UNIVERSITY OF TARTU

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

Institute of Computer Science

Computer Science

Allar Tammik

Interactive Data Sharing Mechanism for Widget-based Microsites

Master's thesis (30 ECTS)

Advisor: Peep Küngas

Author: .... "….." May 2011

Advisor: .... "….." May 2011

Approved for defence

Professor: .... "….." May 2011

Tartu 2011

Contents

Abstract

Nowadays it is very common that modern web sites exchange content between each other by means of syndication and aggregation. This is enabled through APIs, protocols, tools and platforms. The recent trend in content processing is advancing towards more extensive use of widgets along with static content. Although web sites usually use widgets to provide added value to their users, simpler web sites like microsites can be built entirely from widgets. Such kinds of widgets are usually stateless, but not necessarily autonomous. In particular they may also be able to communicate with other components in the same web application, including microsites, which are the key focus of this thesis. Because of the loose coupling, the widgets themselves are not able to capture the state of the microsite, whereas the microsite itself usually does not have a mechanism for storing its state. However, messages exchanged during communication, determine the state of the microsite.

This thesis describes a solution for storing and sharing the state of a microsite by record-and-replay mechanism for messages exchanged by widgets of microsites. Furthermore, the mechanism also allows sharing the stored state of a microsite between friends via social networks. The latter enables user-initiated inter-application content delivery and state exchange, which has been possible so far only between tightly integrated applications.

1. Introduction

In the past few years, new software systems and applications have been made available at the Web. It is common to use applications like collaborative word processors, spreadsheets and calendars directly in the browser instead of installing them to your computer. Moreover, the proliferation of end-user devices requires the content to be available in multiple locations. The web applications strive to become highly interactive and give visually rich and desktop-style user experience. These visually rich web applications are also known as Rich Internet Applications (RIAs). It is believed that in the near future the vast majority of end-user software applications will be written for the Web, instead of conventional target platforms such as specific operating systems, CPU architectures or devices [1]. More specifically, most important characteristics that pervade the technologies are collaboration, interaction and user-generated content. These characteristics are often associated with term “Web 2.0” [2]. The collaboration embraces social aspects that allow a large number of people to collaborate, share data and services in the Web. By interaction it is meant that web applications are built to behave like desktop applications, for example by allowing web pages to update one user interface element at a time by providing support for direct modification of Document Object Model (DOM). According to Gartner [3] such Web 2.0 technologies as blogs, wikis and social networks are now used by progressive mainstream businesses. Furthermore, the use of user-generated reviews and ratings is growing. However, new Web 2.0 business models continue to emerge, for example, sites exposing product information for mashup purposes via Application Programming Interfaces (APIs).

Mashups are software applications that merge separate APIs and data sources into one integrated interface [4]. Mashup development differs from conventional software development in many ways. Mashup development’s main focus is in the reuse, rather than in the implementation of a web site. Moreover, mashups are far more dynamic than conventional software components, thus they cannot be built easily using static programming languages, that require advance compilation and static type checking. What is more, being a mashup developer does not require any formal training or background in software development [1]. Instead, mashups are often developed by people gathered to communities. Gartner uses a term „citizen developer“ to describe the concept [3]. In fact, several big mashup providers have their own communities for people to develop and share their mashups.

The aim of a mashup development is to create new applications and services by combining existing sources in novel settings to deliver added value not leveraged by a single source [5]. Mashups can be categorized in many ways. For instance, by the creator, mashups fall into two categories: programmer-built mashups and user-built mashups. In case of programmer-built mashups, a programmer uses AJAX (Asynchronous JavaScript and XML) technologies to implement a web page, which uses third-party sources. For non-programmers, there are a number of drag-and-drop mashup assembly tools, typically running in browsers, which allow end-users to visually compose a mashup.

Another classification of mashups is by their target group: enterprise mashups and consumer mashups. Enterprise mashup tools link existing resources in enterprise environment within an end-user driven context. The center of attention of enterprise mashups is visualization of back-end resources. In contrast of consumer mashups, they invest more in security and availability aspects. Consumer mashups are targeted towards individuals to create mashups for private use. The main distinction between consumer mashups and enterprise mashups is that components of consumer mashups usually cannot intercommunicate, as opposed to enterprise mashups [6].

Microsite is a small and narrowcast web-site or web application. It has a well-defined target audience and definite purpose. Thus, a single-purpose mashup can be defined as a microsite. In other words, it is a web application or a mashup, which has mostly one feature, and does nothing but that one thing. For example, a mashup that is designed to help users to find rental property, only retrieves data from real estate companies and visualizes all offers on the map.

One shortcoming of current mashup technologies is that the mashup applications, which provide drag and drop user interfaces, have a lot of sophistication hidden in them. The user interfaces might abound with bells and whistles, but the code behind the graphical user interface is significantly complex. This situation can be improved by introducing widget-based approach to mashup development. The higher degree of modularity and cleaner structure of widget-based mashups will lead to better quality and faster implementation cycles compared to traditional mashup built in ad-hoc manner.

A widget is a reusable compact software component that can be embedded into a web page or application to provide a specific functionality or visualization. In terms of terminology, a notation “web widget” is occasionally used to emphasize the use of web technologies. Also word “gadget” or sometimes “badge” is used by different parties. In generic terms “widget”, “web widget”, “badge” or “gadget” refer to the same software items and therefore can be considered the same [6].

Simple widgets are usually used for visualization, such as a map display or a weather forecast. Simple widgets only take data from its sources and visualize it. In most cases these kinds of widgets are isolated and unaware of other widgets or the web application they are running in. However, more complex widgets are able to communicate with each other allowing the use of widgets as building blocks in mashup development. One technology that allows inter-widget communication is OpenAjax Hub 2.0 [7] (hereafter noted simply as OpenAjax Hub). As the name suggests it has a hub to which all the widgets can connect, and therefore are able to exchange information about their state and user interactions. This kind of information sharing makes it possible to build mashups where user interactions in one widget will have an influence on another widget. In effect, these messages exchanged by the widgets change the state of the mashup, whereas by nature, widgets are self-contained and loosely coupled in a mashup, they are not able to store their state. And neither is the web application that mashes up the widgets.

However, different mashup providers have solutions, which are running in their own runtime environments and are able to store the state of the mashup. Nevertheless, solutions that enable users to run widgets in their own web pages, beyond the provider-specific environment, are still not capable of storing the state of the mashup. Likewise, the OpenAjax Hub is no exception, since it also does not provide tools for storing the state of the widgets.

The main concern about widget and mashup platforms currently available is the limitedness of state sharing. It is either possible to mash up widgets in users’ own web pages and not be able to store their state, or to store the state of widgets on a condition that they are run in special environments. It is not possible to have both of these qualities simultaneously.

This thesis describes a solution to the problem of storing and sharing the states of widget-based mashups, especially microsites, by recording messages exchanged by widgets such that they can be replayed at other widget-based microsites. The messages are recorded by a Wookie widget [8] after adding it to the mashup. They are stored using. Wookie provides instanceable widgets and a database for persisting their state. By making mashups stateful, prospects to create mashup-based solutions will broaden. In the long run, when developing a widget-based application, it is ideologically right to use a solution designed widget-based approach in mind. The usage of Wookie facilitates the application of pure widget-based approach, where all primitives of programs are implemented in terms of widgets.

This thesis proposes a solution for mashups, which are built using widgets that are communicating with each other via the OpenAjax Hub 2.0 technology. The proposed and implemented solution for storing the state of mashups predicates on a Wookie widget, that facilitates record-and-replay functionality. This approach allows storing the state of any mashups, with minimal effort. Furthermore, this thesis introduces a novel aspect in widget-based mashups' state sharing through the use of Facebook as the wall posting capability supplements the social measurement of the proposed solution. It enables to share results of a stateful mashup by posting a link referring to the mashup on the Facebook's wall, thus enabling them to collaborate or benefit from each other's work.

The solution proposed is validated using a sample portal as a proof of concept. The proof of concept is based on the OpenAjax Hub technology and enables to record interactions between widgets. The interactions can be stored as well as shared in the Web. The proof of concept implementation demonstrates that the proposed solution allows storing and sharing the state of simple microsites.

The rest of the thesis is organized as follows. Chapter 2 gives an overview of current standards and technologies related to mashups and widgets. Chapter 3 describes a motivating scenario. Chapter 4 describes the technologies and tools used to implement the solution. Chapter 5 gives the detailed overview of the implemented widget. The utilization of the widget is discussed in Chapter 6, which gives an overview of the test application. Possible improvements and future work are discussed in Chapter 7.

2. Related Work

The lack of standards in widgets and mashups raises interoperability problems between components of different vendors. Different widget providers foster their own approaches to widget development, engendering the situation where different party widgets are not able to exchange information. Various solutions have been proposed to solve the problem.

OpenAjax Alliance [9] is an organization of vendors, open source projects and companies that are dedicated to adopt interoperable Ajax-based Web technologies. The alliance's prime objective is to help customers to be able to mix and match solutions from Ajax technology providers. The OpenAjax Hub [7] is one of the initiatives by the alliance, that provides a specification as well as an implementation for inter-widget communication. There has been a research for using the OpenAjax Hub technology to build a semantic integration platform, which allows aggregating data sent between widgets by generating new messages with combined data[10]. The solution proposed in this thesis is fully compliant with the semantic integration platform, as both are based on the OpenAjax Hub technology.

The World Wide Web Consortium has worked on a W3C Widget standard specification [11] to regulate packaging format and metadata for widgets. The standard specifies how a widget should be packaged in order to be able to run the widget with different user agents (for example in Wookie, which is used in this work for packaging the record and replay widget). Standardization of widgets is beneficial for different parties related to widgets and mashups, thus it would allow different widgets to run in multiple environments, providing end users with a wider variety of widgets. In parallel with research in the field of widgets, other widget-like solutions are covered in the research of mashup solutions.

Open Mashup Alliance [12] has developed an XML-based language for enterprise mashups. Enterprise Mashup Markup Language (EMML) is an open markup language that makes data universally readable and therefore increases the interoperability of mashup solutions and improves mashup portability of mashup designs. Mashups written in EMML can be deployed to any EMML-compatible platform [13].

Web Mashup Scripting Language (WMSL) enables end-users to write mashups to integrate two or more web services on the Web without needing any other infrastructure. The

WMSL, combined with HTML and JavaScript, makes it possible to easily build lightweight ontologies containing local semantics of a web service and its data model [14].

These mashup languages have not systematically examined the aspect of storing the state of mashups, thus, that situation can be improved. Since these mashup languages allow the use of JavaScript, the solution proposed in this thesis for storing the state of mashups could be easily used with these mashup languages. Even more, with some effort the functionality could be built into these languages as primitives.

There is a range of mashup providers available. Many of these tools are still under development reflecting the rapidly evolving state of art in mashup development [1]. Some major enterprise mashup products are IBM Mashup Center, Kapow Mashup Server, JackBe Presto, Morfeo EzWeb and Morfeo FAST. IBM Mashup Center [15] is a catalogue of feeds and widgets. It supports feed generation from different sources, like XML, SQL queries or spreadsheets. Kapow Technologies [16] has a product called Kapow Mashup Server, which is a commercial adapter for information access, augmentation and scraping off web-based information. Its key feature is to convert web pages into data sources, which can be used in mashups. JackBe Presto[17] is another enterprise mashup solution that allows publishing of web services and provides collaboration and execution on the presentation layer. JackBe also has solutions in security, administration and management capabilities. In general, enterprise mashup environments provide their own runtime environment [18].

