoing messages . . . . .	33
4.7	Social Weaver - Script Support . . . . .	34
4.8	Requirements for the Script Support . . . . .	35
4.8.1	Container of all necessary information for element matching	35
4.8.2	Decoupled from browser plugin and server backend . . . .	36
4.8.3	Syntax that is easy to read and write . . . . .	36
4.8.4	Extension of the plugin with parsing methods . . . . .	36
4.8.5	Default matching procedure should be provided . . . . .	37
4.8.6	Scripts should be related to a single or a set of URLs . . .	38
4.9	Ambiguity Problem . . . . .	39
4.9.1	Ambiguous Grammar . . . . .	39
4.9.2	Ambiguity for Element Matching . . . . .	39

<b>5</b>	<b>Social Weaver Analysis</b>	<b>42</b>
5.1	Social Weaver in Action . . . . .	42
5.2	Social Weaver Scripts in Action . . . . .	48
5.3	Social Weaver Assessment . . . . .	49
<b>6</b>	<b>Discussion</b>	<b>50</b>
<b>7</b>	<b>Future Work</b>	<b>51</b>
<b>8</b>	<b>Conclusion</b>	<b>51</b>
<b>9</b>	<b>Defintions</b>	<b>52</b>
	<b>Appendices</b>	<b>53</b>
.0.1	Akteure . . . . .	53
.0.2	Use Cases . . . . .	53

## List of Figures

1	Basic idea of Social Weaving . . . . .	8
2	Three dimensions of the requirements types[21] . . . . .	11
3	Social Weaver Module Overview . . . . .	12
4	Social Weaver Prototype Use Case . . . . .	13
5	Workflow for Script Using . . . . .	17
6	An example for a <i>panel</i> that shows a list of anntations . . . . .	20
7	Example for a widget serving as activation button . . . . .	21
8	Partition of the first plugin requirement to sub requirements . . .	25
9	Rectangle shows that the underlying element is possible for an- notation . . . . .	26
10	Rectangle shows that the underlying element is possible for an- notation . . . . .	27
11	Widget representation for different modes . . . . .	28
12	Sample JavaScript code for retrieving JSON objects from a web service with a GET REST request . . . . .	29
13	Screenshot from Social Weaver Persistence Web View . . . . .	32
14	Example for an ambiguous grammar . . . . .	40
15	HTML example for showing ambiguity . . . . .	41
16	DOM tree represenation for HTML code in Figure 4.9.2 . . . . .	41
17	Sequence diagram for a successful plugin update . . . . .	43
18	Sequence diagram for a standard matching procedure . . . . .	44
19	Sequence diagram for standard marking procedure . . . . .	46

## Listings

## **Abstract**

Communication through the internet has been made easy in the last few years. But discussing workflows and functionality of web applications or web pages is still a time consuming task, that requires a lot of explanation. Social Weaving introduces a new concept of communication. Injecting - or how we call it: weaving - social elements like chats, wiki pages and so on directly into the view of your application (without the need of modifying the underlying code). Information becomes directly attached to its relevant position.

This thesis will explain the theory behind Social Weaving and show the prototype: Social Weaver.

# 1 Introduction

This thesis is about Social Weaving. A new technique that combines modern communication methods with existing web sites and web applications. The goal is to create a layer above the existing environment without directly modifying it. When we talk about "modern communication methods" we have social media in mind, like wiki pages, chats, comment boxes, etc. but also support for file upload, appointment invitations to shared calendars and so on. After all it will not matter what exactly is woven into the environment. Since it might be some HTML code, the user has the free choice. What is such an environment that we mentioned above? Informally we define an environment something that is visible through a browser. Now we have large variety of software we see in a browser. We have static and dynamic web pages, web applications using Flash or Java, and a lot of another technologies. The best case for Social Weaving would be to support seamlessly everything - but as we will see this is not possible.

The great number of standards for the web does not prevent that every web page is constructed in a different way. There are no unique identifiers for elements, which would be necessary to guarantee a full Social Weaving support. Even though we cannot change the structures used in the web, we want to show what is possible with Social Weaving even now.

In the section Contribution - 2 we will discuss in detail how the basic idea of Social Weaving works and what problems it solves. The rest of this paper is about Social Weaver - a prototype for Firefox that shows a basic functionality for certain environments. So the second part (??) is an abstract requirement analysis that describes what a Social Weaving system needs. The third part (section 4) shows the architecture of the prototype on a more concrete level.

## 2 Contribution

In the last couple of years the internet developed into a mass medium. It started with the simple asynchronous one-to-one communication such as E-mail. Today we have all kinds of communication types: forums and bulletin boards support many-to-many information exchange, one-to-many is being provided by services like Twitter. Chats or instant messages give us the possibility of synchronous transmissions. With the launch of webcams the internet took regular voice calls to another level adding the opportunity to actually see each other. Success stories like Facebook and Twitter show us, that human communication is still in development. The problem is that we see communication as something we were practicing since we exist. Two persons standing in front of each other and using spoken language. But the internet offers us new possibilities therefore we need to take another perspective on communication.

The literal language, as we know it, is powerful. In fact so powerful, that teaching machines to speak and understand is still one of the greater challenges. It brings great advantages for communication. In case we cannot remember a specific word, it is easy for us to come up with an alternative or to somehow outline it. Even persons that are not speaking the same language, will be able to somehow communicate with each other using gestures or images. But on the other hand literal language has its shortcomings. To describe technical or scientific topics precisely we need a lot of words to bring it into understandable context. Everyone who sat in lecture that was a bit over his skills exactly knows this problem to well. The more concrete and complex something becomes, the more we feel the shortcomings of literal expressions.

Software makes no exception. Applications are built while keeping in mind, that a user will actually see the interface and interact with it. We are using a button because a user sees it and pushes it. Where the button is located or in which context it has which functionality is obvious to the user. At least it should be. But what if he wants to discuss something about this button with his colleagues? This could be a question or criticism. Nevertheless he will need to describe where the button is; in which workflow it appears and so on. Assuming the colleague is not available on location, the usual way would be to create a screenshot, write an explanation, compose an E-mail and send it. From there on the E-mail thread would become the central discussion point related to our button. This is not efficient at all for several reasons:

- What if other colleagues might have something to add to the conversation, but are not included in the receivers list?
- If it is just a short question, the way with a screenshot etc. is not time efficient and exhausting for all participants.
- What if the information in the E-mail thread might be informative for other users in future? They would have to ask the same question again.
- ...



An alternative to using mails for problem solving would be a wiki or forum where all colleagues can collaborate. This partially overcomes the problems listed above. Since anyone who has access to such a platform will have a persistent overview about anything that has been discussed in past and will be in future. Topics can be bundled in threads or articles which allows structuring. Newcomers can use search functions and content tables to easily find content related to a specific issue. Still this approach has the disadvantage of being decoupled from the problem itself. Again lets imagine the problematic button which now will be discussed in a wiki article. First of all the user needs to know about the fact that there is a topic about this issue in the wiki. The experience shows that it is mostly not the first step to use the organizations platform to search for a solution but to ask your colleagues and google instead - and maybe then to start with the search for alternatives. It would be the easiest way if the wiki-article is linked with the button. In this case the user would immediately see that there already is some discussion to that. And following the link would bring him without any detours directly to the related thread.

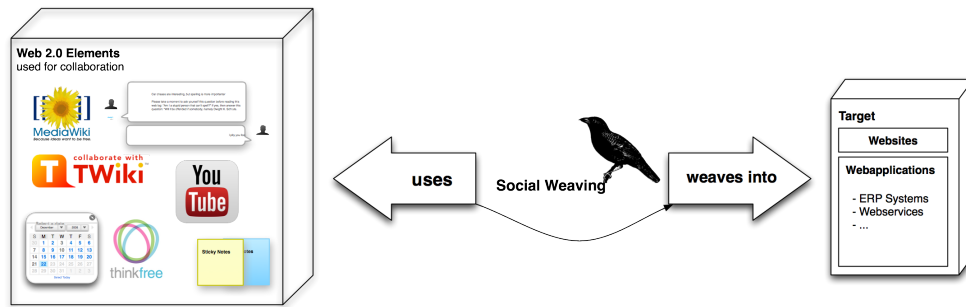


Figure 1: Basic idea of Social Weaving

So in combination we want the possibility to create some form of communication functionality - directly related to the button. And which is visible to a group of users for optionally unlimited amount of time.

Well if the web application would have an implemented comment box beneath the button, that would solve the problem. But that solution brings another bunch of disadvantages. It would require to modify the web application that includes this functionality. And we cannot add comment boxes, links, ... to any element just as precautionary measure. This attempt of solution is not an option.

What if we would have the opportunity to inject social elements directly into our web application without the need to modify it. Basically web applications run in a browser and what is displayed can be modified locally. And that is what we do. We weave social elements into the browser view and synchronize it for different user sessions. This way we reach exactly the functionality we need to solve our problem without touching the web application. We call this process Social Weaving.

Even though Social Weaving overcomes the above mentioned problems - it confronts us with new types of pretty unpleaseant difficulties. Different browser types, unreliable web development styles, complex web technologies, etc. are examples for hurdles that Social Weaving has to deal with.

This thesis will show what is possible with Social Weaving and where its boundaries are.

In the following we are first going to discuss the idea of Social Weaving based on an abstract requirements analysis of a prototype. What functionality does it need to achieve the goals we mentioned above and what are the difficulties? In the second part we are going down on a concrete level where we take the theory from the first part into action and actually explain the architecture and implementation of the prototype, Social Weaver, in detail.

### 3 Social Weaver - Abstract Domain Level

I know it when I see it

---

Potter Stewart

Even though Stewart had something completely different in mind when he used this famous phrase in front of the United States Supreme Court, it is still a quite good explanation why a prototype is useful.

We aim to create a vertical prototype. That means that our system should proof that the general idea is possible to implement. Wiegers writes that a vertical prototype should touch all technical layers to serve as a *proof of concept*[22].

First of all we will start with a prototype driven requirements analysis. In the sections 3.4 What is Social Weaver to ?? we will discuss the requirements on an abstract domain level and put these into a relationship with the *WHY/WHAT/WHO-Model* [21].

Based on that in section ?? we will bring the requirements to a concrete domain level and furthermore explain some interesting details about the derived architecture and implementation.

#### 3.1 The WHY-WHAT-WHO-Model

The WHY-WHAT-WHO model (see figure 2) enables us to discuss on different layers of requirements abstraction. The purpose of the WHY-WHAT-WHO model should not be a strict separation into sections. It is much more to be seen as a context that we can refer to through this paper. The purpose of fitting // @TODO hier muss mehr hin!

In the following all layers are described in detail and connected to parts of this thesis.

##### WHY Dimension

This dimension analyses the existing system or environment we build on to define what is within a possible reach. Potential problems or limitations should be defined in context of this layer. Furthermore we should try to see what impact our system is going to have on the environment.

For this task we first of all need background knowledge which we will gather in a domain analysis. Based on that we will be able to define the problem, that should be our motivation for building such a system.

The sections that are related to the WHY dimension are ?? ?? and ?? ??.

