UNIVERSITY OF INNSBRUCK

MASTER SEMINAR

Social Weaver
A Prototype for Weaving Web 2.0 Features into Web Applications

Author:
Viktor PEKAR

Supervisor:
Dr. Michael FELDERER

Collaboration:
Dr. Dirk DRAHEIM

April 28, 2013

Contents

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 paper is about Social Weaving, a new technique that combines modern communication methods with existing web sites and web applications. The goal is to have 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 invitation 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 unfortunately this is the greatest challenge.

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 (3) is an abstract requirement analysis that describes what a Social Weaving system needs. The third part (section 4) show the architecture of the prototype on a more concrete level. Finally the above mentioned problem about the lack of unique identifiers for web elements will be discussed and compared to other technologies where such identifiers are being used with great success.

2 Contribution

In the last couple of years the internet developed into a mass medium. It started with 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 message 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 the way of how human communicate are still in development. The problem is that we see communication as something we were practicing since we exist. 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. In 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. 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 optimal at all! What if other colleagues might have something to add to the conversation, but are not in included in the receivers list? If it is just a short question, the way through 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? The would have to ask the same question again.

So what we want is 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 a comment box beneath the button would solve the problem. Or a link that leads to a discussion forum - just for the button. But besides that both solutions bring a bunch of disadvantages, it would require to modify the web application that includes our favorite button. So lets drop these possibilities. 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.

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 Requirements Analysis

3.1 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.