The list of major consumer mashup providers includes Google, Yahoo!, Microsoft, Netvibes and Intel can be listed. Google's product iGoogle [19] is a presentation tool that aims to centralize all personal information in a personalized home screen. It is capable of combining RSS feeds and Google gadgets. Yahoo! Pipes [20] provides a graphical tool for building applications that aggregate web feeds. It has a drag-and-drop user interface that allows specifying data input, interconnecting widgets through "pipes" and specifying data output format. Microsoft has a product called Popfly, which has an editor as well as repository. It has a Silverlight-based user interface and its own community for sharing mashups with others [18]. Netvibes [21] is a tool that enables users to assemble widgets, feeds, social networks, email, videos and blogs on one fully-customizable page [18]. Intel MashMaker [22] is an interactive tool for editing, querying, manipulating, and visualizing semi-structured data. It is a browser extension that allows augmenting web pages with third-party sources directly in the browser combining qualities from concepts such as word processors, web browsers and spreadsheets. The goal of MashMaker is to allow creating mashups based on data provided by other users and remote sites.

There are a few proposed solutions in the field of mashups that are not run by big companies. One such end-user programming tool is Marmite [23]. Marmite is currently implemented as a Firefox plug-in using JavaScript and XML User Interface Language (XUL). Marmite's innovation stands in the use of algorithms that are able to automatically infer structures in unstructured text and facilitate data extraction from web pages.

Another such tool is MARGMASH [24] (from "marginal mashup"). It is a framework that allows end-user to add mashup fragments to their favorite web sites. It behaves as a proxy that redirects the request to the web site the user wishes to "margmash". However, it alters the URLs in the response to point them to MARGMASH by modifying the DOM structure. Therefore all the latter interactions are conducted through MARGMASH.

Ousia Weaver [25] is a mashup editor that provides a sophisticated visual editor allowing users to create mashups without writing any code. The mashup creation in Ousia Weaver consists of two phases. In the data phase, a user creates dataflows, which represent the rules of collecting, combining and processing data. The dataflow is drawn by dragging the dataflow components, for example a user input or a visualization widget, in the form of directed graph as characteristical to dataflows. In the visualization phase, a user can define how the data will be visualized by selecting desired visualization widgets provided by Ousia Weaver. The mashup server is built using Python, however the visualizer uses JavaScript.

MyWiWall is a widget portal that allows inter-widget communication by providing a drag-and-drop solution. The solution, which is based on W3C Widgets 1.0 family of specifications, provides widget-to-widget drag-and-drop operations. It has a widget engine written in PHP and MySQL and client-side JavaScript widget container [26].

The problem with all these tools is that they do not have out of the box functionality for saving the state of the mashups the user creates. In some cases it can be implemented. To do so, additional coding is required. Nevertheless, if these tools, which allow intercommunication between widgets, were equipped with messaging ability, like the

OpenAjax Hub widgets, it would be possible to store the state of these mashups using the solution provided in this thesis without additional coding.

There are a few widget containers that make widgets available for any web site, not just in particular mashup platform. Apache Shindig [27] is a reference implementation to OpenSocial API specification and provides infrastructure to host OpenSocial widgets on third-party web applications. It consists of a JavaScript library and a backend server for hosting OpenSocial compatible widgets. Another goal of Apache Shindig is to be language neutral and cover multiple languages. Currently, Java and PHP versions are available [27].

In addition to web widgets, widgets are also available for different platforms and devices, including mobile devices and operating systems, such as Microsoft Windows. To run widgets in different platforms, special user agents are needed. User agent is a software application that hosts an initiated widget [28]. For example, Windows Sidebar is a user agent that was introduced in Windows Vista. As another example, Opera enables users to run Opera Widgets [29] directly in the browser. Due to the standardization in user agents and packaging, it is possible to run widgets in different user agents. There are some reported cases, where the Wookie widget is running in the Opera browser [30]. This indicates that standardization activities are beginning to consolidate. The latter indicates that W3C widget specification is maturing. The solution proposed in this thesis uses Wookie widgets.

3. Motivating Scenario

This section describes a motivating scenario to illustrate how storing a state of a mashup could enhance the user experience and interaction.

Let us consider Mary, who is looking for photo printing services. Her goal is to get a list of service providers including the location and additional information (such as prices, available formats and queue length) of each service provider. She uses the OpenAjax-Hub-enabled mashup development tools to develop a microsite, which will visualize the acquired results in a microsite. Finally, she has a mashup that consists of a listing widget, details widget, prices widget and a map widget, all of which are connected to the OpenAjax Hub. When she clicks on a service provider in the list, the listing widget publishes a message to the hub and all other widgets respond by showing information about the selected item with respect to their properties such as item details, price or location. Then she selects two suitable printing service providers from her mashup and wants to show them to John. Because the microsite has no state sharing mechanism and sending a link to the mashup would not suffice, she takes a screenshot of the mashup and sends it to John. Yet, John knows another very good photo printing service he wants to share with Mary. Thus, John gives Mary a precise description about a third photo lab, which she can add to her mashup. Now she has all the information about three best photo printing services and she wants to share them with James, who is currently using a handheld device with a limited screen area. So he is not able to perceive the needed information sent by Mary since the screenshot does not fit to the mobile device screen.

Now, let us roll back to the initial mashup Mary had with two items selected and imagine that instead of sending a screenshot, she can send a link to the very same mashup in the microsite. John opens the link and sees those two selected photo printing services with all the related data and map. Moreover, he can add the third photo lab himself and Mary can see it immediately (as illustrated in Figure 1). This is possible due to an invisible Wookie widget what was recording all the messages representing modifications of the state of the mashup and now replays them back to the hub.

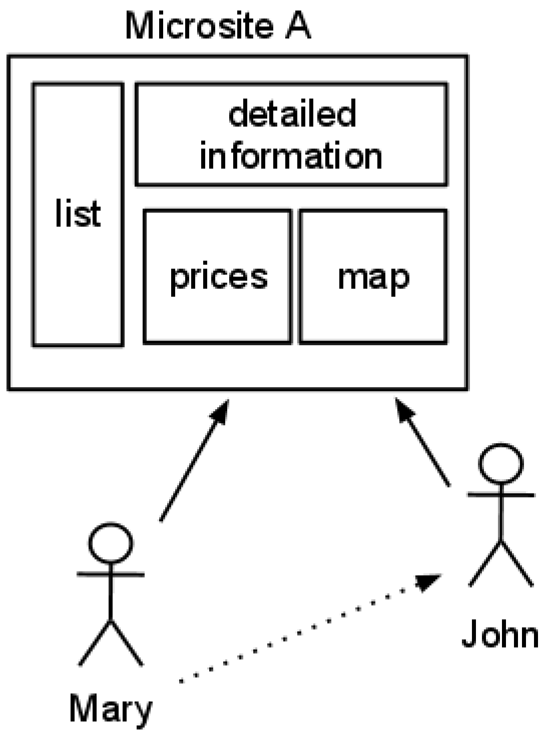


Figure 1. Sharing a state of a microsite.

James is not able to get all the information he needs from the microsite with his PDA, because the mashup in Microsite A does not fit to the device's screen, as did not the screenshot of it. With the possibility to send James a link to a microsite, he can load data from Mary's Microsite A to Microsite B. Microsite B is optimized for smaller screens showing only important information, which is at the moment the map and details of the selected photo labs (as illustrated in Figure 2). This is again possible due to the invisible widget, which recorded all OpenAjax Hub messages at Microsite A while Mary and John were interacting with it and replays them at Microsite B when James loads the recorded interactions. The latter is feasible if the widgets in microsites A and B are built by using the same framework, such as OpenAjax Hub and feature the widgets, which are able to handle the semantics of particular messages.

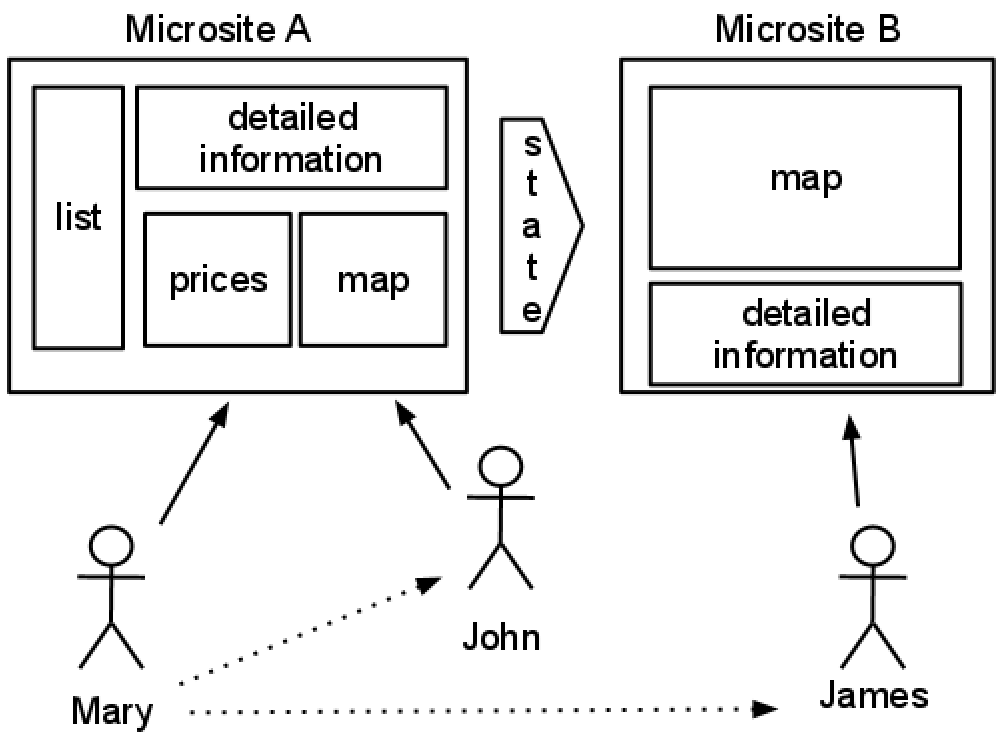


Figure 2. Sharing a state between microsites.

The advantage of this approach is that the microsites A and B do not need to be tightly integrated. They only need a mutual conception of interpreting their input parameters alike to identify the corresponding Wookie widget. A solution applicable for Microsite A (see Figure 1) is implemented as the practical outcome of this thesis.