##### WHAT Dimension

The WHAT dimension contains services, constraints and processes that are direct results from the WHY dimension and will lead to the actual goal system. At this point the results from the WHY dimension should be verified if they are

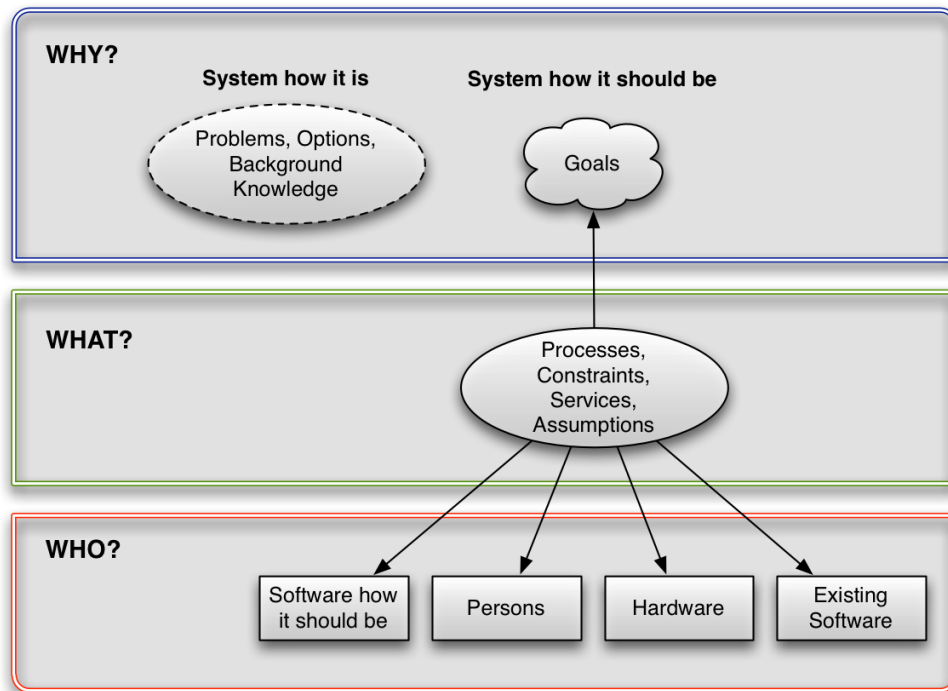


Figure 2: Three dimensions of the requirements types[21]

still applicable and possible to implement. Use Cases and concrete requirements belong to this section.

This thesis sections 3.5 Requirements for Social Weaver and ?? ?? are related to the WHAT dimension.

### WHO Dimension

The WHO dimension is about distinguishing what component has which responsibility. Components in this context do not need to be necessarily hardware components. For instance some user interaction might be classified as component.

// @TODO irgendwie muss behandelt werden wie sämtliche komponenten zusammen spielen...

## 3.2 Domain Analysis

## 3.3 Problem Description

## 3.4 What is Social Weaver

Social Weaver (SoWe) is the name of a prototype system that weaves social web features into web applications. The system consists of a firefox plugin and the server side.

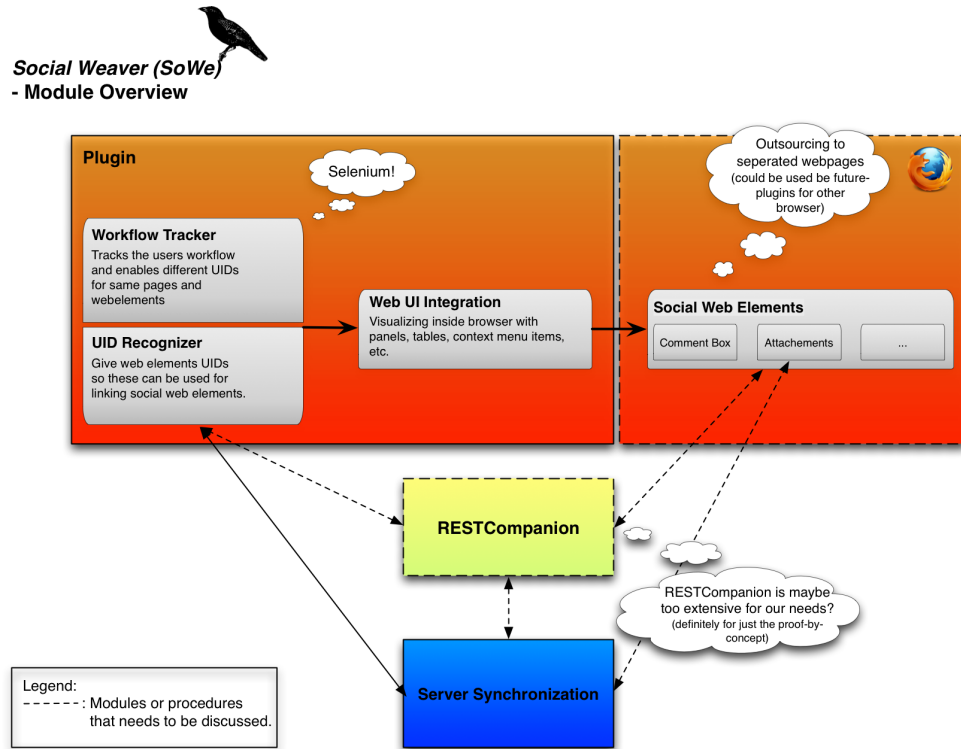


Figure 3: Social Weaver Module Overview

The plugin takes control of one or multiple user sessions and draws the additional content into the browser view. The server application synchronizes with each plugin and distributes updates between several clients.

For a better understanding lets step through a generic use case where a user just opens a web application and modifies some content. The use case enumeration is related to the figure 4.

1. The user opens a web application
2. The SoWe-Plugin sends a notification to the server with all necessary information like user identifier, timestamp, ...

3. After the server receives the plugin message it synchronizes it with its current content in the database
4. The server application responses to the plugin client with content data if some exists
5. The plugin uses the content information from the server to insert all social web elements
6. The user decides to make some changes to the social web content (e.g. adds a comment or creates a new comment box)
7. Again a notification is being sent to the server with containing the changes
8. Server synchronizes the updates and responses
9. Plugin redraws the synchronized content

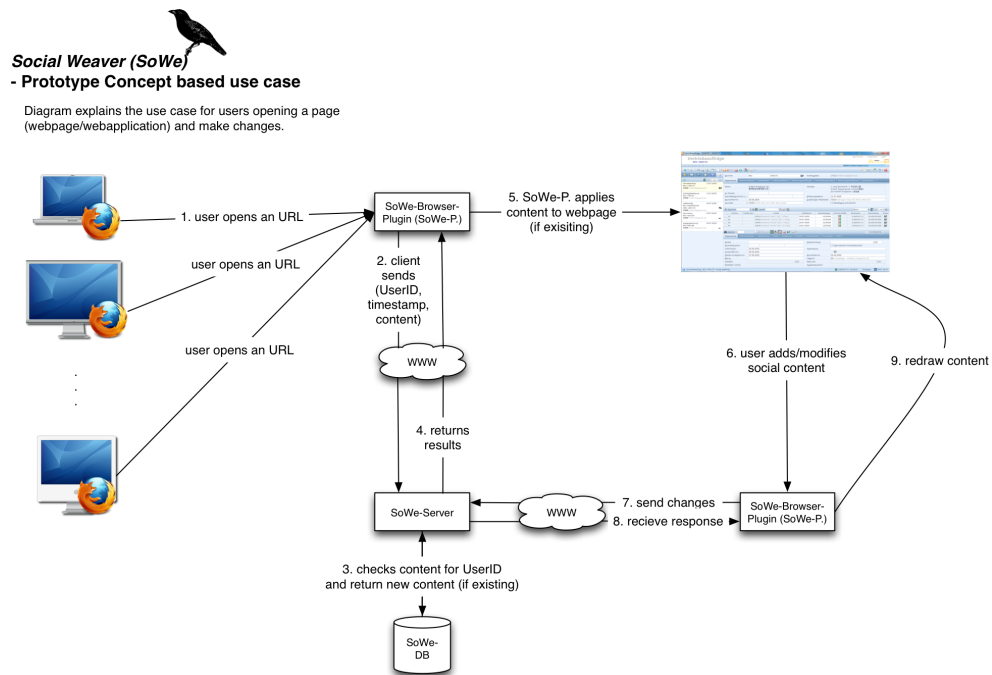


Figure 4: Social Weaver Prototype Use Case

### 3.5 Requirements for Social Weaver

The general goal for Social Weaver is to weave social web 2.0 features into web-based applications. Since this is a broad requirement and impossible to be

applied to any web application right from the beginning; it is necessary to break it down for the prototype.

More specifically the primary goal should be to get a system, that weaves one social web feature into a specific web application. SoWe has to be designed in a modular way, so that it will be possible to add more social media features, support multiple platforms and more web applications. Now that we have a rough idea what SoWe is going to be, lets list some concrete high-level requirements:

1. Browser plugin that supports a comment box
2. Server application that stores and synchronizes data that it receives from different client-plugins
3. Data format for storing and processing data for social web content
4. Communication protocol between plugin and server

With these requirements we can start to specify our enlisting in detail:

### **3.5.1 Browser Plugin**

In the following we define requirements on a abstract domain level according to [21]. A specification to a concrete domain level will follow in section 4.2, where we have specified what technologies to use.

As already mentioned the main requirement is that the plugin supports a comment box. That means that the browser has to display a comment box that is related to specific web element. For example in an online calendar an user adds a comment box related to an appointment that he wants to discuss in detail. Because it should be possible to add multiple comment boxes to any web element, we cannot just drop a box inside the user view, overlapping other interesting parts of the web application. Hence we have the requirement to make additional content visible to the user without interfering with the view on the original content. Possibilities would be fold/unfold-windows or just using small icons as references in the original view and outsource additional social content in external windows.

Of course the plugin needs to be able to communicate to the server application as well. (The server application is explained in the next section: 3.5.2). First of all the plugin needs to receive data that it print to the screen. Secondly changes made by the user has to be reported to the server. Because we are distributing the information between several users, there is also a need for synchronization. User updates may not overwrite updates made by other users etc.

The parser framework will contain application programming interfaces that create and parse the content of our tuples. This way it will be easier to add plugins for other browsers for instance. The data in the content-part of our tuple should have a uniform format no matter what web application or browser is in use. The server application will not need to know anything about the environment the plugin works in - it manages the social web content independently.

Another tricky and important point is the interaction with the web application. Most such sites are dynamic and there exists no static URLs we can refer to. And it is not certain that the same element, that two users refer to in their independent sessions, will have a comparable identifier. This issue definitely needs to be handled specifically for any web application. The good news is that this only affects the plugin. The server application just needs clearly defined identifiers. As a solution for the plugin we will need the possibility to use scripts for identifying elements. For example a script that supports the google calendar will be injected to make the plugin identify same appointments in different user sessions. This requirement is probably the vastly problematic one because it prevents a general usage of Social Weaver.

Lets summarize all the requirements we gained in this section:

1. Displaying and managing a comment box related to specific web element
2. Managing several comment boxes without disturbing the view on original content
3. Communication to server application
4. Creating Anchors
5. Creating content in uniform sending format
6. Parsing content from uniform sending format
7. Identifying web elements across different user sessions

### **3.5.2 Server Application**

The server applications primary requirement is to synchronize different user sessions on one or multiple web applications. A user session is defined within the plugin (which does not mean a plugin can manage only one session). The server basically receives messages from different sessions, synchronizes them and distributes the most current state to all sessions. To establish a lossless synchronization every message contains a timestamp.

We are assuming that every message contains an user identifier, a timestamp and an unique identifier for an element within the web application. This Anchor is the unique identifier for a single user action. For example if a user adds a comment to an already existing comment box that is related to an appointment in a calendar, the server receives the users identifier, the timestamp for the modification and an identifier for the appointment in the calendar. With this information the server can check its database for the comment box and add the new comment.

It is important to remember that the server only uses the received data as identifier. All actions are completely independent to the web application.

Also we may assume that the received message have the same Anchor form as discussed in the previous section.



`(user identifier, timestamp, content)`

The content part from the Anchor will already be in a uniform that has been generated by the plugin. So even the browser type will not matter to the server. The server has to be able to parse the content package and to create a new one that can be parsed by our plugins.

So the requirements for the server application are:

1. Offer service that receives messages from plugin-clients
2. Synchronization for requests from different user-session
3. Persist updates into a database
4. Keep the server application independent to weaved-into web application
5. Parse incoming messages
6. Create outgoing messages

### 3.5.3 Social Weaver - Script Support

The support for external scripts is essential for a generic usage of Social Weaver. The reason why script support is extracted into its own section, is that it should be decoupled from the server and plugin that were discussed before.