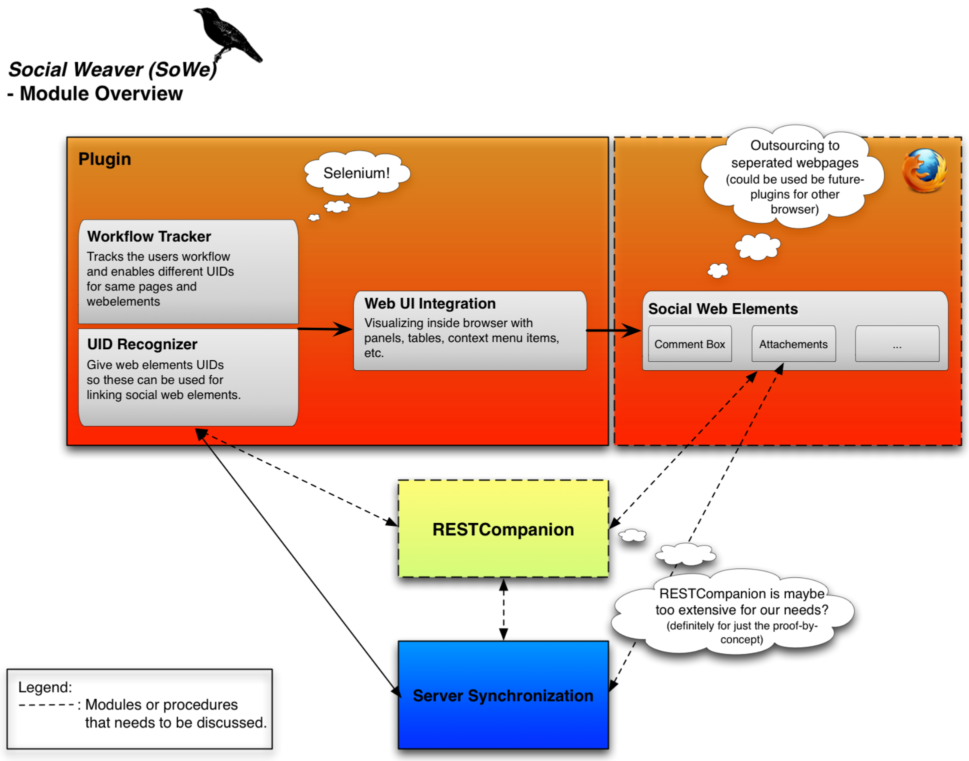

Social Weaver (SoWe)
- Module Overview
Plugin
Selenium!
Workflow Tracker
Tracks the users workflow and enables different UIDs for same pages and webelements
UID Recognizer
Give web elements UIDs so these can be used for linking social web elements.
Web UI Integration
Visualizing inside browser with panels, tables, context menu items, etc.
Outsourcing to seperated webpages (could be used be future-plugins for other browser)
Social Web Elements
Comment Box
Attachements
...
RESTCompanion
RESTCompanion is maybe too extensive for our needs? (definitely for just the proof-by-concept)
Server Synchronization
Legend:
------: Modules or procedures that needs to be discussed.

Figure 1: 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 2.

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

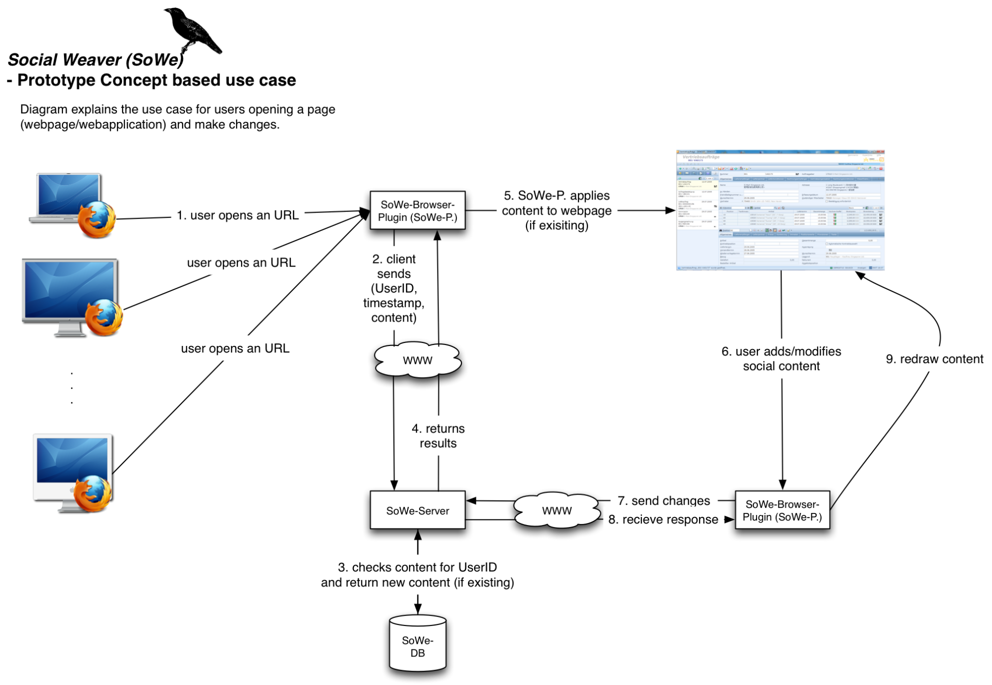

Social Weaver (SoWe)
- Prototype Concept based use case
Diagram explains the use case for users opening a page (webpage/webapplication) and make changes.
1. user opens an URL
user opens an URL
user opens an URL
SoWe-Browser-Plugin (SoWe-P.)
5. SoWe-P. applies content to webpage (if exisiting)
2. client sends (UserID, timestamp, content)
WWW
4. returns results
6. user adds/modifies social content
9. redraw content
SoWe-Server
WWW
7. send changes
8. recieve response
SoWe-Browser-Plugin (SoWe-P.)
3. checks content for UserID and return new content (if existing)
SoWe-DB

Figure 2: Social Weaver Prototype Use Case

3.2 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:

3.2.1 Browser Plugin

In the following we define requirements on a abstract domain level according to [13]. A specification to a concrete domain level will follow in section 4.4, 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: 3.2.2). 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

3.2.2 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

3.2.3 Communication Protocol

3.3 Technologies

This section should provide a short overview about the technologies that are being used for Social Weaver.