4. Solution Description

In order to overcome the limitations stated in the Introduction, a solution, which combines the strengths of widget-based software development approach and mashup ideology has been designed and implemented. The solution is called the Wookie-OpenAjax Hub Bridge. This section of the thesis gives an overview of the architecture of that solution. For describing the architecture of the solution, the 4+1 View Model of Architecture [31] framework is used. The 4+1 View Model of Architecture describes software architecture using five concurrent views, which cover a set of concerns. The four views are logical, process, physical and development view. Putting those four together, a fifth view called scenarios is constituted to validate and illustrate the architecture design. To tailor the model, some of the views can be omitted. Considering the size of the solution implemented in this thesis, only logical and physical architecture will be described. This section also introduces the components used in the proposed solution and integration details. The solution provided in this thesis is called the Wookie-OpenAjax Hub Bridge.

The Wookie-OpenAjax Hub Bridge is an application that helps to connect two technologies: OpenAjax messaging on one side and Wookie on the other side. Combination of those two provides a mechanism for storing the state of the microsite. This solution gives advantage over the dedicated storing solutions, which do not have a widget engine, by allowing the usage of the same data in another microsite, which is compatible with OpenAjax technologies. One major advantage of using Wookie is that Wookie widgets have instances, thus they are stateful. In other words, Wookie is appropriate for the proposed solution, because its widgets are able to run in a third party web applications as well as it provides data management capabilities for each widget instance. Moreover, the same widget instance can be included in multiple mashups and it is dynamically initializable, thus it can be invoked by mashups automatically. Additional advantage is that the state is stored in a server-side database as opposed to client-side storage, such as cookies. The latter makes it a perfect solution for sharing data between different applications and users.

OpenAjax Hub is a conjoining technology in terms of this thesis. It allows communication between components that are isolated by nature. The key aspect in the OpenAjax Hub is the message-driven intercommunication. The OpenAjax Hub is used to send information between widgets in the browser. Furthermore, it specifies the widgets' structure and defines a standardized API for sending and receiving messages.

a social dimension to the proposed solution is given via Facebook, allowing users to share their stateful mashups with others. Facebook's wall-writing mechanism is used as an environment for exchanging stored states. The Facebook integration is a part of the microsite's implementation.

4.1.Software Architecture

The logical architecture of this microsite consists of the OpenAjax Hub, a number of widgets connected to it, a Wookie Engine and a Wookie's database. The widgets communicating on the OpenAjax Hub can act in different roles – while some of them might accept user input, others might visualize something to the user. There may be invisible widgets, whose purpose is to retrieve, aggregate or store the data from the hub. The Wookie-OpenAjax Hub Widget operates as a mediator between the hub and the Wookie Engine, which stores and reads the data from the database. The logical architecture is illustrated in Figure 3.

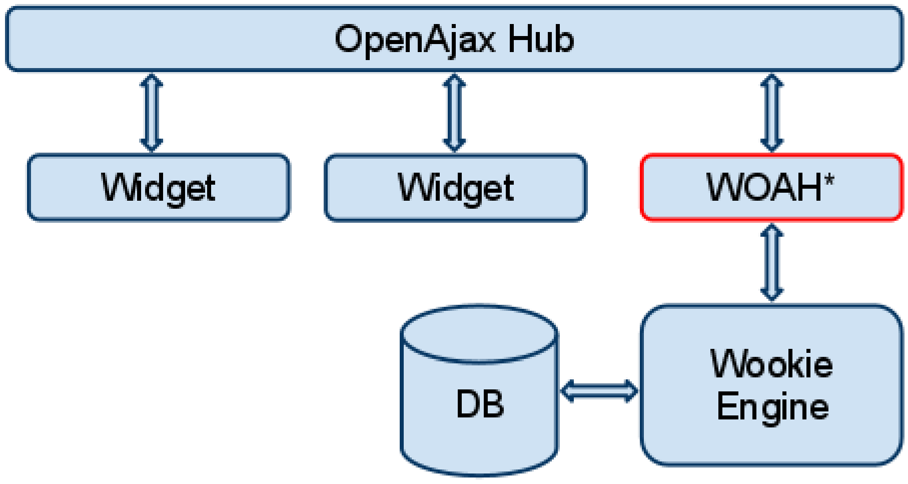


Figure 3. The logical view to the architecture. The Wookie-OpenAjax Hub Widget is marked with an asterisk (*).

To present the whole system, including the surrounding web application or the so called microsite, it is appropriate to introduce another view from the 4+1 views. The following is the physical view, which exhibits the topology of software components on the physical layer and the physical connections between these components. As shown in Figure 4, the microsite and the Wookie Engine may run in separate web servers. The microsite initiates the Wookie widget using a REST call from its JavaScript. The same-origin policy (SOP) in web browsers governs access control among different web objects and prohibits a web object from one origin from accessing web objects from a different origin [32]. According to the browsers' same-origin policy two web sites have the same origin if the originating host, port and protocol are the same for both web sites. In the example, the microsite is running on a port 8080, but the Wookie Engine is running on a port 9080. Due to the SOP, the microsite's JavaScript is not allowed to make a call to the Wookie Engine's REST API directly. To be able to make such a cross-domain request, there has to be a proxy that mediates information between domains.

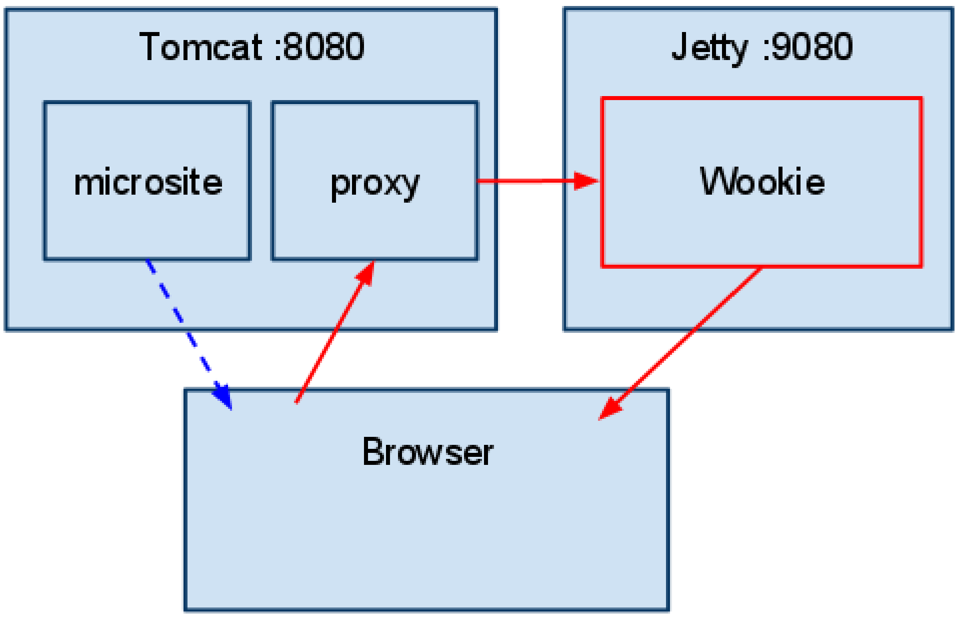


Figure 4. The physical view to the architecture.

4.2. Components

4.2.1. Wookie

Wookie is an implementation of W3C Widget specification [11] incubated in the Apache Software Foundation [33]. The Wookie Engine is a widget engine that is designed to allow widgets to be used in a number of third party web applications [34]. In technological terms The Wookie Engine is a Java server that allows user to upload and deploy widgets for user's applications. Most of these widgets are mini-applications such as a weather forecast widget or a single player puzzle-game Sudoku. However the number of fully-collaborative applications such as chats, shared to-do lists and collaborative drawing applications is growing. Wookie is based on W3C Widgets specification [11], but also widget supporting

APIs such as Google Wave [35] gadgets and OpenSocial [36]. Wookie currently has plug-ins for Elgg, Drupal, LAMS, Moodle and Wordpress to embed widgets in them [8].

Additional widgets can be added to the Wookie server using an administrator web interface or copying widget package directly to the Wookie hot deploy folder. In either case the new widget will be deployed automatically [37].

In order to use Wookie widgets in mashups, widget instance has to be created. A widget instance is a persistent instance of a particular widget. Each instance has its own storage area in Wookie's database [38]. This means that each widget instance has a capability to store its state in the Wookie's database allowing a widget to preserve user preferences. User preferences are stored as key-value pairs. Furthermore, the key is unique in the scope of widget instance and the value will be overwritten if the same key is used twice.

Widget instances can be created using the Wookie administrator web interface or the Wookie REST API [39]. A widget instance can be identified using a combination of parameters described in Table 1. When requesting a new widget instance from the Wookie REST API, an XML, as shown in Example 1, is returned to the user. The XML contains information about the widget instance location and the widget parameters. The *url* in the XML denotes the location from where the widget instance can be loaded. *Identifier* is Wookie's internal unique widget identifier. *Title*, *height* and *width* specify the title and the dimensions of the widget. Parameter *maximize* is not a part of W3C [11] specification and as pointed out in the Wookie developer's list [40] will be removed in the future. If the same parameters are used when initiating a widget instance repeatedly, the very same widget instance is returned every time.

Parameter	**Explanation**
api_key	The key issued to a particular application
shareddatakey	The key generated by an application representing a specific context (e.g. a page, post, section, group or other identified context)
userid	An identifier (typically a hash rather than a real user Id) issued by an application representing the current viewer of the widget instance
widgetid	The URI of the widget this is an instance of (optional, see servicetype below)
servicetype	Where an individual widget is not requested by URI as above, this parameter should contain the category of widget to be instantiated, e.g. "chat"
locale	The preferred locale of the widget, expressed using a BCP47 Language Tag[41]

Table 1. Parameters that can be used for identifying a widget instance, taken from [39].

```
<widgetdata>
	<url>URL TO ACCESS WIDGET</url>
	<identifier>IH6rjs75tkb6I.pl.k0hUq7YdnFcjw.eq.</identifier>
	<title>Weather</title>
	<height>125</height>
	<width>125</width>
	<maximize>false</maximize>
</widgetdata>
```

Example 1. Wookie's response to widget creation request, taken from [39].

Let us assume that the widget instance is now loaded from the URL and running in the browser. The widget is written in JavaScript and HTML. In addition to the logic written by the author of the widget, the widget instance has an access to Wookie's common services. This allows the widget instance to manipulate data in its storage area. In the JavaScript global scope, there is an object `Widget`, which allows fetch and store parameters by key. As seen in Example 2, the widget instance's preference named "foo" is fetched on the first line. The second line sets preference for a key "bar" the value "foobar".

```
Widget.preferences.getItem("foo");
Widget.preferences.setItem("bar", "foobar");
```

Example 2. Getting and setting Widget preferences.

The Wookie widget is packaged into a .wgt archive, which is a valid ZIP archive. In some other W3C Widgets implementations different extensions and packaging formats are used [28]. The archive must contain one or more start files and a configuration document. Optionally it can also contain icons, digital signatures and arbitrary files [11]. The overall structure of the widget package is illustrated in Table 2.

Location	Description
/	Widget's root folder
/config.xml	The configuration document
/index.html	Start file, file name is defined in config.xml
/images	Folder that contains all the graphical content
/scripts	Folder that contains all the JavaScript source files where the widget's behavior is described.
/locales	Container of localized content

Table 2. Widget package contents, based on [11].