The underlying problem is the problematic identification of elements of a web view. There is simply no generic way of identifying elements in the users view across all web sites and applications. For that reason we need an extendable method to support more websites and applications. This could even mean that third-parties could support their own systems by just adding the script without the need to modify Social Weaver directly. In this section we will briefly discuss what the purpose of such scripts is in detail and what requirements we have to fulfill.

The term *script* in our context should contain only information that is needed by the plugin to identify an element. Let us consider the google calendar example once again. The case where we want to match the same appointment field across different user sessions brings the problem that there will be no identifier for the element itself. To the user it is obvious to identify it because of the appointment name, date and time. And those parameters could be just the information we need to extract into our script. How this will look in detail should be discussed in the concrete domain level section.

The usage of scripts should be related to one or a set of URLs. This affects mostly the root URL of a server. But might be used for subparts of a web page or application. As example a script related to `http://www.opensource.org/` will be applied to all subpages like `http://opensource.org/docs/board-annotated`.

But it might be of use to have a special matching procedure for sub pages. In that case a script for `http://opensource.org/faq` would overwrite the more general script.

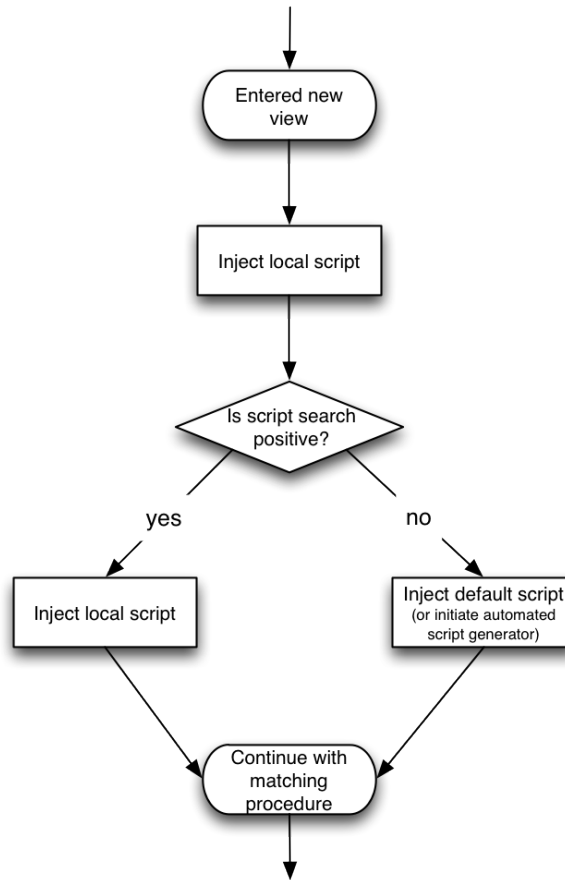


Figure 5: Workflow for Script Using

A set of URLs could be used for scripts that are applicable for many websites.

The workflow when a script is used and when the default matching procedure that comes with the plugin is quite straightforward (see Figure5). When opening a new URL then the plugin should check whether there is a script for that case and depending on the search results proceed with the script or default matching procedure.

On the abstract level we will have the following requirements:

1. Container of all necessary information for element matching
2. Decoupled from browser plugin and server backend
3. Syntax that is easy to read and write
4. Extension of the plugin with parsing methods

5. Default matching procedure should be provided (so the overall functionality is not limited when no scripts exist)
6. Scripts should be related to a single or a set of URLs

## 4 Social Weaver - Concrete Domain Level

### 4.1 Social Weaver - Firefox Plugin

This section will briefly explain what technologies we use for firefox plugin development and describe in detail how the Social Weaver plugin is implemented.

#### Firefox

Firefox is a free web browser that has been released 2004 by the Mozilla Foundation<sup>1</sup>. It is being multi-licensed under Mozilla Public License (MPL)<sup>2</sup>, GNU Lesser General Public License and GNU (LGPL)<sup>3</sup> General Public License (GPL)<sup>4</sup>.

The reasons why we chose Firefox as prototype environment are the high distribution of the browser and an easy extendability with plugins, extensions and so on.

#### Firefox Plugin Development

To improve readability of the coming section 4.2 Requirements for the Plugin, we will discuss some aspects from the Mozilla Add-on SDK (Version 1.13)<sup>5</sup>. Readers who are not interested to much into technical detail or are familiar with the technologies can skip this section.

The used methods will not be just explained independently but brought into context to our prototype planning so it becomes clear what purpose they have.

The Add-on SDK allows to create add-ons for the browser using the most common web technologies (like HTML, CSS, JavaScript, ...). Furthermore it provides a Low-Level-API and a High-Level-API set. The most important interfaces that are being used for our prototype are High-Level-Interfaces and will be explained in the following.

##### Panel

A *panel*<sup>6</sup> is very flexible dialog window. Its appearance and behavior is specified by a combination of a HTML and a JavaScript file. Additionally a CSS file might be used to change the look even further. The limitations of a panel are the limitations of the mentioned technologies. A panel is meant to be visible temporary and they are easy to dismiss because any user interaction outside the

We will use the *panel* for getting user input, displaying information (like in screenshot 6) and to integrate our social media web elements.

---

<sup>1</sup>[www.mozilla.org](http://www.mozilla.org)

<sup>2</sup><http://www.mozilla.org/MPL/1.1/>

<sup>3</sup><http://www.gnu.org/licenses/lgpl-3.0.de.html>

<sup>4</sup><http://www.gnu.org/licenses/gpl-3.0.html>

<sup>5</sup><https://addons.mozilla.org/en-US/developers/docs/sdk/latest>

<sup>6</sup><https://addons.mozilla.org/en-US/developers/docs/sdk/latest/modules/sdk/panel.html>

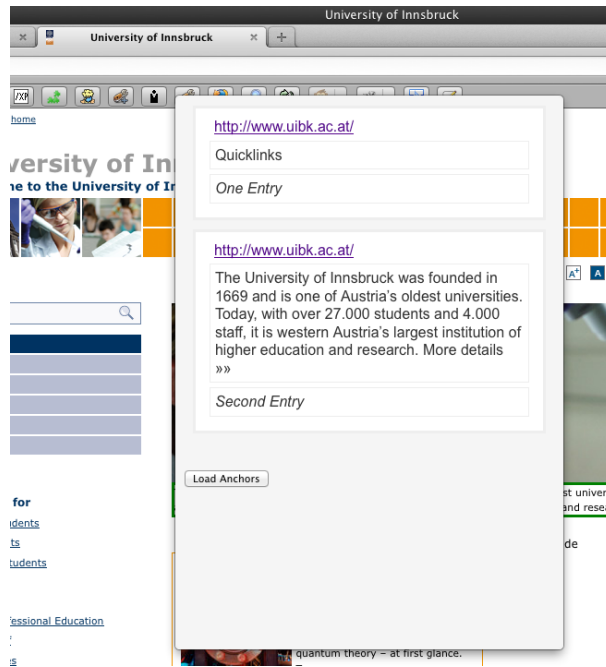


Figure 6: An example for a *panel* that shows a list of annotations

Actually the flexibility of *panel* is the reason why our prototype is able to support social weaving for basically any web element that can be represent in HTML code. Still it is necessary to embed the external HTML code which may leads to boundaries and difficulties.

#### Simple-Storage

This module<sup>7</sup> is an easy to use method to store basic properties (booleans, numbers, strings, arrays, ...) across browser restarts.

With an operation like

```
1 var ss = require("sdk/simple-storage");
2 ss.storage.myNumber = 41.99;
```

we store a number like an object and can it access just as easy like that. The price for such a simple usage is paid with high limitations. For instance searching is basically not possible. Nevertheless we can store an array and search the array.

That is exactly the way how we store our annotations for our prototype. More details will be provided in the next section.

<sup>7</sup><https://addons.mozilla.org/en-US/developers/docs/sdk/latest/modules/sdk/simple-storage.html>

### Page-Mod

The *page-mod*<sup>8</sup> module enables us to act in a specific context related to a web page. Then it becomes possible to attach JavaScripts to it and to parse or modify certain web page parts.

In our context we are going to use *page-mod* to parse the HTML code to find elements that can server as anchors for annotations. And of course to find elements that are already annotated.

### Widget

The module that is called *widget*<sup>9</sup> is simply an interface to the Firefox add-on bar<sup>10</sup>. It is possible to attach *panels* and trigger operations by clicking the *widget*.



Figure 7: Example for a widget serving as activation button

We will use a widget to switch between different modes (see coming section 4.2.1 for more details about how the mode-system works).

### Self

*Self*<sup>11</sup> provides access to add-on specific information like the Program ID<sup>12</sup>, which is important for an official distribution of the add-on. More meta information like the name or the version are accessible via the *self* module. Also bundled external files are integrated by *self*.

<sup>8</sup><https://addons.mozilla.org/en-US/developers/docs/sdk/latest/modules/sdk/page-mod.html>

<sup>9</sup><https://addons.mozilla.org/en-US/developers/docs/sdk/latest/modules/sdk/widget.html>

<sup>10</sup>[https://developer.mozilla.org/en-US/docs/The\\_add-on\\_bar](https://developer.mozilla.org/en-US/docs/The_add-on_bar)

<sup>11</sup><https://addons.mozilla.org/en-US/developers/docs/sdk/latest/modules/sdk/self.html>

<sup>12</sup><https://addons.mozilla.org/en-US/developers/docs/sdk/latest/dev-guide/guides/program-id.html>

Even though it is an important module and part of the plugin - there is no specific usage relation to the system we use.

#### Notifications

This module<sup>13</sup> displays toaster<sup>14</sup>-messages<sup>15</sup> that disappear after a short time.

We use these to keep the user informed without bothering him too much by forcing him to dismiss trivial notifications.

#### Request

This simple to use but yet powerful module *request*<sup>16</sup>, lets us perform network requests. Once we create a *Request* object we can specify whether it is a GET, PUT or POST request. These request types are specified by the REST standard so any web service that supports REST is able to interact with this module[9]. The response from a server is directly accessible like any other JavaScript object.

We are going to use *request* for our communication with our synchronization web service. This includes sending updates, made with the plugin instance, to the server and receiving updates, that were made in other sessions or with different plugin instances, from the server.

#### jQuery

*jQuery*<sup>17</sup> is a free JavaScript library under the MIT License<sup>18</sup> that offers many functions for modifying DOM trees. It has been released 2006 in context of a BarCamp<sup>19</sup> in New York.

Even though this library is not a part of the Mozilla Add-on SDK it is being heavily used by it. Basically any operation that changes or traverses the HTML code (like changing the background color of web elements) is being reached with jQuery.

The reason why jQuery makes it so easy to handle operations on HTML code is based on the selector that searches the DOM tree. Most jQuery operations are based on elements in the DOM tree. With the selector `jQuery()` (which can be equally written as `$()`) any element in the DOM tree can be used to create a jQuery object. This object can be used to perform modifying operations on it. For instance let's step through the following short operation:

---

<sup>13</sup><https://addons.mozilla.org/en-US/developers/docs/sdk/latest/modules/sdk/notifications.html>

<sup>14</sup>Toasters are commonly called notifications that just appear, or slides in the users view - like a toast pops up.

<sup>15</sup>[http://en.wikipedia.org/wiki/Toast\\_\(computing\)](http://en.wikipedia.org/wiki/Toast_(computing))

<sup>16</sup><https://addons.mozilla.org/en-US/developers/docs/sdk/latest/modules/sdk/request.html>

<sup>17</sup><http://jquery.com/>

<sup>18</sup><http://www.mit.edu/>

<sup>19</sup><http://en.wikipedia.org/wiki/BarCamp>

```

1 $('<strong>.user-name</strong>').click(function(){
2     this.empty();
3 });

```

In line one we use the `$` selector to find any element in the DOM tree that has the id `user-name`. Since we use the jQuery selector, any found element can be handled as a jQuery object. This way `click(function(){...})` appends a click observer to all elements. Any time one of those elements is clicked by the user, this triggers the function. This function includes `this.empty()` that will only remove all content inside the element.