- Firefox plugin API
- REST as Web service Interface
- JavaEE for server application
- PostgreSQL for persistence layer
- JSON for formatting data

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.

4.2 Firefox

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

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.

4.3 Firefox Plugin Development

To improve readability of the coming section 4.4 Requirements for the Plugin, we will discuss some aspects from the Mozilla Add-on SDK (Version 1.13) [5]. Readers who are not interested to much into technical detail can skip this section.

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*[6] 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 3) and to integrate our social media web elements.

[1] `www.mozilla.org`

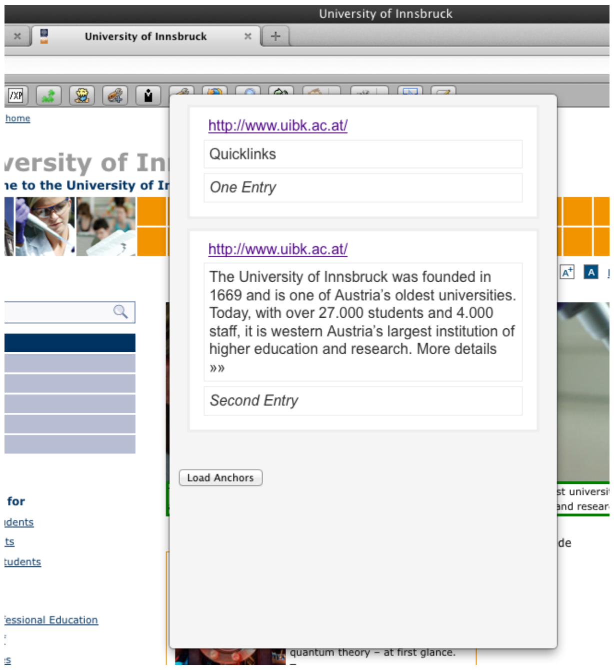
[2] `http://www.mozilla.org/MPL/1.1/`

[3] `http://www.gnu.org/licenses/lgpl-3.0.de.html`

[4] `http://www.gnu.org/licenses/gpl-3.0.html`

[5] `https://addons.mozilla.org/en-US/developers/docs/sdk/latest`

[6] `https://addons.mozilla.org/en-US/developers/docs/sdk/latest/modules/sdk/panel.html`



http://www.uibk.ac.at/

Quicklinks

One Entry

http://www.uibk.ac.at/

The University of Innsbruck was founded in 1669 and is one of Austria's oldest universities. Today, with over 27.000 students and 4.000 staff, it is western Austria's largest institution of higher education and research. More details »»

Second Entry

Figure 3: An example for a *panel* that shows a list of anntations

Simple-Storage

This module[7] is an easy to use method to store basic properties (booleans, numbers, strings, arrays, ...) across browser restarts.