The root folder must contain a configuration file and a start file. The best-practice for packaging a widget advises to hold JavaScript files in a subdirectory called scripts and graphical context in subdirectory of images. The specification defines a concept of folder-based localization, which requires a container of localized content (simply a folder named “locales”) in the widget’s root directory. In turn the container of localized contents contains a subfolder for each locale. For example the correct folder structure would be “/locales/en-us” and “/locales/et”.

4.2.2. OpenAjax Hub

The OpenAjax Hub is a set of standard JavaScript functionality defined by the OpenAjax Alliance that addresses key interoperability and security issues that arise when multiple Ajax libraries or components are used within the same web page. The OpenAjax Hub is one of the primary technical contributions of OpenAjax Alliance to the Ajax community in accordance with the Alliance's mission [7].

The OpenAjax Hub is an Ajax [42] library that allows integration of multiple client-side components within a single Web application. The Hub provides a client-side framework that allows third-party widgets to co-exist within the same Web page and intercommunicate securely via the hub [43].

The key feature of the OpenAjax Hub is the publish-subscribe engine that includes a "Managed Hub" mechanism that allows a host application to isolate distrusted components into secure Containers (sandboxes as shown in Figure 5). The Managed Hub mechanism ensures that all communications between components pass through the hub’s security manager [44], which allows or denies each publish or subscribe request based on corresponding callback functions (`onPublish` and `onSubscribe`), resulting a safe integration of distrusted third party components [43].

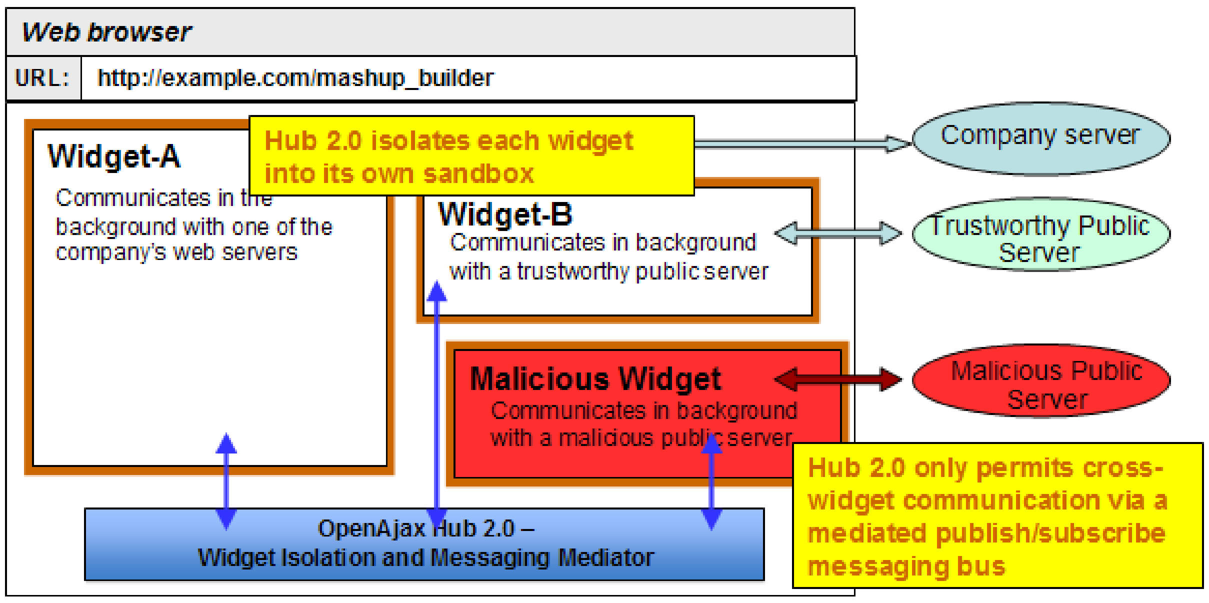

Web browser
URL: http://example.com/mashup_builder
Widget-A
Communicates in the background with one of the company's web servers
Hub 2.0 isolates each widget into its own sandbox
Widget-B
Communicates in background with a trustworthy public server
Malicious Widget
Communicates in background with a malicious public server
Company server
Trustworthy Public Server
Malicious Public Server
Hub 2.0 only permits cross-widget communication via a mediated publish/subscribe messaging bus
OpenAjax Hub 2.0 – Widget Isolation and Messaging Mediator

Figure 5. How OpenAjax Hub. Addresses the Security Challenges, taken from [44].

The OpenAjax Hub has two built-in types of containers: an IframeContainer and an InlineContainer. The IframeContainer sandboxes its content within an HTML <iframe> element, hence all the communication with the client application is marshaled across the iframe boundary. The InlineContainer places its content inside an HTML element (like <div>). In essence the widget will be in the same browser window as the Manager Application[1]. For that reason the inline container is less secure, but requires less memory and publishes messages more quickly than the IframeContainer [45].

Figure 6 explains the initialization of the OpenAjax Hub driven mashup. First the manager application has to preload the OpenAjax Hub framework library. The Managed Hub instance is created using the security manager callback functions provided by the manager application (see Example 3).

[1]In the context of the OpenAjax Hub the main HTML document for the application is denoted as the manager application [45].

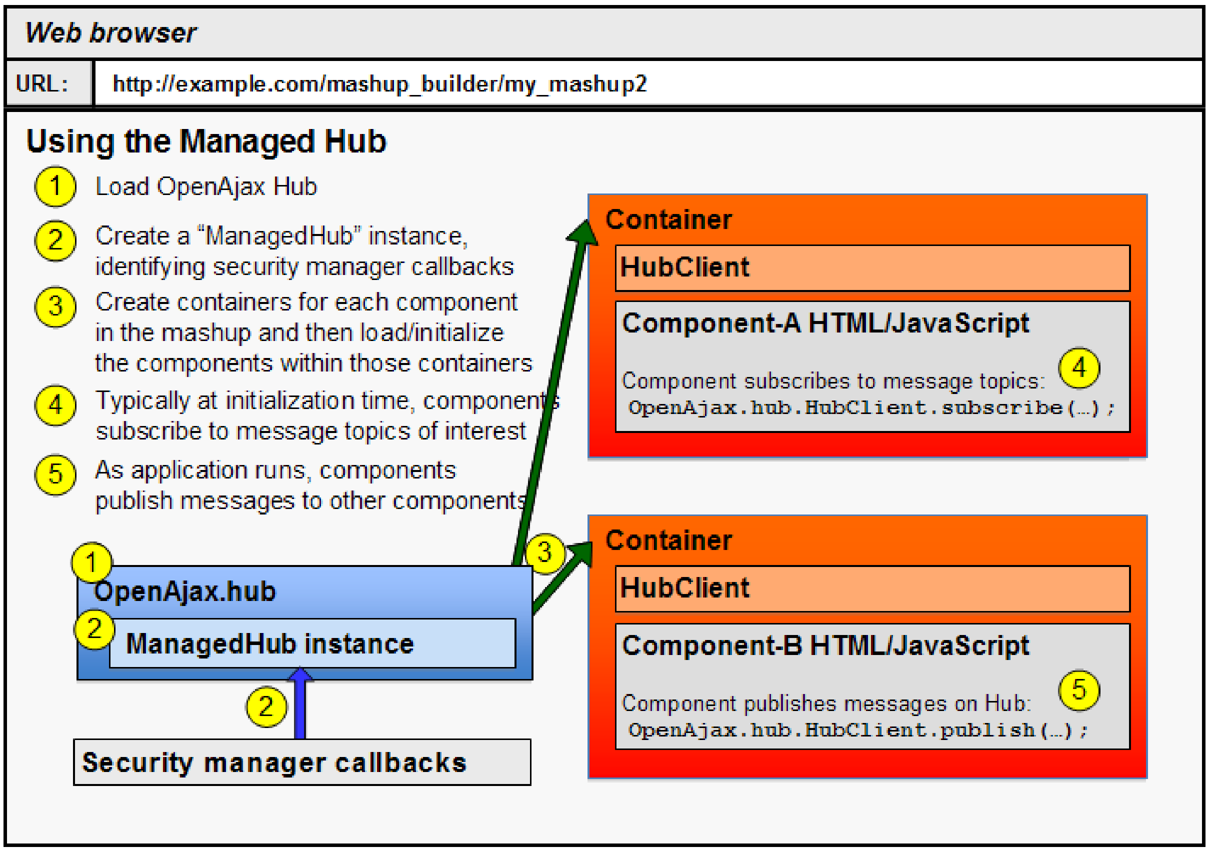

Web browser
URL:
http://example.com/mashup_builder/my_mashup2
Using the Managed Hub
1 Load OpenAjax Hub
2 Create a "ManagedHub" instance, identifying security manager callbacks
3 Create containers for each component in the mashup and then load/initialize the components within those containers
4 Typically at initialization time, components subscribe to message topics of interest
5 As application runs, components publish messages to other components
1
OpenAjax.hub
2
ManagedHub instance
2
Security manager callbacks
3
Container
HubClient
Component-A HTML/JavaScript
Component subscribes to message topics:
OpenAjax.hub.HubClient.subscribe(…);
4
Container
HubClient
Component-B HTML/JavaScript
Component publishes messages on Hub:
OpenAjax.hub.HubClient.publish(…);
5

Figure 6. Managed Hub Initialization and Usage, taken from [45].

```
var managedHub = new OpenAjax.hub.ManagedHub(
            {
              onPublish:        onMHPublish,
              onSubscribe:      onMHSubscribe,
              onUnsubscribe:    onMHUnsubscribe,
              onSecurityAlert:  onMHSecurityAlert
            }
      );
```

Example 3. Managed hub initialization. onMHPublish, onMHSubscribe, onMHUnsubscribe and onMHSecurityAlert are managed hub security callbacks. Taken from [45].

When the hub instance is created, client applications (widgets of the mashup) will be embedded within a parent application. For each client application, a sandbox (called a Container) associated with a given Managed Hub is constructed (see Example 4) and the widget itself (called a Component) is initialized inside the sandbox. Penultimate action for the client is to connect to the hub as shown in Example 5. As the final step of the OpenAjax Hub initialization process, a component might subscribe to message topics as shown in Example 6.

```
hubClient = new OpenAjax.hub.IframeHubClient({
        HubClient: {
          onSecurityAlert: clientSecurityAlertHandler
        }
      });
```

Example 4. IframeHubClient initialization, clientSecurityAlertHandler handles client's security alerts. Based on [45].

```
hubClient.connect( connectCompleted );
```

Example 5. Connect to managed hub, connectCompleted callback is called when this operation completes [47].

```
function connectCompleted ( hubClient, success, error ) {
   if (success) {
     /* Call hubClient.subscribe(...) to subscribe to message topics */
   }
}
```

Example 6. Callback that is invoked upon successful connection to the Managed Hub, based on [45].

Topics consist of tokens separated by the period character (.), for example "org.example.topic". Also wildcards such as "*" or "**" can be used when subscribing to topics. A single asterisk (*) matches exactly one token, for example the topic "org.example.*" matches topics "org.example.topic" and "org.example.important", but not "org.example.important.topic". A single asterisk can be used between the tokens, for example "org.*.topic". However, the trailing wildcard token, a double asterisk (**), which matches one or more tokens with any value, can be used only at the end of tokens [46]. For example, "org.example.**" is a valid topic that matches "org.example.topic" as well as "org.example.important.topic". Topic names such as "org.**.invalid", "org.in*alid.topic" and "org..topic" are invalid.