Another and more related example of jQuery usage is used in the selection part of the plugin:

```

1 $("<strong>:visible</strong>").filter(function(index) {
2     if(this.content()){
3         return true;
4     }else{
5         return false;
6     }
7 }).mouseenter(function() {...}

```

First of all we request all elements that are visible to the user by using `$(":visible")`. Because it is unlikely that a user will try to select an element without any content - we filter all elements from the result set that are empty. (Such elements can be placeholders, flexible empty space and so on.) Now that our results contains mostly relevant elements to the user, we append `.mouseenter()` to recognize when the cursor is placed above the element. The skipped function itself contains operations to recognize more user operations and element analysis. To discuss more code in detail would be beyond the frame.

We only have seen a brief overview about jQuery operations. To fully understand how the prototype works and to operate with the script support it is advised to gather a better overview using for instance the official W3 introductions: <http://www.w3schools.com/jquery/>.



## 4.2 Requirements for the Plugin

Let us recap what requirements we gathered in section 3.5.1 on the abstract domain level [21]:

1. Displaying and managing a comment box related to specific web elements
2. Managing several comment boxes without disturbing the view on original content
3. Communication to server application
4. Creating Anchors
5. Creating content in uniform sending format
6. Parsing content from uniform sending format
7. Identifying web elements across different user sessions

In the following concretization we apply the abstract requirements to our environment which is the *Mozilla Plugin Development SDK*<sup>20</sup>. In the following we will handle each and every requirement, mentioned above, separately:

### 4.2.1 Display Management Requirement

Obviously "Displaying and managing a comment box related to specific web elements" consists of multiple sub requirement that we need to distinguish.

Before we are able to annotate something, we first of all need a function to select or recognize a web element the users cursor points to (check 1.1 in figure 8). *Selecting* in this context means that we analyze the Document Object Model (DOM)<sup>21</sup> tree. The selection itself is easy to implement using the function *mouseenter* and *on('click')* from jQuery library<sup>22</sup>. It becomes problematic to find this element again without any user interaction. Therefore we need to set well chosen parameters that are used to determine the element. Next time we need to find the element - only the parameters can be used to find the element in the DOM tree. This issue will be topic in the section 4.7 Social Weaver - Script Support.

Theoretically it could be possible for user to select any element in the web view - but practically this would make the selection procedure confusing for development as well as for the user. Therefore we apply some filters. Elements like empty boxes, placeholders and so on will not be selectable. But still it should be clear to the user what he might select.

To achieve this functionality we create thin rectangles around every element that is an selection option. These rectangles only appear for time the plugin is in selection-mode and uglify the web view just for a certain time. More

---

<sup>20</sup><https://addons.mozilla.org>

<sup>21</sup><http://www.w3.org/DOM/>

<sup>22</sup><http://jquery.com/>

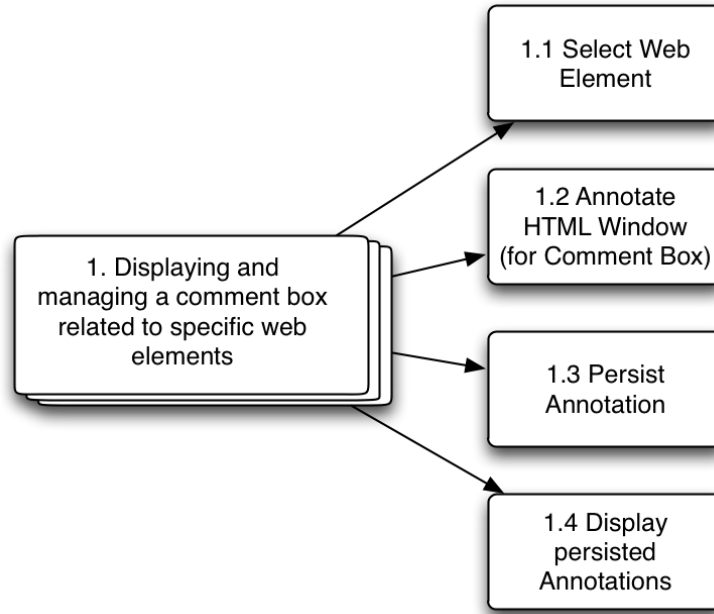


Figure 8: Partition of the first plugin requirement to sub requirements

on the different mode types in 4.2.2 Managing several comment boxes without disturbing the view on original content .

Now that we can locate a specific web element we may annotate some social web element. For reasons of flexibility and simplicity we just annotate a HTML window (check 1.2 in figure 8), where we can inject any external HTML code. The Mozilla SDK high-level APIs <sup>23</sup> offers all necessary tools to insert a HTML box as a *Panel*<sup>24</sup>.

The annotation anchors will be visible to the user in form of a colored background that we create by modifying the DOM tree. If the user clicks on such an element the already existing panel will be opened.

To decouple our annotated data (like anchors, annotations, ...) from the actual synchronization, which will be covered later, we want to use a storing system that is also provided by the Mozilla SDK (see 1.3 in figure 8). The high level API *simple-storage* <sup>25</sup> enables us to store all information we need and recall them. Just the synchronization mechanism should modify this data set. All displaying procedures should be outside of server communication reach.

<sup>23</sup><https://addons.mozilla.org/en-US/developers/docs/sdk/latest/modules/high-level-modules.html>

<sup>24</sup><https://addons.mozilla.org/en-US/developers/docs/sdk/latest/modules/sdk/panel.html>

<sup>25</sup><https://addons.mozilla.org/en-US/developers/docs/sdk/latest/modules/sdk/simple-storage.html>

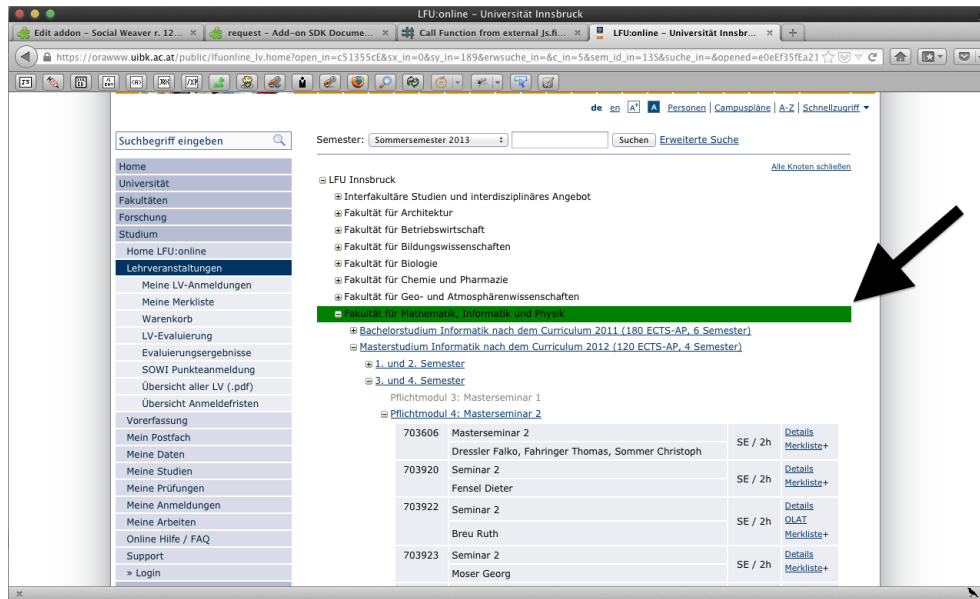


Figure 9: Rectangle shows that the underlying element is possible for annotation

This means that if an annotation is created, the changes are written locally into the plugin local data storage. And only after the persistence is complete, the synchronization procedure will be initiated. While synchronizing the changes will be transmitted to the server.

The last sub requirement is to redisplay existing annotations (check 1.4 in figure 8) from our *simple-storage*. Besides using the same techniques for drawing content and retrieving it from the storage we need to match the web page content to our saved annotations. For that we use a matcher instance that checks the DOM tree for IDs that we are already using.

This is actually only trivial on a very simple basis. Let us assume that we will have more than one element attached to the same web element. Or we have different user sessions and/or include a workflow so that we need to distinguish the same element for different instances of the webpage. Than it becomes quite complicated to generate IDs that we can rely on. Nevertheless these issues will just affect the way we assign IDs to elements and how we retrieve them. The requirement 1.4 is just about matching existing IDs to a web page.

As already mentioned we use a matcher that checks the DOM tree for IDs. In case we have an anchor in our *simple storage* then we modify the web page HTML code similar as we did for requirement 1.1. Visual differences are that we do not modify the background of an element but generate an rectangle around it instead (see screenshot 10).

This way we are able to show the user which elements are annotated. Of course without further information it is not obvious what is annotated exactly.

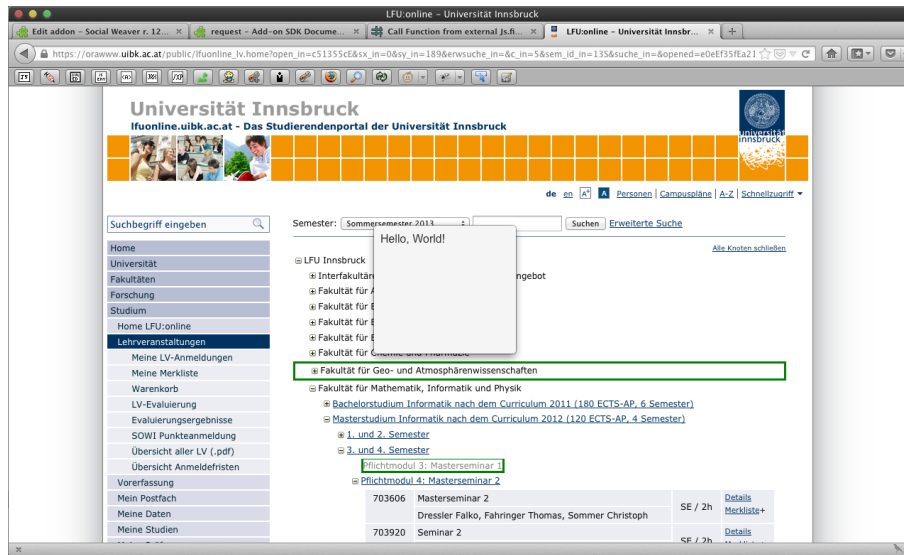


Figure 10: Rectangle shows that the underlying element is possible for annotation

What we need is a easy to access functionality so that the user can find out what the annotation is.

For that reason we modify the above mentioned matcher class to generate a panel in case the user performs a *mouseenter* operation. This panel should show a brief version of the attached social element. In our case it could be the name of the context the comment box is related or the names of the attendees (our example screenshot just print outs "Hello, World!" 10).

#### 4.2.2 Managing several comment boxes without disturbing the view on original content

This requirement is not directly about functionality but should ensure a positive user experience. It will be possible to attach annotations to nearly any element in a web application. This is a lot of potential additional footage. Nevertheless the user needs to be in the position to navigate like usually within the application.

Basically there are two options to guarantee such a requisition. Either we minimize the overhead that we display into the web view or we display additional information only at the appropriate time. To achieve the best results we combine both possibilities by introducing the mode system:

The plugin will have several modes that support different functionality and show different content. Those modes can be switched by clicking the widget in the addon bar. Every mode has its own logo, see Figure 11.