With an operation like

```
var ss = require("sdk/simple-storage");
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.

Page-Mod

The *page-mod*[8] module enables us to act in a specific context related to

[7]`https://addons.mozilla.org/en-US/developers/docs/sdk/latest/modules/sdk/simple-storage.html`

[8]`https://addons.mozilla.org/en-US/developers/docs/sdk/latest/modules/sdk/page-mod.html`

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* [9] is simply an interface to the Firefox add-on bar[10]. It is possible to attach *panels* and trigger operations by clicking the *widget*.

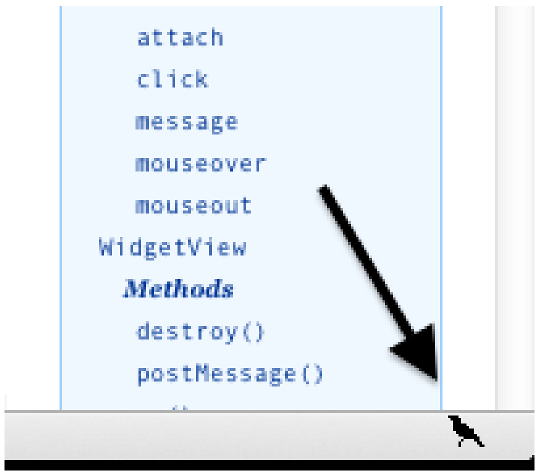


Figure 4: Example for a widget serving as activation button

We will use a widget to switch between different modes and to display an overview (see screenshot 4).

Self

Self [11] provides access to add-on specific information like the Program ID[12], 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*.

Notifications

[9] `https://addons.mozilla.org/en-US/developers/docs/sdk/latest/modules/sdk/widget.html`

[10] `https://developer.mozilla.org/en-US/docs/The_add-on_bar`

[11] `https://addons.mozilla.org/en-US/developers/docs/sdk/latest/modules/sdk/self.html`

[12] `https://addons.mozilla.org/en-US/developers/docs/sdk/latest/dev-guide/guides/program-id.html`

This module[13] displays toaster[14]-messages that disappear after a short time.

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

Request

This simple to use but yet powerful module *request*[15] 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[7]. The response from a server is directly accessible like a JavaScript object.

We are going to use *request* for our communication with our synchronization web service.

JQuery

jQuery[16] is a free JavaScript library under the MIT License[17] that offers many functions for modifying DOM trees. It has been released 2006 in context of a BarCamp[18] 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 the HTML code (like changing the background color of web elements) is being reached with jQuery.

4.4 Requirements for the Plugin

Let us recap what requirements we gathered in section 3.2.1 on the abstract domain level [13]:

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

[13]`https://addons.mozilla.org/en-US/developers/docs/sdk/latest/modules/sdk/notifications.html`

[14]`http://en.wikipedia.org/wiki/Toast_(computing)`

[15]`https://addons.mozilla.org/en-US/developers/docs/sdk/latest/modules/sdk/request.html`

[16]`http://jquery.com/`

[17]`http://www.mit.edu/`

[18]`http://en.wikipedia.org/wiki/BarCamp`

7. Identifying web elements across different user sessions

In the following specialization we apply the abstract requirements to our environment which is the *Mozilla Plugin Development SDK* [19].

4.4.1 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 5). *Selecting* in this context means that we analyze the Document Object Model (DOM)[20] tree and retrieve its *id* from the closest DOM ancestor. This is easy to implement using the function *mouseenter* from the JavaScript [21] library jQuery [22].