After the hub and the widgets have been initialized, the widgets can publish messages to the hub. As shown in Figure 7, Component-B publishes a message to a topic. Given that Component-B is inside the IframeContainer, the message gets marshaled across browser frames to the Managed Hub. If the Security manager callbacks decide to let this message through, it will be sent to every container whose component has subscribed to this topic. In current case, the message will be passed to Component-A's callback.

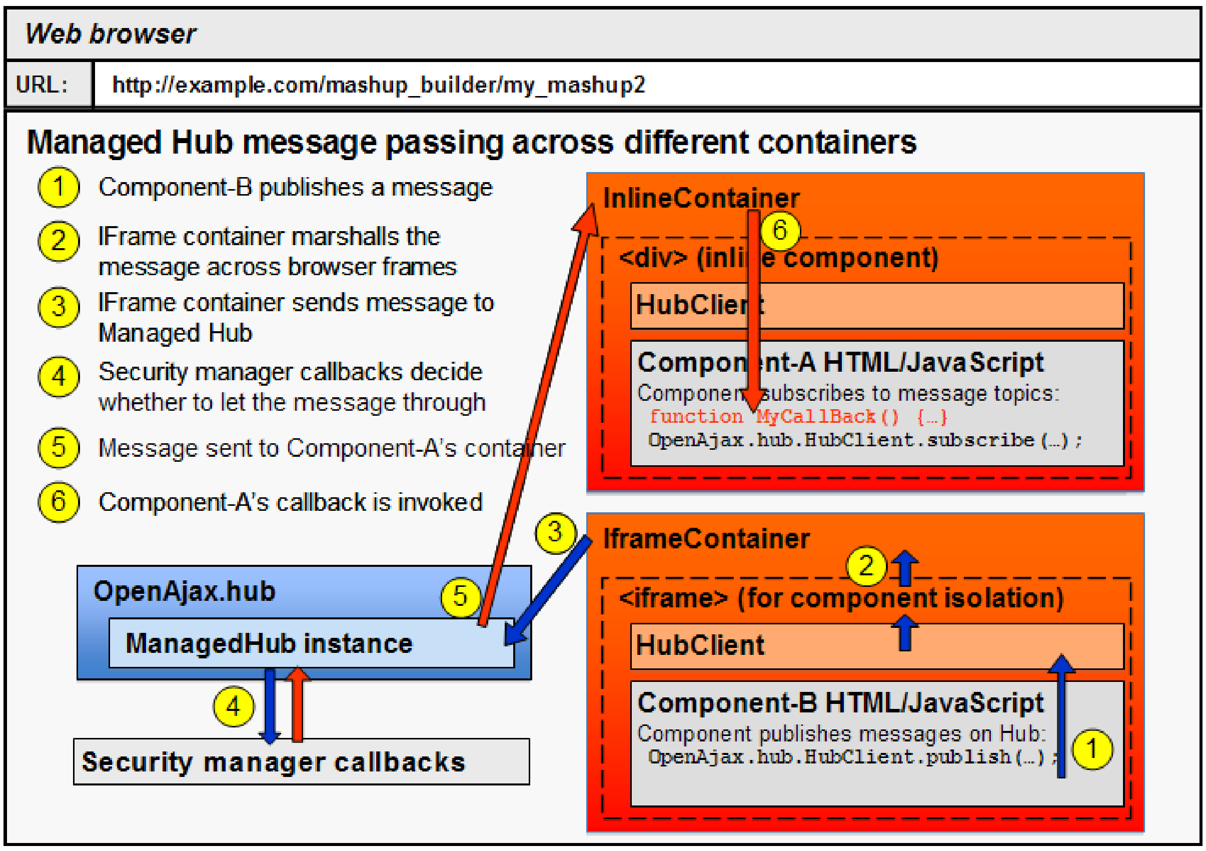

Web browser
URL: http://example.com/mashup_builder/my_mashup2
Managed Hub message passing across different containers
1 Component-B publishes a message
2 IFrame container marshalls the message across browser frames
3 IFrame container sends message to Managed Hub
4 Security manager callbacks decide whether to let the message through
5 Message sent to Component-A's container
6 Component-A's callback is invoked
InlineContainer
<div> (inline component)
HubClient
Component-A HTML/JavaScript
Component subscribes to message topics:
function MyCallBack() {…}
OpenAjax.hub.HubClient.subscribe(…);
IframeContainer
<iframe> (for component isolation)
HubClient
Component-B HTML/JavaScript
Component publishes messages on Hub:
OpenAjax.hub.HubClient.publish(…);
OpenAjax.hub
ManagedHub instance
Security manager callbacks

Figure 7. Managed Hub Message Passing, taken from [45].

As the OpenAjax Hub specification [48] states, the topic has to be a string describing the published message. The topics published are not allowed to contain any wildcards [46]. The message itself can equally be a string as shown in Example 7 or a JSON-serializable JavaScript object as shown in Example 8.

```
hubClient.publish("org.example.foo","foo");
```

Example 7. Publish a string value, based on [48].

```
var location = { lat:current_latitude, long:current_longitude };
hubClient.publish("org.example.bar",location);
```

Example 8. Publish a JSON-serializable JavaScript object, based on [48].

4.2.3. Facebook

Facebook was chosen as the underlying social network to enable effective sharing of states of widget-based microsites.

Facebook is a social network service launched in 2004. The number of users has grown over the years. This is illustrated by the fact that by 2007 Facebook was reported to have more than 21 million registered members [49]. By the January of 2011 it is said to have over 600 million monthly active users [50].

Facebook has a micro-blogging feature called the wall, on which a user can post one's thoughts, links or even photos for others to see. Furthermore, Facebook provides an API that enables integration of Facebook with other web-sites. It also provides tools for reading and writing data to Facebook, or including a "Like" button to your web-site. Moreover, it provides API for Single Sign-on (SSO), for using other web applications, which require logging in without prompting the user to re-enter credentials [51].

To enable a third party application to post something directly on Facebook's wall, the application has to be registered in Facebook. Registration of the application links the host name used for registration with the application identifier given by Facebook. If the application is registered in Facebook, it is allowed to post web addresses with additional information about the address to user's wall [51], whereby the application identifier is included with the request. However, Facebook makes a check whether the host where the request is coming is in correspondence with the host linked with the given application identifier.

The wall posting gives a social measurement to the proposed solution. It enables to share results of a stateful mashup by posting a reference to the mashup on the wall.

4.3.Integration

The Wookie-OpenAjax Hub Bridge is a Wookie widget, that is capable of communicating (subscribing and publishing) with widgets connected to the OpenAjax Hub. In another point of view, it is an OpenAjax Hub client, which has access to Wookie's context variables and therefore is capable of persisting instance preferences in the Wookie's database. It is a widget that has qualities from both Wookie and OpenAjax Hub, so to say.

The general purpose of the Wookie-OpenAjax Widget is to listen to every message sent through the OpenAjax Hub and to store the information to the Wookie's database. The messages are stored as a key-value pairs. The topic is stored as a key and the message content is stored as the value of the key. If the message content is a simple string object, it is stored without a conversion. However, if the message is a JavaScript object, it will be serialized to JSON format before saving it to the Wookie's database.

When the same widget instance is loaded later, the widget plays back all the gathered messages. In other words it loads all the persisted messages from the Wookie's database and publishes each one of them to the correct topic in the hub. The messages are loaded in the same order they were recorded. Due to the fact that the keys are unique and the values might get overwritten let us examine Example 9 step by step.

`Widget.preferences.setItem("foo", "one"); (1)`	`foo = "one"`

`Widget.preferences.setItem("bar", "two"); (2)`	`foo = "one"`
	`bar = "two"`

`Widget.preferences.setItem("foo", "three"); (3)`	`foo = "three"`
	`bar = "two"`

Example 9. Widget preference "foo" gets overwritten.

After step (1), the only preference set in the database is “foo” with the value “one”. After the preference “bar” is added at step (2), there are two preferences “foo” and “bar” in the database, with the values correspondingly “one” and “two”, whereby the adding order is preserved. Finally, when step (3) is executed, the value under the key “foo” is overwritten, however, the order of the preferences in the database will remain the same. Accordingly, the outcome of this piece of code is that the widget instance has two preferences set: “foo” is “three” and “bar” is “two” in the same order.

5. Implementation

The implementation of the proof of concept is divided into two parts: the Wookie-OpenAjax Hub Widget and the surrounding web application, or the so called microsite.

The Wookie-OpenAjax Hub Bridge is a widget that is served from the Wookie Engine and is capable of communicating with other widgets in the web application using the OpenAjax Hub infrastructure. The widget is intended to run in an HTML <iframe> element, to be physically isolated from other widgets in terms of JavaScript security. The OpenAjax Hub provides a communication channel for those isolated components. The widget is built using pure HTML and JavaScript and it is compatible with W3C Widgets 1.0 family of specifications [11]. The widget is intended to have no visual output.

The widget is packaged into a .wgt archive (woah.wgt), which is a valid ZIP archive. In fact Wookie provides tools for building a widget from its source. Apache Ant [52] script is used to make the .wgt file.

The created widget archive must contain one or more start file and a configuration document. Optionally it can also contain icons, digital signatures and arbitrary files [11]. The Wookie-OpenAjax Hub Bridge widget package contains the files summarized in Table 3.

Location	Description
/	Widget’s root folder
/config.xml	The configuration document
/index.html	Start file, file name is defined in config.xml
/build.xml	Widget’s ant build file
/images	Folder that contains images
/images/icon.png	Default icon
/scripts	Folder that contains all the JavaScript source files where the widget’s behavior is described.
/scripts/properties.js	Methods for storing and fetching the widget’s properties from the Wookie Engine
/scripts/oahHelper.js	Helper methods for connecting to OpenAjax-Hub
/lib	Folder that contains all the necessary supporting JavaScript files (i.e. frameworks and libraries) for the widget
/lib/OpenAjaxManagedHub-all.js	OpenAjax Hub 2.0 implementation

Table 3. The Wookie-OpenAjax Hub Widget structure.

The configuration document is an XML document that has a widget element in the root. The widget element is in the widget’s namespace `http://www.w3.org/ns/widgets` and serves as a container element. The widget element has another attribute *id*, which specifies the identifier for the widget. To prevent naming collisions, namespace is often used as a part of the *id*. In particular, there could be a number of weather widgets, which could be tangled without a namespace prefix. So instead of using just “`weather`”, “`http://example.org/weather`” is preferable. In order to follow the preferred naming convention, “`http://wookie.apache.org/widgets/woah`” is used, as shown in Example 10. The widget element contains several children elements. The *name* and the *description* elements specify the human-readable name and the description for the widget. The *content* element defines the custom start file, which the user agent is presumed to use when the widget is instantiated. The source attribute `src` allows an author to point to a file inside the widget container. The *icon* element is similar to the content element as its `src` attribute points to an icon file. The icon file is used in the Wookie Engine widget library helping the user to recognize the widget at a glance. The *author* element marks the person who created the widget. The *license* element specifies the license under which the widget is distributed. The example configuration document is shown in Example 10.

```
<widget xmlns="http://www.w3.org/ns/widgets"
            id="http://wookie.apache.org/widgets/woah" >
      <name>woah</name>
      <description>Wookie-OpenAjaxHub bridge</description>
      <content src="index.html"/>
      <icon src="images/icon.png"/>
      <author>Allar Tammik</author>
      <licence>
      …
      </licence>
</widget>
```

Example 10. Wookie widget's configuration document.