We distinguish the following modes:

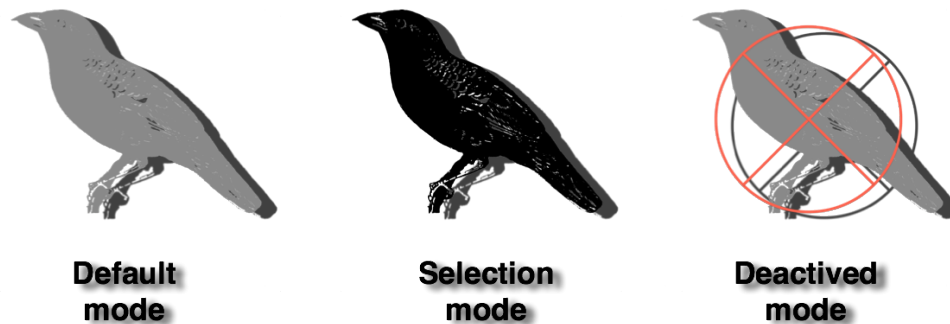


Figure 11: Widget representation for different modes

#### 1. Default mode

In the default mode the plugin runs passively. It does not disturb the user but runs synchronization and matching procedures in background. When the user opens a new web view, the plugin checks its database for annotations that belong to the view. If a positive match is found, it will be drawn into the user's webview.

The navigation of the browser is as usual, except for the annotations that include a click handler that triggers the function for opening the social element.

#### 2. Selection mode

The selection mode interferes eminently with the navigation and the representation of the regular browser view. Activating the selection mode will mark all selectable elements with rectangles around it. Moving the cursor around the web view will additionally mark the element beneath the cursor to visualize what element will be marked when clicked. In this mode clicking links, buttons and so on, will not trigger the functionality of those elements but instead select them so an annotation might be created.

Disabling the selection mode will clear the view from the previously mentioned marks. Only successfully created annotations will remain persistent.

#### 3. Deactivated mode

Deactivating the plugin entirely disabled any functionality. This can be useful in worst cases like that the plugin prohibits regular navigation or the annotation marks interfere with the user's view.

The mode system could be a starting point for extending Social Weaver for

```

1 var sync = Request({
2     url: 'http://localhost:9998/anchor',
3     onComplete: function(response) {
4         for(var i = 0; i<response.json.length; i++){
5             var r = response.json[i];
6
7             newAnchor = new Array(r.anchorURL,
8                 r.ancestorId, r.anchorText);
9             var newAnnotationText = r.annotationText;
10            handleNewAnnotation(newAnnotationText,
11                newAnchor);
12        };
13    }
14 });
15 sync.get();

```

Figure 12: Sample JavaScript code for retrieving JSON objects from a web service with a GET REST request

multiple user sessions in one plugin context. Or providing securely authenticated sessions that can only be activated when correct credentials are provided.

#### 4.2.3 Communication to server application

To share our comments or annotations with other users we will need a server side synchronization procedure. This section is only about the requirements that are related to the plugin side. (Details to the server side will be discussed in ?? ??.)

The first step to achieve this goal is to establish a communication between the plugin and a web service. For this purpose we are going to make use of the *request* module from the Mozilla Add-on SDK. It provides an easy to use JSON<sup>26</sup> and REST([9]) assistance.

We split this into the following sub requirements:

Plugin receives updates from server

What the plugin needs to know from the server is a set of Anchors. Those Anchors contain information like the author identification, a timestamp and of course the content. (see ??) So at this point we assume that our server will provide a set formatted in JSON. The plugin generates a request to retrieve this data.

This goal is surprisingly simple to achieve. In the sample code 4.2.3 we just need to specify the URL of the web service and we are able to access the JSON objects right away exactly like JavaScript objects. Then we use the JSON objects to create an anchor entity and use the existing `handleNewAnnotation(newAnnotationText, newAnchor)` method to store it in our *simple-storage* list.

---

<sup>26</sup><http://www.json.org/>

Our prototype is a proof-by-concept system, therefore we keep the synchronization really simple. Instead of checking for new annotations and match them with the already existing data, we just rewrite our local plugin data set with a copy from the server. This technique could easily lead to corrupt and inconsistent data sets. But since the prototype is not meant to be used for confidential data or in any real world scenario at all - we will just take our risks.

Plugin sends updates to server

When a user creates a new annotation or modifies it - the plugin should send an update to the web service immediately. Again we will set up a method using the *request* module.

How the data format looks like and how we will parse incoming messages will be reviewed later (in the sections ??, ?? and ??.)

### 4.3 Social Weaver - Web Service

The coming part will be about the implementation of the web service that provides interfaces for our previously built plugin.

### 4.4 Used Technologies

Before we discuss our web service architecture we first of all will list the used technologies and give a brief explanation. Readers who are familiar with the following terms may skip this section. After pointing out the architecture, we will map our defined requirements to our implementation and explain how those are achieved.

Model View Controller (MVC)

OpenJPA

PostgreSQL

RESTful Web Services

Spring

AspectJ

### 4.5 Web Service Architecture

The web service is a common MVC architecture that uses JSON/RESTful interfaces. The persistence layer is connected to a PostgreSQL database using OpenJPA.

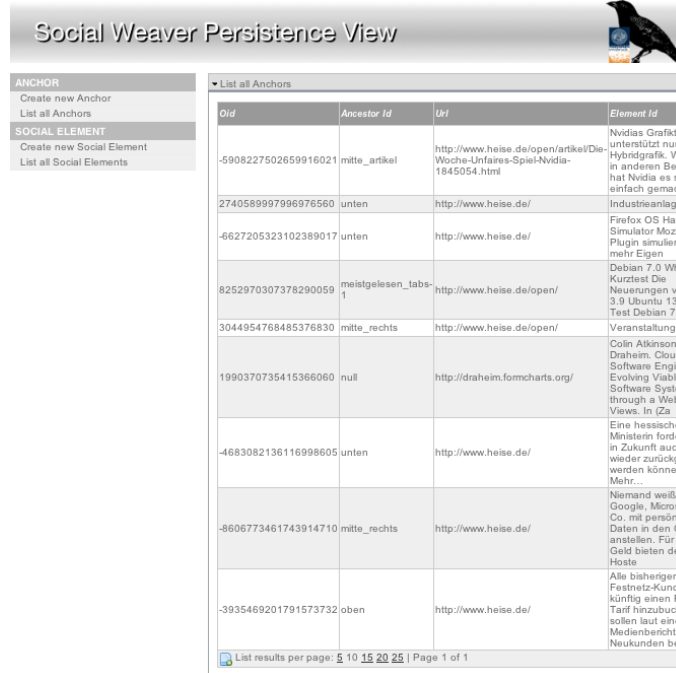
Our main entities are the anchor and social element. The anchor has been already discussed from the concept point of view. The entity contains all the parameters that are necessary for matching web elements. This set of parameters can differ from one web application to another. Additionally the anchor entity contains an unique object identifier (OID) that defines the anchor even across different user sessions. This way updates can be performed more easily without having to check for all parameters every time. Which would be hard especially in those cases where an unusual set of matching parameters is being used. To avoid this difficulty but to still keep all parameters related we generate a hash from a combination of all relevant parameters. This hash is used as OID. Besides that the Anchor holds a timestamp with information about the last modified date. We use this data for the synchronization procedure.

The social element entity is handled as a separate entity but from the concept perspective it is a part of the Anchor. Basically it works as a container for any kind of social content. Because our prototype will just provide a simple HTML inject, the SocialElement will contain an URL and a reference to the Anchor. But it will be extendable to hold data for native and more complex social element types.



The entities are implemented as beans and therefore being directly persisted in the PostgreSQL database.

According to the model view controller pattern there are also controllers for the entities that provides several interfaces that are accessible through the standard REST requests.



Id	Ancestor Id	Url	Element Id
-5908227502659916021	mitte_artikel	http://www.heise.de/open/artikel/Die-Woche-Unfares-Spiel-Nvidia-1845054.html	Nvidias Grafiktre unterstützt nun Hybridgrafik. Wie in anderen Bere hat Nvidia es sic einfach gemach
2740589997996976560	unten	http://www.heise.de/	Industrieanlager
-6627205323102389017	unten	http://www.heise.de/	Firefox OS Hanc Simulator Mozilla Plugin simuliert r mehr Eigen.
8252970307378290059	meistgelesen_tabs-1	http://www.heise.de/open/	Debian 7.0 Whe Kurztast Die Neuerungen vor 3.9 Ubuntu 13.0 Test Debian 7.0
3044954768485376830	mitte_rechts	http://www.heise.de/open/	Veranstaltungsk
1990370735415366060	null	http://draheim.formcharts.org/	Colin Atkinson, I Draheim. Cloud Software Engine Evolving Viable Software System through a Web t Views. In (Za
-4683082136116998605	unten	http://www.heise.de/	Eine hessische Ministerin fordert in Zukunft auch wieder zurückge werden können. Mehr...
-8606773461743914710	mitte_rechts	http://www.heise.de/	Niemand weiß, v Google, Microso Co. mit persönlich Daten in den Ck anstellen. Für w Geld bieten deu Hoste
-3935469201791573732	oben	http://www.heise.de/	Alle bisherigen Festnetz-Kunde künftig einen Fix Tarif hinzubuchc sollen laut einen Medienbericht *) Neukunden be

Figure 13: Screenshot from Social Weaver Persistence Web View

The View from our Model View Controller architecture is a web view that allows the user to check the content manually (see Figure 13). This is just a pleasant side feature and not related to our Social Weaving use cases and for that reason this part should be discussed no further.

## 4.6 Requirements for the Web Service

In section 3.5.2 we defined the following requirements for the web service:

1. Offer service that receives messages from plugin-clients
2. Synchronization for requests from different user-session
3. Persist updates into a database
4. Keep the server application independent to weaved-into web application

- 5. Parse incoming messages
- 6. Create outgoing messages

#### **4.6.1 Offer service that receives messages from plugin-clients**

Clearly we accomplish this requirement with the RESTful web service interfaces. The messages that we transmit are the Anchor information right away.

#### **4.6.2 Synchronization for requests from different user-session**

#### **4.6.3 Persist updates into a database**

#### **4.6.4 Keep the server application independent to weaved-into web application**

#### **4.6.5 Parse incoming messages**

#### **4.6.6 Create outgoing messages**

## 4.7 Social Weaver - Script Support

The following part will explain how the script support looks like in detail for our firefox plugin. In section 3.5.3 we learned about the purpose of the script support and what its goals are. The coming part will apply this knowledge practically to the prototype development. Before we start stepping through the gathered requirements from 3.5.3, a brief example of a script use case will be explained to provide a better overview to the reader.

A script contains the information about how an element in a web view might be found. We use JSON as format for the scripts because of the support that JavaScript provides for that standard. The script defines a set of rules that will be used to perform operations on the selected element. The results from these operations will be generated to a payload. This payload might be the information for anchors, URL and content related to social elements. The server handles this payload as one value and does not parse it. All meta information that are needed by the server will be transmitted separately from it. To understand how the script is used to generate an anchor format take a look at the example:

```
1 {
2   rules:
3   [
4
5     {"doc_location" : "document.location.toString()"},
6     {"element_content" : "$(matchedElement).text()"},
7     {"element_children" : "$(matchedElement).children().text()"},
8     {"element_content": "matchedElement.innerHTML"},
9     {"dom_path": "true"}
10  ]
11 }
```

The purpose of this script example is to show the different possibilities and not a real scenario use case. It will become obvious why this set of parameters would be no good choice.

Basically a script is a set of rules. A rule consists of a keyname and the actual operation that is being used to perform an action. The keyname can be any string withing quotes chosen by the script author. The keynames should be unique in one script though. The operation part offers different opportunities:

- jQuery Operation

jQuery is a great possiblity to traverse the DOM tree and it is possible to inject jQuery commands directly in the script. It is necessary that the command returns a string that is used for identfitication. The first line

```
1 {"doc_location" : "document.location.toString()"}

```

would tell the plugin to save the document location - which is the plain URL in most cases.

To trigger an operation related to the element that has been clicked by the user we might use the keyword `matchedElement`. In case a jQuery

method is used it is still necessary to transform the matched element to jQuery format (`matchedElement`  $\rightarrow$  `$(matchedElement)`).