Now that we have located a specific web element we may annotate our comment box. For reasons of flexibility and simplicity we just annotate a HTML window (check 1.2 in figure 5), where we inject an external comment box (but basically every HTML-code is going to work). The Mozilla SDK high-level APIs [23] offer all necessary tools to insert a HTML box as a *Panel*[24].

The annotation anchors will be visible to the user in form of a colored background rectangle that we create by modifying the DOM tree. While the user moves the cursor above the web elements, while the plugin is activated, all elements that are annotatable will be marked with such a rectangle (see screenshot 6).

Clicking on an element that is supported will append the above mentioned HTML panel. After that annotated elements will be marked with the rectangle. If the user clicks on such an element the already existing comment box will be reopened.

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 5). The high level API *simple-storage* [25] enables us to store all information we need and recall them. The synchronization mechanism should just modify this data set. All displaying procedures should be outside of server communication reach.

The last sub requirement is to redisplay existing annotations (check 1.4 in figure 5) 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

[19]`https://addons.mozilla.org`

[20]`http://www.w3.org/DOM/`

[21]`https://developer.mozilla.org/en-US/docs/JavaScript`

[22]`http://jquery.com/`

[23]`https://addons.mozilla.org/en-US/developers/docs/sdk/latest/modules/high-level-modules.html`

[24]`https://addons.mozilla.org/en-US/developers/docs/sdk/latest/modules/sdk/panel.html`

[25]`https://addons.mozilla.org/en-US/developers/docs/sdk/latest/modules/sdk/simple-storage.html`

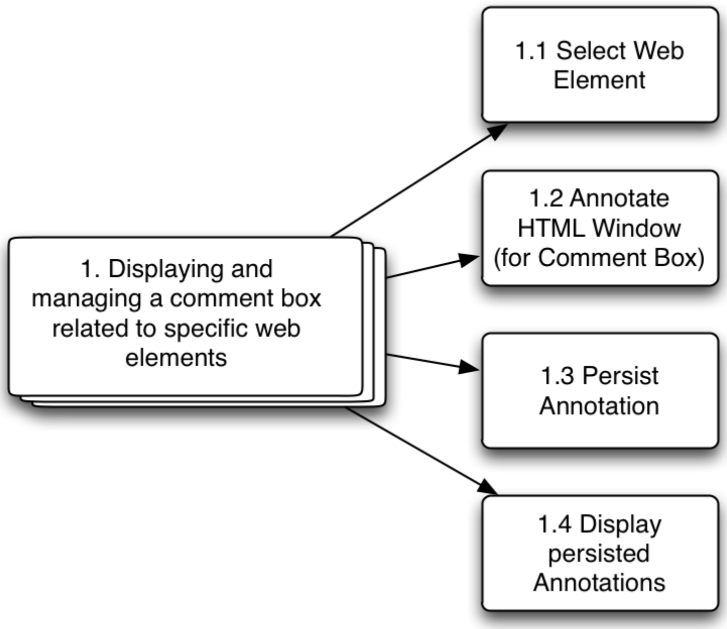
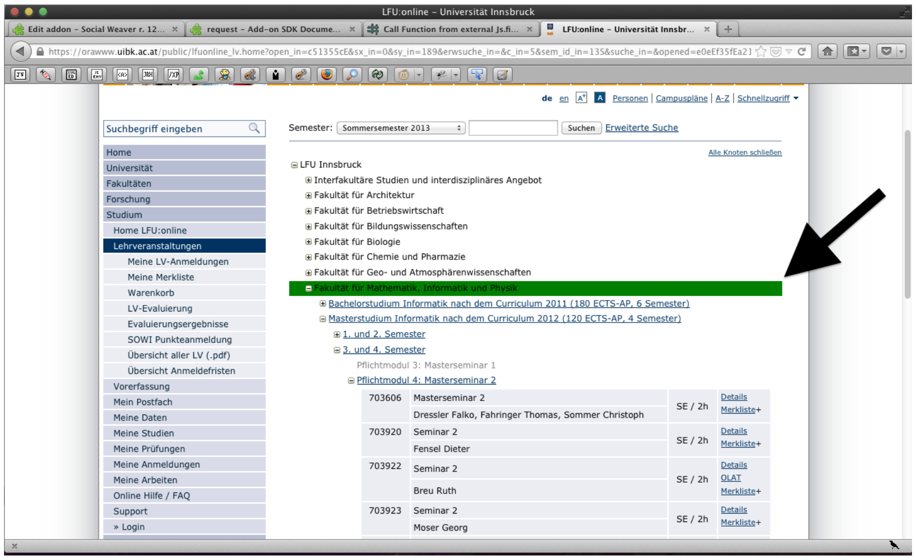

1.1 Select Web Element
1.2 Annotate HTML Window (for Comment Box)
1. Displaying and managing a comment box related to specific web elements
1.3 Persist Annotation
1.4 Display persisted Annotations

Figure 5: Partition of the first plugin requirement to sub requirements


LFU:online – Universität Innsbruck
Edit addon – Social Weaver r. 12...
request – Add-on SDK Docume...
Call Function from external Js.fi...
LFU:online – Universität Innsbr...
https://orawww.uibk.ac.at/public/lfuonline_lv.home?open_in=c51355cE&sx_in=0&sy_in=189&erwsuche_in=&c_in=5&sem_id_in=135&suche_in=&opened=e0eEf35fEa21
de en Personen | Campuspläne | A-Z | Schnellzugriff
Suchbegriff eingeben
Semester: Sommersemester 2013
Suchen
Erweiterte Suche
Alle Knoten schließen
Home
Universität
Fakultäten
Forschung
Studium
Home LFU:online
Lehrveranstaltungen
Meine LV-Anmeldungen
Meine Merkliste
Warenkorb
LV-Evaluierung
Evaluierungsergebnisse
SOWI Punkteanmeldung
Übersicht aller LV (.pdf)
Übersicht Anmeldefristen
Vorerfassung
Mein Postfach
Meine Daten
Meine Studien
Meine Prüfungen
Meine Anmeldungen
Meine Arbeiten
Online Hilfe / FAQ
Support
» Login

⊟ LFU Innsbruck
⊞ Interfakultäre Studien und interdisziplinäres Angebot
⊞ Fakultät für Architektur
⊞ Fakultät für Betriebswirtschaft