The class diagram of the Wookie-OpenAjax Hub Widget is shown in Figure 8. Methods are divided into two separate JavaScript files by the field of operation. All the functions that are connected to the OpenAjax Hub are collected in `oahHelper.js` and the functions responsible for interactions with Wookie service layer is put in `properties.js`.

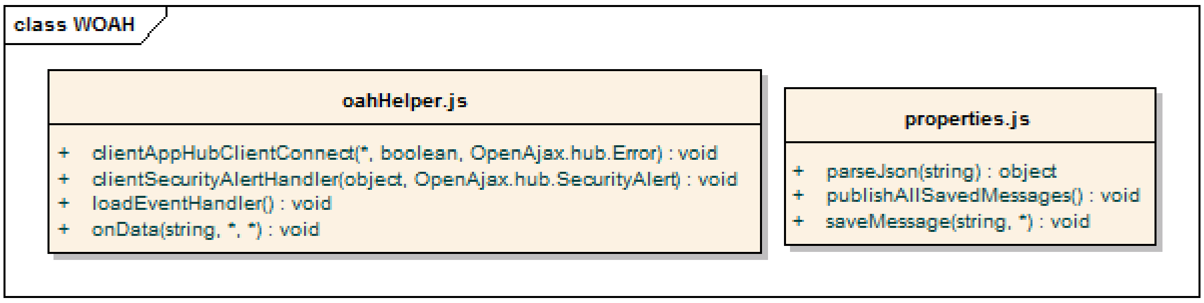

class WOAH
oahHelper.js
+ clientAppHubClientConnect(*, boolean, OpenAjax.hub.Error) : void
+ clientSecurityAlertHandler(object, OpenAjax.hub.SecurityAlert) : void
+ loadEventHandler() : void
+ onData(string, *, *) : void
properties.js
+ parseJson(string) : object
+ publishAllSavedMessages() : void
+ saveMessage(string, *) : void

Figure 8. Structure of JavaScript functions in the Wookie-OpenAjax Hub Bridge.

The start file is a very simple HTML file that basically loads the JavaScript files shown above in the Table 3. When everything is loaded, the `loadEventHandler` function in the `oahHelper.js` is executed with HTML `body` tag `onload` attribute value. The `loadEventHandler` creates a `OpenAjax.hub.IframeHubClient` using a `clientSecurityAlertHandler` function as an `OpenAjax.hub.SecurityAlert` callback and calls its `connect` method. If the asynchronous `connect` call completes, the `clientAppHubClientConnect` callback is invoked with three parameters: the item on which the connect operation was invoked, `boolean` to indicate the successfulness of the operation and `OpenAjax.hub.Error` type error code if an error occurs.

When the connecting process is successfully finished, the widget subscribes to all topics using a special wildcard “**” (double asterisk). The subscription is required for enabling retrieving all messages. Every time a message is broadcasted to the hub the `onData` callback is invoked with three parameters: the topic, the event's payload data, which can be any JSON-serializable value and the `subscriberData` that the caller might have added to the `subscribe` call. The `onData` callback invokes the `saveMessage` function in `properties.js`. The aim of `saveMessage` function is to save the `topic` and the `message` given as parameters to the Wookie storage area. Since the message parameter can be any JSON-serializable value, the function checks the data type of the message with a `typeof` operator. If the message variable is not a string type, the message is serialized to a JSON string with `stringify` method (as shown in Example 11). The serialized message is saved to the Wookie's database using Wookie widget instance's preference storing mechanism. The topic from the message is saved as a key and the serialized message as a value as illustrated in Example 12.

```
JSON.stringify(message);
```

Example 11. Message is serialized with stringify method.

```
Widget.preferences.setItem(topic, message);
```

Example 12. Message is stored in widget instance's preferences in the Wookie's database.

When the widget instance is reinitialized, all stored messages will be published to the hub by invoking `publishAllSavedMessages` method in the `properties.js`. It fetches all the widget instance's preferences from the Wookie's database and iterates over the key-value pairs' keys (which hold message topics). For each key the value is taken with `Widget.preferences.getItem(key)`. Among the keys, there is one extra key "sharedDataKey", which is stored in the database by the Wookie Engine and will be ignored by the `publishAllSavedMessages` method. The values which are either stringified JavaScript objects or simply strings will be deserialized with `parseJSon` function, which internally uses `JSON.parse` method (as shown in Example 13).

```
JSON.parse(str);
```

Example 13. Message is deserialized with parse method.

Each topic can appear once in one widget instance's preferences store. This can be very useful if widgets communicating in the hub wish to overwrite previous values. Let us imagine an example of a location-based OpenAjax hub mashup, where one of the widgets has a functionality that allows a user to enter its location, for example a city name. The user's choice will be published to other widgets connected to the hub, including the Wookie-OpenAjax Hub Widget, and some additional information about the city will be shown to the user. If the user accidentally makes a typo or purposely wants to change the location, another message will be broadcasted with a new value. The new value will overwrite the location in the Wookie's database. Thus, when the same Wookie-OpenAjax Hub Widget instance is reinitialized it only publishes the recent value. Therefore the first, invalid location will not be broadcasted to the hub by the Wookie-OpenAjax Hub Widget.

6. Test Application

The test application is a simple widget-based web application that follows the widget-centric paradigm of application development and features the developed solution. When activating the test-application an OpenAjax Hub instance is created and all the widgets, including the Wookie-OpenAjax Hub Widget, are initialized. Also state sharing through social networks is exemplified through Facebook integration.

6.1.Package structure

The test application is presented as a standard exploded web application archive (WAR). The `WEB-INF` directory contains a deployment descriptor – a `web.xml` file, which describes the structure of the application. The deployment descriptor declares a "welcome file" for the application. The welcome file specifies the opening page of the application, which in the context of this test application is the `manager.html`.

6.2.Initialization

The opening page of the microsite loads all the necessary libraries, such as the OpenAjax Hub and the jQuery as well as custom functions. The jQuery library is used to make an HTTP request to the Wookie REST API. Custom functions are divided into two JavaScript files by area of responsibility. All the logic related to the OpenAjax Hub widgets of the microsite is collected to `mashup.js`. These widgets are the core of the mashup. Additional logic needed to initialize the Wookie-OpenAjax Hub Bridge is located in the `wookieWidgetFactory.js`. JavaScript files are loaded using an HTML <script> tag as shown in Example 14.

```
<script src="OpenAjaxManagedHub-all.js"></script>
<script src="jquery-1.5.2.min.js"></script>
<script src="wookieWidgetFactory.js"></script>
<script src="mashup.js"></script>
```

Example 14. Including JavaScript files into the application.

Initialization of the OpenAjax Hub is activated by the HTML `body` tag `onload` attribute. The instantiation of the OpenAjax Hub is done in `mashup.js`. The `loadEventHandler` function creates a `ManagedHub` instance and two `IframeContainer` instances, which will be appended to the HTML body. These two components are OpenAjax widgets that are implemented in the `ClientPublisher.html` and `ClientSubscriber.html` correspondingly.

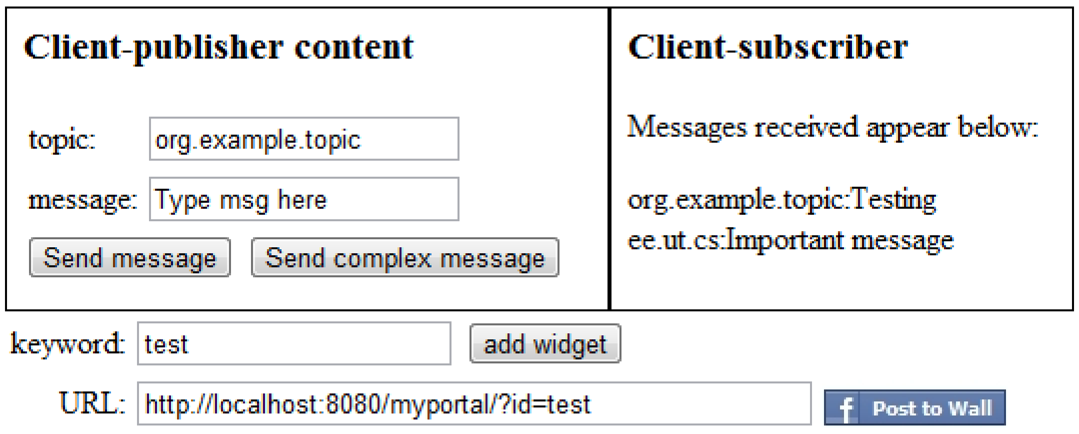
In addition, to be able to integrate with Facebook as a Facebook application, the Facebook JavaScript Software Development Kit (SDK) must be loaded as well as the Facebook application identifier specified. To initialize the library, `FB.init` must be invoked. Also, it is mandatory to have an HTML <div> element with an id “fb-root” within the document as well. Initialization of Facebook SDK is shown in Example 15.

```
<script src="http://connect.facebook.net/en_US/all.js#xfbml=1"></script>
<div id="fb-root"><script>
      var FB_APP_ID = '207749675918893';
      FB.init({
            appId:FB_APP_ID, cookie:true,
            status:true, xfbml:false
      });
</script></div>
```

Example 15. Initializing Facebook SDK.

6.3.User Interaction

The publisher is a widget, which purpose is to send messages to the hub. It has input fields for a topic and a message; and a “Send message” button that transmits the message to the hub. Moreover, there is another button “Send complex message”, which sends a whole JavaScript object as the message. By clicking the button, a complex JavaScript object is assembled and sent to the hub. The subscriber is a widget that subscribes to all topics of the hub. If a message is sent to the hub, the subscriber outputs the message. If the message received is a JavaScript object, it is serialized to JSON for visualization. Screenshot of the working prototype can be viewed in Figure 9.