- JavaScript Operation

Even though most functionality related to DOM tree traversing should be covered with jQuery it might be of use sometimes to use JavaScript commands directly.

```
1           {"element_content": "matchedElement.innerHTML"}
```

This line is an example for how to retrieve the HTML content of an element. This information might be used for matching elements for instance.

- Predefined Operation Some functions we need are not directly provided using JavaScript. The best example for such a case is the DOM tree path. We can use the DOM tree path for distinguishing similar elements. This functionality is provided by the plugin and can be enabled or disabled with the rule:

```
1           {"dom_path": "true"}
```

## 4.8 Requirements for the Script Support

In the abstract section (3.5.3) we defined a couple of abstract requirements that we need to specify for a proper implementation. Those requirements were:

1. Container of all necessary information for element matching
2. Decoupled from browser plugin and server backend
3. Syntax that is easy to read and write
4. Extension of the plugin with parsing methods
5. Default matching procedure should be provided (so the overall functionality is not limited when no scripts exist)
6. Scripts should be related to a single or a set of URLs

### 4.8.1 Container of all necessary information for element matching

For matching different elements we need the flexibility to use a variable set of rules. It would be a drawback to have a static rule system. The problem is that in some web views we need different operations to match elements than on others. Depending on the environment we need a specific container with the rule set.

We solve this issue by introducing the payload container that contains a JSON array with all information that is set by the script. This way we are able to extend information for element matching and modify it just by changing the script. The plugin and backbone of our system will not necessarily have to be changed because it will always be handled as a single JSON string.

### 4.8.2 Decoupled from browser plugin and server backend

The script itself does not contain any information of the browser plugin type or the server module. The defined rules need to be supported by the plugin though.

A plugin that does not support jQuery commands would not work with our script example from above. Or the *predefined operation* (mentioned in 4.7) is another case that needs to be supported by the plugin. Using unknown or unsupported operations will lead to unexpected results.

The server on the other hand is completely decoupled from the script support. The payload is just one column and the information about element anchors is of no use for the server. This is way the payload will not be parsed on the server.

Even corrupt payloads that cannot be used by the plugin will be perfectly synchronized at the back end. This might seem not much of use at first appearance. But there is the case when a the payload for matching an element becomes defect not by using wrong rules, but because of some changes in the web view. Then we still want to keep the information about the element and just re-create the payload to it. Hence it makes sense to keep even those elements in the server database.

### 4.8.3 Syntax that is easy to read and write

It is a desirable goal for the future to have a system, like a script generator, that can be used by persons without any technical knowledge. But since we conduct prototype development we will at least assume knowledge about web development. To choose the right rules and operations for element matching, requires this kind of knowledge anyway.

JSON is commonly used format and even possible to read by persons without technical knowledge. When providing a template with some example rules it is obvious to see how these rules might be extended. The most challenging part is to find and define the correct set of rules - but this is beyond the scope.

Beyond the criterion that scripts are human readable it would be easy to create a form or generator with a graphical interface to generate such scripts. Error handling could be supported already at this point. Even more desirable is an automatic script generator that generates scripts from user behavior without his active interaction (or at least minimal).

### 4.8.4 Extension of the plugin with parsing methods

When we were talking about rules and operations for element matching, this included short Java Script or jQuery commands. Technically we are not limited to short commands and it is possible to insert a whole command block as a rule. But for debugging reasons this is not a good solution. And some operations may require more than just a chain of jQuery commands.

For example lets take the procedure of determining the DOM path of an element in the tree. The idea is to start with the class name of the element

itself. Then check the class name of its parents so long until the root of the DOM tree is reached. All names chained together describe the path of the element in the tree. The fact that this result is not necessarily unique underlies the problem that DOM trees are often ambiguous. We will take a closer look on this issue in 4.9 Ambiguity Problem. But for some web views the DOM path still might be the right choice.

To understand why such a procedure is not possible to be fitted in a script we need to take a look at the code first:

```

1      var rightArrowParents = [];
2
3      $(this).parents().not('html').each(function() {
4          var entry = this.tagName.toLowerCase();
5
6          if (this.className) {
7              entry += "." + this.className.replace(/ /g, '.');
8          }
9
10         rightArrowParents.push(entry);
11     });
12     rightArrowParents.reverse();
13     var domPath = rightArrowParents.join("");

```

In line three we initiate a loop for all parent elements of the selected element `this`. We watch out for every parent that has a `className` in line six. In a positive case we add this information to our path result.

The problem why we cannot insert such an operation into a script is, that we need an outside variable for keeping track of the results through the loop iterations. Line three until eleven is the actual jQuery command. But it would be a mess to pass the whole code with the script. Theoretically it would be possible to support such scripts but it would make the script idea even more error prone.

Instead we use *predefined functions*. Those functions are implemented directly in the plugin and can be triggered using defined rules in the script.

To enable the DOM path parameter in the rule set we would just need to insert this rule:

```

1      {"dom_path": "true"}

```

The plugin would recognize this rule and run the code that determines the DOM path. In the section 4.8.2 we already mentioned that this way of supporting more complex operations leads to minor coupling between plugin and script.

#### 4.8.5 Default matching procedure should be provided

Even though it is not possible to provide a default matching algorithm that will apply to any web view, it is desirable to at least have some default matching in case to specific script for the visited URL exists. If the user visits an new environment there is at least the chance that some elements might be matched.

To fulfill this requirement the plugin will provide a default script. When a new web view will be loaded - all scripts will be checked whether one of them

applies to the new page. If it is not the case the default script will be chosen and the user notified.

#### **4.8.6 Scripts should be related to a single or a set of URLs**

Every script, except for the default script from 4.8.5, mostly needs to be related to a specific environment. When we talk about environment in this case, it means views that inherit from a root URL. As example a script can be related to `http://www.gnu.org`. But any page that has this URL as root, like for instance `http://www.gnu.org/philosophy/philosophy.html`, will still apply to the same script.

The assumption is that views in the same environment apply to the same web architecture and hence its elements can be identified similarly.

Obviously this must not hold for every case. Theoretically a web view can have a totally different architecture than its root. But for prototype development this drawback is considered as tolerable.

## 4.9 Ambiguity Problem

Throughout the abstract (3) and concrete (4) section about the prototype development we often noticed issues that make it tricky to achieve the proposed goals. The greatest challenge by far is the unique recognition of elements in a web view. More precisely: the finding of an element in the DOM tree.

This procedure consists of the two steps:

1. User choses element from view
2. Plugin searches for this element from the DOM tree

The first step is easy because we have the clear information from the user input. The hard work is done by the user. Still our problematic situation takes place already at this point.

Hence the plugin needs to know in what way it is going to identify the element in the second step, it is necessary to gather all the information in first step. This is were the rules from a script (4.7) come in. Those rules are commands that are executed and retrieve results. Those results identify the element.

In the second step, when a view is opened, all appearing elements are checked with the script rules and the results compared to our results from step one. If the results appears to be the same - we assume it is the searched element.

The fact that we can only assume, finally leads us to the actual problem of this section. The DOM tree can be ambiguous and as a matter of fact this is no exception.

### 4.9.1 Ambiguous Grammar

We know the origin of ambiguity in computer science from formal grammars. A grammer is ambiguous for which there exists a string that has more than one leftmost derivation. Check Figure 14 for a common example of a ambiguous grammar.

1  $S \rightarrow SS \mid (S) \mid \epsilon$

This generates a chain of correct opening and closing brackets. The grammar is ambiguous because we have to ways of generating the same string  $()()()$ . For the parse tree and step by step creation see again the figure 14.

### 4.9.2 Ambiguity for Element Matching

What is the concern of ambiguous grammars when we talk about web architectures? We will use the basic idea of ambiguity to describe the problem giving elements unique parameters. Imagine we would try to identify elements just on their location in the DOM tree. At a first glance this may appear as an effective way since we are used to storing data in tree based data structures.

We explain the problem with the HTML example in Figure ...

Determining the DOM path for **Software Development** would return the value: `<html>,<head>,<title>`. In this case the path would be a unique



$S \rightarrow SS \mid (S) \mid \epsilon$

Two leftmost derivations for:  $()()$

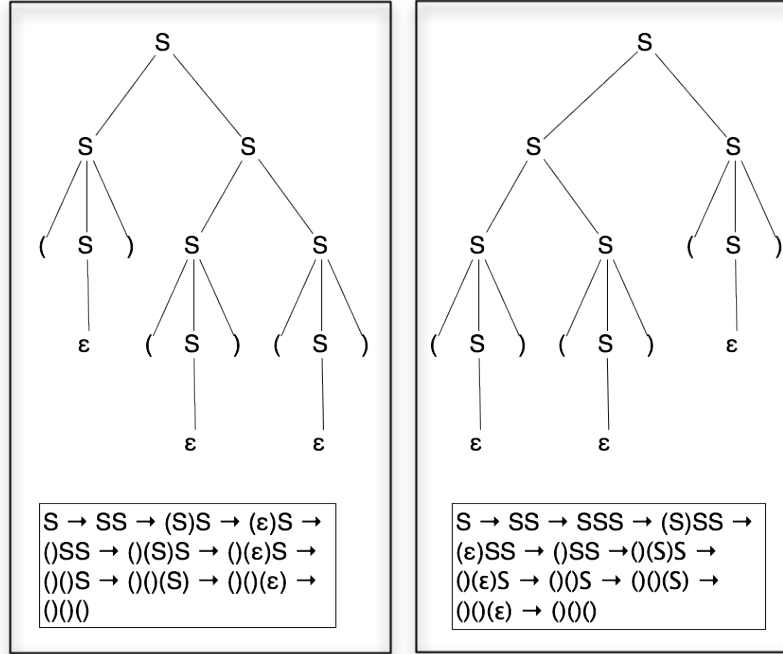


Figure 14: Example for an ambiguous grammar

identifier in this environment. The DOM path for **Construction** would be  $\langle \text{html} \rangle, \langle \text{body} \rangle, \langle \text{ul} \rangle, \langle \text{li} \rangle$ . This path would not only apply to all  $\langle \text{li} \rangle$  elements in the same  $\langle \text{ul} \rangle$  context, but to all  $\langle \text{ul} \rangle$  lists in the same subpath (which is  $\langle \text{html} \rangle, \langle \text{body} \rangle$  so far). Using this path would apply to all six list elements.