⊞ Fakultät für Bildungswissenschaften
⊞ Fakultät für Biologie
⊞ Fakultät für Chemie und Pharmazie
⊞ Fakultät für Geo- und Atmosphärenwissenschaften
⊟ Fakultät für Mathematik, Informatik und Physik
⊞ Bachelorstudium Informatik nach dem Curriculum 2011 (180 ECTS-AP, 6 Semester)
⊟ Masterstudium Informatik nach dem Curriculum 2012 (120 ECTS-AP, 4 Semester)
⊞ 1. und 2. Semester
⊟ 3. und 4. Semester
Pflichtmodul 3: Masterseminar 1
⊟ Pflichtmodul 4: Masterseminar 2

703606	Masterseminar 2 Dressler Falko, Fahringer Thomas, Sommer Christoph	SE / 2h	Details Merkliste+
703920	Seminar 2 Fensel Dieter	SE / 2h	Details Merkliste+
703922	Seminar 2 Breu Ruth	SE / 2h	Details OLAT Merkliste+
703923	Seminar 2 Moser Georg	SE / 2h	Details Merkliste+

Figure 6: Rectangle shows that the underlying element is possible for annotation

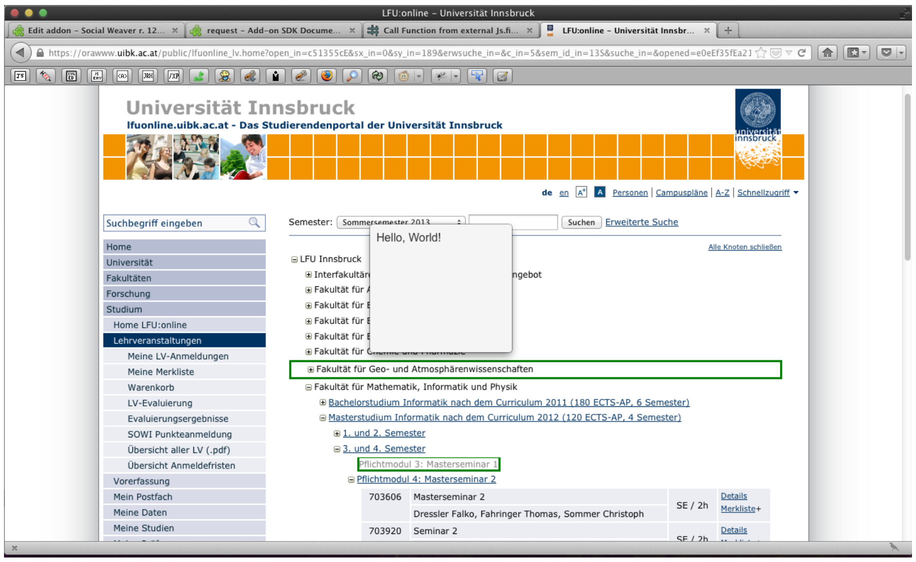

LFU:online – Universität Innsbruck
Universität Innsbruck
lfuonline.uibk.ac.at - Das Studierendenportal der Universität Innsbruck
Suchbegriff eingeben
Home
Universität
Fakultäten
Forschung
Studium
Home LFU:online
Lehrveranstaltungen
Meine LV-Anmeldungen
Meine Merkliste
Warenkorb
LV-Evaluierung
Evaluierungsergebnisse
SOWI Punkteanmeldung
Übersicht aller LV (.pdf)
Übersicht Anmeldefristen
Vorerfassung
Mein Postfach
Meine Daten
Meine Studien
Semester:
Suchen
Erweiterte Suche
Hello, World!
Alle Knoten schließen
LFU Innsbruck
Fakultät für Geo- und Atmosphärenwissenschaften
Fakultät für Mathematik, Informatik und Physik
Bachelorstudium Informatik nach dem Curriculum 2011 (180 ECTS-AP, 6 Semester)
Masterstudium Informatik nach dem Curriculum 2012 (120 ECTS-AP, 4 Semester)
1. und 2. Semester
3. und 4. Semester
Pflichtmodul 3: Masterseminar 1
Pflichtmodul 4: Masterseminar 2
703606 Masterseminar 2
Dressler Falko, Fahringer Thomas, Sommer Christoph
SE / 2h
Details
Merkliste+
703920 Seminar 2

Figure 7: Rectangle shows that the underlying element is possible for annotation

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 7).

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. 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!" 7).

4.4.2 Managing several comment boxes without disturbing the view on original content

This requirement is not directly about functionality but should assure 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.

Fortunately we already had this in mind while working on the previous requirement. Existing annotations are marked with a rectangle that is displayed around the element we use as anchor point (see screenshot 7). This way is probably not beautiful but efficient in not disturbing the structure of the application beneath.

4.4.3 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.

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[26] and REST([7]) 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. 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 4.4.3 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 `handleNewAnnotion(newAnnotationText, newAnchor)` method to store it in our *simple-storage* list.
>
> 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.

[26] `http://www.json.org/`