Client-publisher content

topic: org.example.topic

message: Type msg here

Client-subscriber

Messages received appear below:

org.example.topic:Testing
ee.ut.cs:Important message

keyword: test

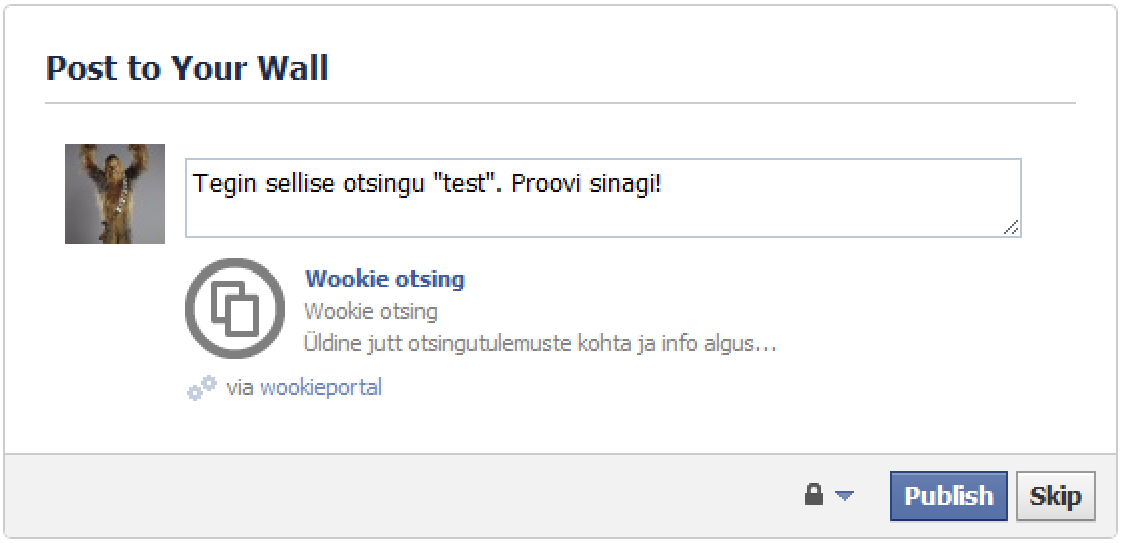
URL: http://localhost:8080/myportal/?id=test

Figure 9. Screenshot of the prototype of the microsite.

Under the mashed OpenAjax Hub widgets, there is a keyword input and a button to instantiate a Wookie-OpenAjax Hub Bridge. When the button is pushed the `initWookieWidget` function in `wookieWidgetFactory.js` is invoked. As a result, the invisible Wookie-OpenAjax Hub Bridge is initialized and a URL to identify the specific mashup is generated to the following input. In other words, the "id" parameter in the URL identifies the Wookie-OpenAjax Hub Widget instance, thus the widget instance can be reinitialized afterwards. This generated URL can now be shared between browsers or computers. Particularly the URL can be posted to Facebook's wall for other users to view. The "post to wall" button is implemented in the `manager.html` as shown in Example 16. When pressed, the `submitToWall` function is invoked, which makes a request to the Facebook API that opens a dialog, like shown in Figure 10. If a user presses the "Publish" button, the link will be posted to user's wall in Facebook. If the link is clicked in Facebook, the user is redirected back to the microsite. Due to the "id" parameter in the URL, the Wookie-OpenAjax Bridge will be revived.

```
<a  href="javascript:void(0)"  onclick="submitToWall()"  class="fb_button
fb_button_small">
      <span class="fb_button_text">Post to Wall</span>
</a>
```

Example 16. Facebook post to wall button.


Post to Your Wall

Tegin sellise otsingu "test". Proovi sinagi!

Wookie otsing
Wookie otsing
Üldine jutt otsingutulemuste kohta ja info algus…
via wookieportal

Figure 10. Facebook Post to Wall dialog.

The Wookie-OpenAjax Hub Bridge initialization passes through the following steps. Firstly, a POST request is made via the proxy to the Wookie REST API to get a widget instance. The Wookie Engine composes a response, which is an XML document (see

Example 1 above). When the response arrives back to the microsite, a `widgetResponseCallback` is executed in the `wookieWidgetFactory.js`. The callback traverses the response XML and extracts Wookie widget's URL. Then the `initIframeContainer` function creates an invisible HTML <iframe> element and loads the Wookie-OpenAjax Hub Widget in it with a GET request directly to the Wookie Engine. The widget will become operational after all its `onload` activities are done. The widget instantiation is described in Figure 11.

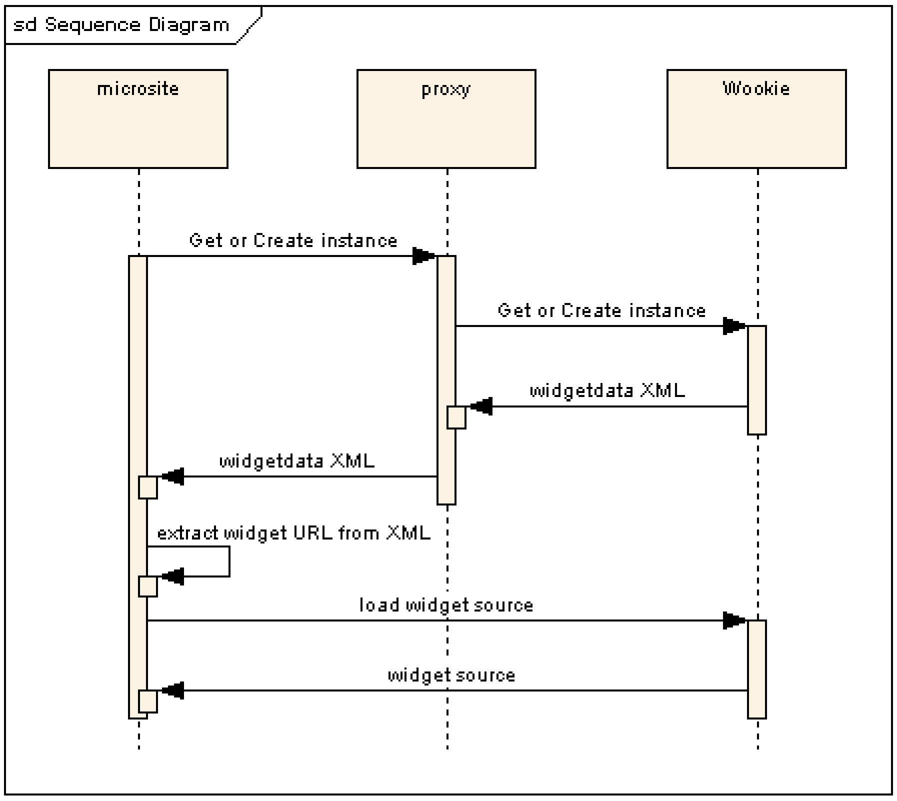


Figure 11. Sequence diagram of a widget instantiation.

7. Conclusions and Future Work

This thesis focuses on the data sharing of widget-based applications by enabling mashups to store their states within application. Furthermore, it enables to exchange the states between applications. The latter enables usage of social networks in inter-application state exchange.

This thesis proposed a solution for the OpenAjax Hub enabled mashups to record and reproduce messages exchanged by widgets to overcome the limitations current platforms have. The Wookie-OpenAjax Hub Bridge implemented in this thesis, combines Wookie's state storing abilities and the OpenAjax Hub's message delivery capabilities to enable a mashup to store its state without additional coding. In other words, instead of implementing the state-storing mechanism for your system, this widget can be included to your mashup. Additionally, a test application was implemented to demonstrate the Wookie-OpenAjax Hub Bridge in action. The test application included two widgets in addition to the Wookie-OpenAjax Hub Widget. Furthermore, a feature to share the state of the mashup via Facebook was implemented in the test application.

As a future work, it is possible to improve the Wookie-OpenAjax Hub Bridge in many ways by adding features that could make the state sharing more intelligent. One certain direction to extend the solution could be to make the topics that the widget records or replays dynamically configurable. Some systemic data could be stored in the Wookie's database besides the messages sent by other widgets in the mashup. This can be done by defining some special messages – a set of commands to manipulate the systemic data. For example, a microsite can send a special message, with a predefined topic that would define which messages would be stored and replayed. Of course there should be a mechanism to filter out all the systemic preferences not to replay them to other widgets. This enhancement allows scenarios where the microsite can dynamically manage, which parts of the state is restored from the database.

Yet another quite simple but crucial improvement would be to remove the messages from a Wookie-OpenAjax Hub Widget's instance. This can be done by implementing a special command, which removes the preferences in the database. By utilizing this feature, resetting of the state of the mashup can be evoked.

Currently, messages with identical topics will be overwritten in the Wookie-OpenAjax Hub Widget. If that is not considered a feature, a technical improvement can be implemented. Depending on the expected outcome, messages that are currently overwritten could be added together, or perhaps ignored at all.

Another possible scenario for a future development is to add multiple Wookie-OpenAjax Hub Widgets to a microsite. These widgets can be configured in such a way that each widget instance would be recording different topics. This kind of approach would make possible to use these widget instances as separate sources in terms of mashups. These sources could be later used either separately or mashed with each other later. As an example of this scenario, let us imagine there is a mashup that deals with data from Central Commercial Register and Land Records. The responsibility to store messages could be shared by two widget instances correspondingly.

The Wookie-OpenAjax Hub could be improved by introducing a systemic read-only flag, which would denote whether the data stored in the Wookie's database could be modified. This improvement would enable the mashup creator to lock the state before sharing it. In another words, one user could share the state of the mashup without worrying about the possibly unwanted modifications the other user could make.

To combine the latter and the penultimate, more sophisticated mechanisms could be developed. For example, branching which is common in Version Control Systems. In particular, if one user shares a mashup to another user, the state is copied from one Wookie-OpenAjax Hub Widget instance to another. As a result, there will be two copies of the state which will be modified in parallel by two different users.

Summary (in Estonian)

Vidinapõhiste veebirakenduste interaktiivne andmevahetus-mehhanism.

Magistritöö (30 EAP)

Allar Tammik

Kokkuvõte

Tänasel veebimaastikul on kasvavaks trendiks veebilehtede vaheline sisu jagamine. Staatilise sisu asemel kasutatakse üha enam vidinaid. Vidinad on taaskasutatavad veebikomponendid, mis sisaldavad mingit konkreetset funktsionaalsust. Lihtsamad vidinad on enamasti olekuta. Keerukamad vidinad oskavad veebirakenduses omavahel suhelda, näiteks saates üksteisele sõnumeid. Nii saavad vidinad üksteist mõjutada ja seeläbi olla *mashup*'i laadse veebirakenduse ehitusklotsideks, määratledes selle oleku. Kuna vidinad on nõrgalt sidestunud komponendid, ei ole nad võimelised veebirakenduse olekut salvestama.

Antud magistritöö pakub välja lahenduse veebirakenduse sees olevate vidinate vahetatud sõnumite jäädvustamiseks ja taasesitamiseks, võimaldades seeläbi veebirakenduse olekut salvestada ja sõpradega jagada. Lahendus baseerub jaoturipõhisel sõnumivahetusel, kasutades tehnoloogiana *OpenAjax Hub* raamistikku. See tähendab, et kõik ühes veebirakenduses olevad vidinad on jaoturiga ühenduses. Sõnumite salvestamiseks kasutatakse *Wookie* nimelist vidinamootorit, mis võimaldab luua olekuga vidinaid ja seda olekut ka salvestada.

Magistritöö raames realiseeritakse iseseisev vidin, mis „sillana“ ühendab neid kaht tehnoloogiat. Sellest tulenevalt on realiseeritud vidina nimeks *Wookie-OpenAjax Hub Bridge*. Loodud vidin kuulab kõiki jaoturisse saadetud sõnumeid ja salvestab need andmebaasi. Kui sama vidina isend hiljem taaselustada, siis see vidin taasesitab kõik andmebaasi salvestatud sõnumid jaoturisse.