For the moment we forget that we can use other parameters (like the  $\langle \text{li} \rangle$  content or surrounding parameters like the  $\langle \text{b} \rangle$  blocks. When we represent the HTML code as DOM tree, we receive a structure where it seems that every element can be located uniquely (see Figure 16). Then the path for **Construction** would be  $\langle \text{html} \rangle, \langle \text{body} \rangle, \#1 \langle \text{ul} \rangle, \#2 \langle \text{li} \rangle$ . The practical problem is that it is very hard to retrieve in the DOM context. Every element has information about its ancestor but not about its siblings. The most effective way would be to map the whole DOM tree an a structure where we attach parameters to the elements. This approach would work precisely on static environments. But most web pages and applications are not static. Therefore we cannot assume a not-changing DOM tree. Even a little change leads to a restructuring

```

1 <html>
2   <head>
3     <title>Software Development</title>
4   </head>
5   <body>
6     <b>Software Development Activities</b>
7     <ul>
8       <li> Requirements
9       <li> Construction
10      <li> Deploying
11    </ul>
12    <b>Software Development Methodologies</b>
13    <ul>
14      <li> Spiral
15      <li> V-Model
16      <li> Scrum
17    </ul>
18  </body>
19 </html>

```

Figure 15: HTML example for showing ambiguity

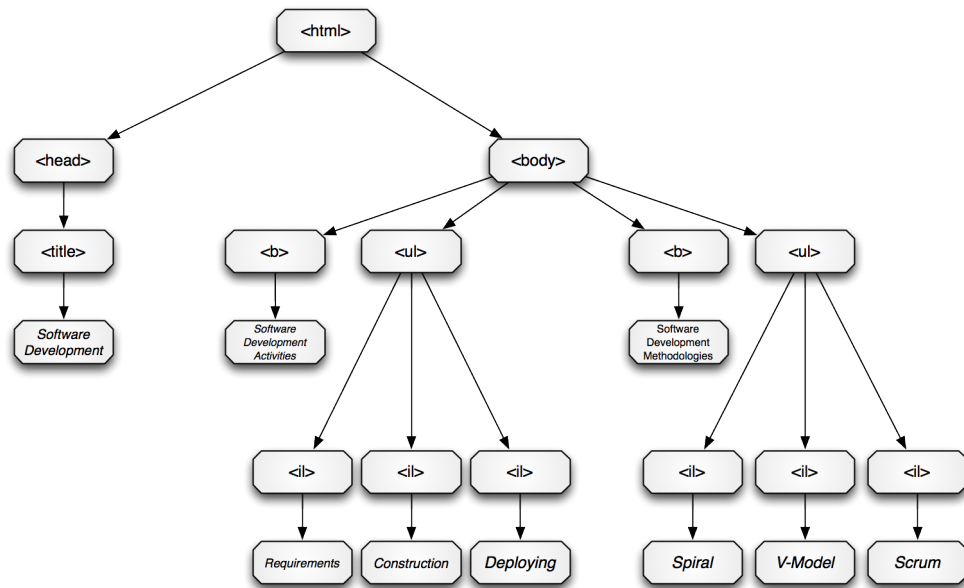


Figure 16: DOM tree representation for HTML code in Figure 4.9.2

## 5 Social Weaver Analysis

### 5.1 Social Weaver in Action

This chapter will lead us through an real example where Social Weaver is being used. It will be explained which components are used in what situations and how they interact with each other. Because we used the Google Calendar example several times it is only fair to use it finally for an overview.

#### Initial Scenario

The following explanations will be based on a scenario with two users (Alice and Bob) who both are running the Social Weaver plugin in Firefox and are connected to the same Web Service (which means they share the same Social Weaver session).

Additionally they obviously need a shared google calendar. For accessing the calendar they use the default web service provided by Google and no alternative client.

The scenario will consists of the following steps:

1. Alice weaves a comment box to an appointment in the shared calendar
2. Alice uses the comment box to leaves a question
3. Bob logs in and answers Alice's question
4. We manually destroy the anchor directly in the database
5. Alice recognizes this failure and relinks the comment box

In the following two sub-procedures, update and matching, will be explained. The reason why we handle those seperately is, that we will use them more frequently in the whole process. That way we can just refer to them and keep the focus on the actual workflow.

#### Update Procedure

The synchronisation for Social Weaver is quite simple. Basically the plugin sends at start up (or when asked) two parameters in a JSON array to the web service using a authenticated POST request. Those parameters are the current URL and the timestamp of the last update. If Alice starts up her first browser the first time, the plugin would send the following JSON file:

```
1 {  
2   last_update=12341234,  
3   current_url='www.cal.google.com'  
4 }
```

The web service uses those information to determine whether there are new and relevant anchors or not. In the positive case (see Figure 17) the anchors

are returned. In Alice's case nothing will be returned since we have no marked elements.

What happens at the server with those data in detail? We use the timestamp of the last update and the current URL to create a query that receives only the corresponding anchors.

Through a simple HTTP header authentication we know which user is getting access to the anchor data. Even though it is not really relevant in our simple case. It would be more an issue when having multiple users in different session in one Social Weaver context. But such scenarios will not be covered in this thesis.

**Sequence Diagram for successful Synchronization**

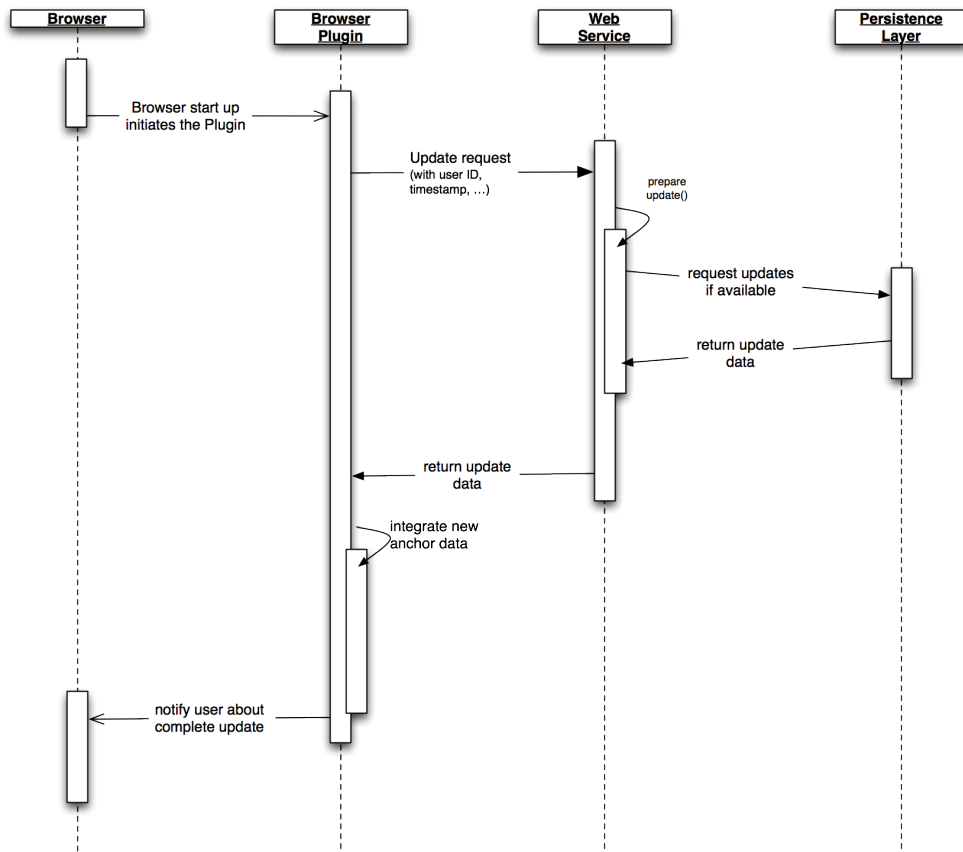


Figure 17: Sequence diagram for a successful plugin update

## Matching Procedure

When we use the term matching procedure it means that existing anchors are visualized to the user. Before every matching procedure we assume that an

update is triggered to ensure that the newest data is being used.

Beyond the update procedure there is no need communicate with the server. When the user opens a new URL it triggers the matching procedure. The plugin searches its local content whether there are some anchors for this URL. In a positive case (see Figure 18) the content is visualized to the browser view.

At this point Alice would receive nothing from the plugin since no anchors exist for `www.cal.google.com`.

The way how anchors are retrieved from the plugin is quite similar how it is done at the backend. The major difference is that we do not use any timestamps at this time. There is no need for that since we assume everything is up to date after the update procedure.

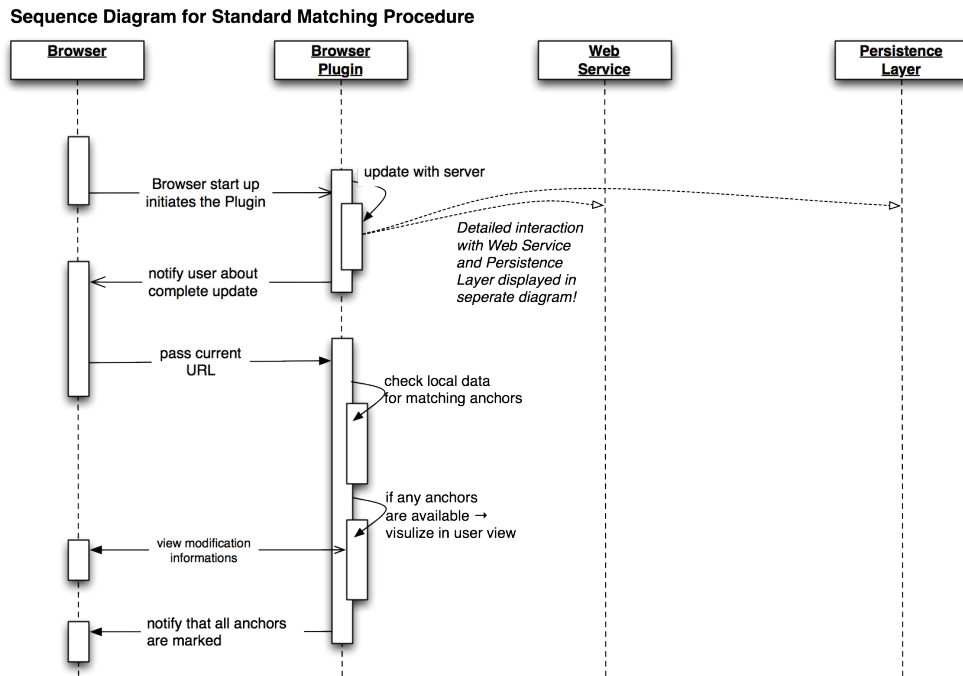


Figure 18: Sequence diagram for a standard matching procedure

## Scenario Execution

Now that we learned about the two sub-procedures we are able to start with the actual scenario. First step is going to be that Alice weaves a comment box to an appointment:

Alice enters `www.cal.google.com` which first of all triggers an update. Afterwards potentially new anchors would be displayed in the browser - which is not the case right now. Now Alice is able to mark an appointment. By clicking

an element (in edit mode, see ???) she appends a comment box. In the background the plugin runs the script (or scripts) that are related to the URL to define an identifier for the element. Using this identifier, the content-data for the comment box and the current URL the plugin creates a message in JSON format and passes it to the server.

The content for the comment box in this case is just a link. We use an external comment system that will be injected as HTML code. Where or how exactly this comment box is defined, is not relevant to our matters.

For our example the JSON file might look like:

```
1 {  
2   content-data='www.chatsystem.com/alice-appointment-17349',  
3   element-id='7234808234088023480',  
4   current-url='www.cal.google.com'  
5 }
```

The message is passed as an authenticated POST Rest request. The web service performs some checks before the anchor is persisted. For instance it could be the case that there is already an anchor for exactly this element (because another user created one in the meantime or the element identifier is not unique in this context).

In our scene everything works out fine and the web service persists the new anchor in the PostgreSQL database. The web service returns a positive status code to the plugin. This again triggers the previously discussed update and matching procedure. Alice will see her comment box attached to the appointment after it is guaranteed to be persisted on the server. It is not possible that the plugin creates locally new anchors that do not exist on the server.

Finally it is possible for Alice to use the comment box. This step is very simple. As we already mentioned the comment box is an external service that is only injected by a link into our system. Therefore Alice can add a comment without any consequences to our system at all.

Now it is Bobs turn. This process is quite similar and partially redundant to what happened when Alice created the anchor. For that reason we will not go into detail like we did for the first step.

Bob opens the Google Calendar website, which triggers the update and matching procedure for this URL. Since there is an existing anchor now - Bob's plugin receives the data for displaying the comment box entered by Alice.

The last two steps are getting more interesting again. We basically simulate a evolution of a website that destroys our anchor mechanism. That can happen very easily depending on the type of script we are using or how fast the webpage evolves, but this issue will be discussed more deeply in the coming section (Referenceessowe-assessment 5.3). What we do is to modify the element identifier directly in the database. This way it becomes impossible to match this anchor for the given URL.