```
var sync = Request({
      url: 'http://localhost:9998/anchor',
      onComplete: function(response) {
        for(var i = 0; i<response.json.length; i++){
            var r = response.json[i];

            newAnchor = new Array(r.anchorURL,
                r.ancestorId, r.anchorText);
            var newAnnotationText = r.annotationText;
            handleNewAnnotation(newAnnotationText,
                newAnchor);
     };
  }
});
sync.get();
```

Figure 8: Sample JavaScript code for retrieving JSON objects from a web service with a GET REST request

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 server synchronization works in detail or how the server architecture looks like, will not be mentioned in this context. The reason is that the synchronization and data processing on the server is not directly related to Social Weaving and would be not contribution to our research.

5 Fazit

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.

References

[1] 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.

[2] 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.

[3] Dirk Draheim, Christof Lutteroth, and Gerald Weber. *Source Code Independent Reverse Engineering of Dynamic Web Sites.* Freie Univ., Fachbereich Mathematik und Informatik, 2004.

[4] 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.

[5] Thomas Erl. *Service-Oriented Architecture A Field Guide to Integrating XML and Web Services.* Prentice Hall, 2004.

[6] Thomas Erl. *SOA Principles of Service Design.* Prentice Hall, 2008.

[7] 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.

[8] Mario Jeckle, Chris Rupp, Jrgen Hans, Barbara Zengler, and Stefan Queins. *UML 2 Glasklar.* Hanser, 2004.

[9] T. Lethbridge and R. Laganiere. *Object-oriented software engineering.* McGraw-Hill Higher Education, 2001.

[10] Peter Liggesmeyer. *Software-Qualitt.* Spektrum, 2002.

[11] James McGovern, Oliver Sims, Ashish Jain, and Mark Little. *Enterprise Service Oriented Architectures.* Springer, 2006.

[12] Ian Summerville. *Software Engineering.* Pearson Education Limited, 2001.

[13] A. Van Lamsweerde. Requirements engineering: from system goals to uml models to software specifications. 2009.

[14] K.E. Wiegers. Software requirements. 2003.