So Alice visits her calendar to check whether Bob has answered her question. Again an update and matching procedure is started. The update will work seamlessly but an error occurs while the matching progresses. The plugin runs

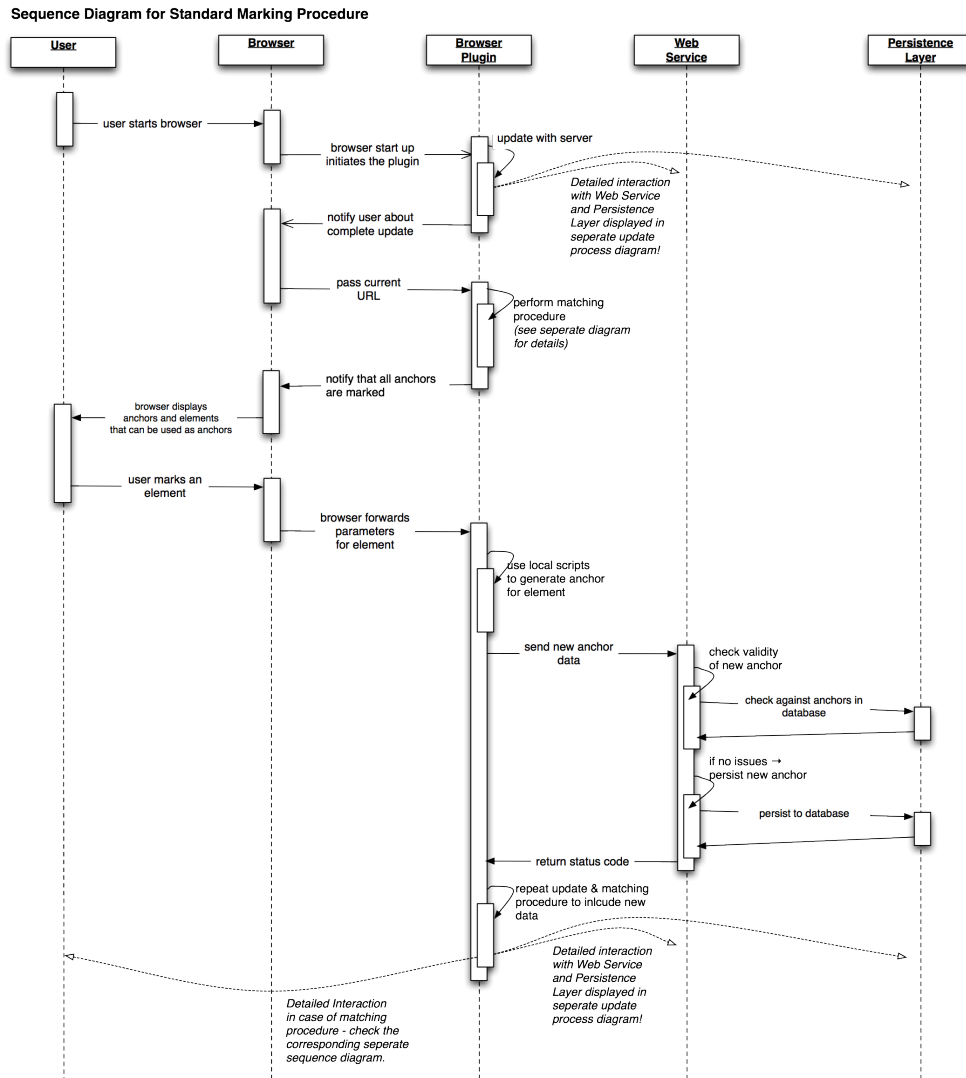


Figure 19: Sequence diagram for standard marking procedure

the script to determine the element that belongs to the element id - but with no success.

For that case the plugin enables Alice to relink the content to the correct element or as in this case - appointment. The plugin performs the two following steps:

1. Create a new anchor element, that is basically a copy to the old one but with a correct element-id. This step is identical to the above described procedure when Alice weaved the comment box into an appointment the

first time.

2. Additionally to the first step, we need to remove the broken anchor from the server. This is done by sending a remove Rest request to the server including the old element identifier.

After those steps are finished, it is necessary to run the update and matching procedures again. Now Alice and Bob are able once again to use the comment box.

Re-linking an anchor does not necessarily has to be due to an error or webpage evolution. For instance if Bob changed the time of the appointment - the anchor would not work either. In this case a re-link would solve the problem as well.



## 5.2 Social Weaver Scripts in Action

### 5.3 Social Weaver Assessment

In the following we will analyse how good Social Weaver will work in several real scenario cases. Since it is developed as a proof-by-concept prototype, a general support for all web sites or web application was out of reach. Anyway the script support allows us to reach at least some flexibility. The testing range should cover static and more dynamic websites. Furthermore some freely available web applications will be tested. We will distinguish some criteria:

- **Level of Marking Support**  
This criterion is the ability of the plugin to recognize elements in a web view. This means first of all that all relevant elements should be recognized. The best case would that elements like advertisements or scrolling bars would be left out. Still all buttons, form elements and similar elements would be spotted. This criterion is not purely objective since relevant elements may differ for each user.
- **Level of Matching Support**  
Matching Support describes the ability of the plugin to find element that were previously marked. Even though this is at least as important as the Marking Support, there is no guarantee that matching will be handled equally well as marking. For instance if we match an element only by its path in the DOM tree; This path might be ambiguous to another element. In this case our social element would be weaved into the wrong place. We consider this as the worst case even worse as if no element could have been matched.
- **Level of Anchor Reliability**  
Anchor Reliability can be seen as part of the matching criterion. But with reliability we refer to the time relevant aspect. With the evolution of a web page, our anchor information might become obsolete. The chance for this to happen is increasing with time. News pages are the best example for a very fast evolution. An anchor attached to an article on the frontpage would not last more than a couple of days. But even on such a dynamic web page there mostly are elements that are more reliable (e.g. the search column or navigation bars).  
  
This criterion should evaluate how probably it is that anchors will outlast time.
- **Expense**  
Expense in this context means how much effort has been used to give support for the tested environment. The extent of the script itself and an appraisal how tricky the construction of the script is, whether just standard procedure has been used or if it was necessary to insert some hacks.

## 6 Discussion

## **7 Future Work**

## **8 Conclusion**

In the abstract concept level we learned what a Social Weaving system needs to provide and what problems might appear. We defined the requirements and roughly an architecture. Using this knowledge in the second part we described an implementation of a prototype on a more detailed and technical level. This should be seen as a proof of concept that Social Weaving is basically possible.

## 9 Defintions

**Anchor** is an information container that includes data to determine an element in a web view. It also includes data for the social web feature that will be weaved to the element.

**Annotation** see Anchor

**Marks** are the visual representation for an anchor. An element that has an anchor will show a mark in form of a rectangle, or colored background.

**Social Weaving**

**Social Weaver**

# Appendices

Use Cases Use Cases In diesem Abschnitt werden Use Cases aufgefñhrt, die aus den genannten Anforderungen abgeleitet werden konnten.

## .0.1 Akteure

- User  
Common user who uses the plugin to use Social Weaver.
- Plugin  
In context of the thesis the firefox plugin mentioned in section 4.1 Social Weaver - Firefox Plugin.
- Server Service

## .0.2 Use Cases

UseC. A. User can mark a web element for annotation

- Use Case  
User should be able to see what elements in the web view are annotatable. In case his cursor moves above a annotatable element it should be visually marked.
- Initiator  
User who is performing an interface action.
- Pre condition  
none
- Process  
// @TODO  
  
(a) ...
- After condition  
A successful marking is the precondition for an annotation action.

UseC. B. User can annotate a web element

- Use Case  
User should be able to annotate a specific web element so that we can use it as anchor in the future.

- Initiator  
User who is performing an interface action.
- Pre condition  
UseC A. is the precondition for this Use Case.
- Process  
// @TODO  
  
(a) ...
- After condition  
A successful annotation is the precondition for a visualization of an annotation object.

#### UseC. C. Plugin can display annotated elements

- Use Case  
Already annotated web elements in a view should be recognized by the plugin and signals shown to the user where to find which annotations.
- Initiator  
Indirectly by a user who opens a view, which triggers the matching process of the plugin.
- Pre condition  
Already existing annotated elements that might be displayed.
- Process  
// @TODO  
  
(a) ...
- After condition  
none

#### UseC. D. Plugin can send Annotations to Server

- Use Case  
...
- Initiator  
...
- Pre condition  
...
- Process  
...  
(a) ...
- After condition  
...

- Alternative
- ...

UseC. E. Plugin can retrieve Annotations from Server Service

- Use Case
- ...
- Initiator
- ...
- Pre condition
- ...
- Process
- ...
- (a) ...
- After condition
- ...
- Alternative
- ...

UseC. F. Server Service can retrieve Annotations from Plugin

- Use Case
- ...
- Initiator
- ...
- Pre condition
- ...
- Process
- ...
- (a) ...
- After condition
- ...
- Alternative
- ...

UseC. G. Server Service can send Annotations Updates to Plugin

- Use Case
- ...
- Initiator
- ...
- Pre condition
- ...



- Process
  - ...
  - (a) ...
- After condition
  - ...
- Alternative
  - ...

## References

- [1] Claus Brabrand, Robert Giegerich, and Anders Møller. *Analyzing ambiguity of context-free grammars*. Springer, 2007.
- [2] IEEE Computer Society. Software Engineering Standards Committee and IEEE-SA Standards Board. Ieee recommended practice for software requirements specifications. Institute of Electrical and Electronics Engineers, 1998.
- [3] World Wide Web Consortium et al. Document object model (dom), 2001.
- [4] Dirk Draheim, Christof Lutteroth, and Gerald Weber. Generator code opaque recovery of form-oriented web site models. In *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, pages 302–303. IEEE, 2004.
- [5] Dirk Draheim, Christof Lutteroth, and Gerald Weber. *Source Code Independent Reverse Engineering of Dynamic Web Sites*. Freie Univ., Fachbereich Mathematik und Informatik, 2004.
- [6] Dirk Draheim, Christof Lutteroth, and Gerald Weber. A source code independent reverse engineering tool for dynamic web sites. In *Software Maintenance and Reengineering, 2005. CSMR 2005. Ninth European Conference on*, pages 168–177. IEEE, 2005.
- [7] Thomas Erl. *Service-Oriented Architecture A Field Guide to Integrating XML and Web Services*. Prentice Hall, 2004.
- [8] Thomas Erl. *SOA Principles of Service Design*. Prentice Hall, 2008.
- [9] Roy T Fielding and Richard N Taylor. Principled design of the modern web architecture. In *Proceedings of the 22nd international conference on Software engineering*, pages 407–416. ACM, 2000.
- [10] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
- [11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns. 1995. *Reading, Massachusetts: Addison-Wesley. ISBN 0-201-63361-2*.
- [12] Michael A Harrison. *Introduction to formal language theory*. Addison-Wesley Longman Publishing Co., Inc., 1978.
- [13] Mario Jeckle, Chris Rupp, Jrgen Hans, Barbara Zengler, and Stefan Queins. *UML 2 Glasklar*. Hanser, 2004.
- [14] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, August 1988.

- [15] Avraham Leff and James T Rayfield. Web-application development using the model/view/controller design pattern. In *Enterprise Distributed Object Computing Conference, 2001. EDOC'01. Proceedings. Fifth IEEE International*, pages 118–127. IEEE, 2001.
- [16] T. Lethbridge and R. Laganriere. *Object-oriented software engineering*. McGraw-Hill Higher Education, 2001.
- [17] Peter Liggesmeyer. *Software-Qualitt*. Spektrum, 2002.
- [18] Joe Marini. *Document Object Model*. McGraw-Hill, Inc., 2002.
- [19] James McGovern, Oliver Sims, Ashish Jain, and Mark Little. *Enterprise Service Oriented Architectures*. Springer, 2006.
- [20] Ian Sommerville. *Software Engineering*. Pearson Education Limited, 2001.
- [21] A. Van Lamsweerde. Requirements engineering: from system goals to uml models to software specifications. 2009.
- [22] K.E. Wiegers. Software requirements. 2003.
- [23] Pree Wolfgang. *Design patterns for object-oriented software development*. Reading, Mass.: Addison-Wesley, 1994.