Realiseeritud vidina kasutamiseks luuakse väike näidisportaal, mille sees olevad vidinad omavahel sõnumeid vahetavad. Näidisportaal demonstreerib saadetud sõnumite salvestamist ja taasesitamist, ning ka oleku jagamist suhtlusvõrgustikuga *Facebook*.

Bibliography

[1] Antero Taivalsaari and Tommi Mikkonen, "Mashups and Modularity: Towards Secure and Reusable Web Applications," in *Proceedings of 1st Workshop on Social Software Engineering and Applications*, L'Aquila, Italy, 2008, pp. 25-33.

[2] Tim O'Reilly, "What Is Web 2.0: Design Patterns and Business Models for the Next Generation of Software," *Communications and Strategies*, vol. 65, pp. 17-38, 2007.

[3] Ray Valdes et al., *Hype Cycle for Web and User Interaction Technologies*.: Gartner, 2010.

[4] Nan Zang, Mary Beth Rosson, and Vincent Nasser, "Mashups: who? what? why?," in *Human Factors in Computing Systems*, Florence, Italy, 2008, pp. 3171-3176.

[5] Jin Yu, Boualem Benatallah, Fabio Casati, and Florian Daniel, "Understanding Mashup Development," *IEEE Internet Computing*, pp. 44-52, September/October 2008.

[6] Florian Urmetzer et al., "Fast and Advanced Storyboard Tools. State of the art in gadgets, semantics, visual design, SWS and Catalog," Deliverable D2.1.2, FP7 Project FAST, 2010.

[7] OpenAjax Alliance. OpenAjax Hub 2.0 Specification. Accessed: 16.05.2011. http://www.openajax.org/member/wiki/OpenAjax_Hub_2.0_Specification

[8] Apache Software Foundation. Apache Wookie. Accessed: 16.05.2011. http://incubator.apache.org/wookie/

[9] OpenAjax Alliance. Accessed: 16.05.2011. http://www.openajax.org

[10] Rainer Villido, "Semantic Integration Platform for Web," University of Tartu, Tartu, Master Thesis 2010.

[11] World Wide Web Consortium. Widget Packaging and Configuration. Accessed: 16.05.2011. http://www.w3.org/TR/widgets/

[12] Open Mashup Alliance. Accessed: 16.05.2011. http://www.openmashup.org/

[13] Open Mashup Alliance, "EMML Changes Everything: Profitability, Predictability & Performance through Enterprise Mashups," 2009.

[14] Marwan Sabbouh, Jeff Higginson, Salim Semy, and Danny Gagne, "Web mashup scripting language," in *Proceedings of the 16th international conference on World Wide Web*, Banff, Alberta, Canada, 2007, pp. 1305-1306.

[15] IBM. IBM Mashup Center. Accessed: 16.05.2011. http://www.ibm.com/software/info/mashup-center/

[16] Kapow Software. Accessed: 16.05.2011. http://kapowsoftware.com/

[17] JackBe. Presto: The Real-Time Intelligence Solution. Accessed: 16.05.2011. http://www.jackbe.com/products/

[18] Volker Hoyer and Marco Fischer, "Market Overview of Enterprise Mashup Tools," in *Proceedings of International Conference on Service-Oriented Computing*, Berlin, Germany, 2008, pp. 708-721.

[19] Google. iGoogle. Accessed: 16.05.2011. http://www.google.com/ig

[20] Yahoo! Pipes. Accessed: 16.05.2011. http://pipes.yahoo.com/pipes/

[21] Netvibes. Netvibes. Accessed: 16.05.2011. http://www.netvibes.com/

[22] Robert J. Ennals and Minos N. Garofalakis, "MashMaker: mashups for the masses," in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, Beijing, 2007, pp. 1116-1118.

[23] Jeffrey Wong and Jason I. Hong, "Making Mashups with Marmite: Towards End-User Programming for the Web," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, San Jose, California, 2007, pp. 1435-1444.

[24] Oscar Díaz, Sandy Pérez, and Iñaki Paz, "Providing Personalized Mashups Within the Context of Existing Web Applications," in *Web Information Systems Engineering – WISE 2007*. Berlin, 2007, pp. 493-502.

[25] Ikuya Yamada, Yamaki Wataru, Nakajima Hirotaka, and Takefuji Yoshiyasu, "Ousia Weaver: A tool for creating and publishing mashups," in *International World Wide Web Conference*, Raleigh, North Carolina, 2010, p. 8.

[26] Stéphane Sire, Micaël Paquier, Alain Vagner, and Jérôme Bogaerts, "A Messaging API for Inter-Widgets Communication," in *Proceedings of the 18th international conference on World wide web*, Madrid, 2009, pp. 1115-1116.

[27] Apache Software Foundation. Apache Shindig. Accessed: 16.05.2011. http://shindig.apache.org/

[28] World Wide Web Consortium. (2008, Apr) Widgets 1.0: The Widget Landscape. Accessed: 16.05.2011. http://www.w3.org/TR/widgets-land/

[29] Opera Software. Opera Widgets. Accessed: 16.05.2011. http://widgets.opera.com/

[30] Scott Wilson. (2011, Apr.) W3C Widgets with Opera 11. Accessed: 16.05.2011. http://scottbw.wordpress.com/2011/04/13/w3c-widgets-with-opera-11/

[31] Philippe B. Kruchten, "The 4+1 View Model of Architecture," *IEEE Software*, vol. 12, no. 6, pp. 42-50, Nov. 1995.

[32] Chris Karlof, Umesh Shankar, J. Doug Tygar, and David Wagner, "Dynamic pharming attacks and locked same-origin policies for web browsers," in *Proceedings of the 14th ACM conference on Computer and communications security*, New York, 2007, pp. 58-71.

[33] Apache Software Foundation. Apache Incubator. Accessed: 16.05.2011. http://incubator.apache.org/

[34] Scott Wilson, "Wookie Widget Developer's guide," 2009.

[35] Google. Google Wave. Accessed: 16.05.2011. http://wave.google.com/

[36] OpenSocial Foundation. OpenSocial. Accessed: 16.05.2011. http://www.opensocial.org/

[37] Apache Software Foundation. Wookie Server Administrators Guide. Accessed: 16.05.2011. http://incubator.apache.org/wookie/wookie-server-administrators-guide.html

[38] Apache Software Foundation. Wookie Plugin Developers Guide. Accessed: 16.05.2011. http://incubator.apache.org/wookie/wookie-plugin-developers-guide.html

[39] Apache Software Foundation. Wookie REST API. Accessed: 16.05.2011. http://incubator.apache.org/wookie/wookie-rest-api.html

[40] Scott Wilson. Wordpress plugin updated. Accessed: 16.05.2011. http://www.mail-archive.com/wookie-dev@incubator.apache.org/msg01418.html

[41] World Wide Web Consortium. Language tags in HTML and XML. Accessed: 16.05.2011. http://www.w3.org/International/articles/language-tags/Overview.en.php

[42] World Wide Web Consortium. Scripting and Ajax. Accessed: 16.05.2011. http://www.w3schools.com/ajax/default.asp

[43] OpenAjax Alliance. OpenAjax Hub 2.0 Specification Introduction. Accessed: 16.05.2011. http://www.openajax.org/member/wiki/OpenAjax_Hub_2.0_Specification_Introduction

[44] OpenAjax Alliance. Introducing OpenAjax Hub 2.0 and Secure Mashups. Accessed: 16.05.2011. http://www.openajax.org/whitepapers/Introducing%20OpenAjax%20Hub%202.0%20and%20Secure%20Mashups.php

[45] OpenAjax Alliance. OpenAjax Hub 2.0 Specification Managed Hub Overview. Accessed: 16.05.2011. http://www.openajax.org/member/wiki/OpenAjax_Hub_2.0_Specification_Managed_Hub_Overview

[46] OpenAjax Alliance. OpenAjax Hub 2.0 Specification Managed Hub APIs. Accessed: 16.05.2011. http://www.openajax.org/member/wiki/OpenAjax_Hub_2.0_Specification_Managed_Hub_APIs

[47] OpenAjax Alliance. OpenAjax Hub 2.0 Specification Topic Names. Accessed: 16.05.2011. http://www.openajax.org/member/wiki/OpenAjax_Hub_2.0_Specification_Topic_Names

[48] OpenAjax Alliance. OpenAjax Hub 2.0 Specification Publish Subscribe Overview. Accessed: 16.05.2011. http://www.openajax.org/member/wiki/OpenAjax_Hub_2.0_Specification_Publish_Subscribe_Overview

[49] Nicole B. Ellison, Charles Steinfield, and Cliff Lampe, "The Benefits of Facebook "Friends": Social Capital and College Students' Use of Online Social Network Sites.," *Journal of Computer-Mediated Communication*, vol. 12, no. 4, pp. 1143–1168, July 2007, http://onlinelibrary.wiley.com/doi/10.1111/j.1083-6101.2007.00367.x/full.

[50] Nicholas Carlson. Facebook Has More Than 600 Million Users, Goldman Tells Clients. Accessed: 16.05.2011. http://www.businessinsider.com/facebook-has-more-than-600-million-users-goldman-tells-clients-2011-1

[51] Facebook. Facebook Developer's Documentation. Accessed: 16.05.2011. http://developers.facebook.com/docs/

[52] The Apache Software Foundation. Apache Ant. Accessed: 16.05.2011. http://ant.apache.org/

[53] Apache Software Foundation. FAQ. Accessed: 16.05.2011. http://incubator.apache.org/wookie/faq.html

[54] Apache Software Foundation. Downloading and Installing Wookie. Accessed: 16.05.2011. http://incubator.apache.org/wookie/downloading-and-installing-wookie.html

[55] Facebook. Create Application. Accessed: 16.05.2011. http://www.facebook.com/developers/createapp.php

Appendix A. Installation

To deploy a Wookie Engine, Java 6 and Apache Ant 1.7.1 must be present. There are documented cases when running with Apache Ant 1.8 causes problems [53]. Wookie will be compiled from its source code, which can be downloaded from Apache SVN repository. Since Wookie is still in incubation and is a subject to active development, very specific revision is used in this thesis. The Wookie-OpenAjax Hub Bridge is tested using Wookie revision 1089020. System requirements are summarized in Table 4.

Java	Java 6
Widget server	Apache Wookie revision 1089020
Build tool	Apache Ant 1.7.1
Application server for test application	Apache Tomcat 6

Table 4. System requirements.

Wookie's source code can be downloaded using the following SVN command:

```
svn co -r 1089020 http://svn.apache.org/repos/asf/incubator/wookie/trunk
wookie
```

The Wookie Engine will be running in standalone mode using a Jetty container and a Derby database. To build and run, the Ant build scripts are used. To run Wookie in standalone mode with default settings type following command in the folder that was created during the last command:

```
ant run
```

Additional running options can be specified by adding `-Drun.args="<property_name>=<property_value>"`. Available running options are "`port`" and "`initDB`". The parameter "`port`" defines the port for the Jetty to run; the "`initDB`" defines whether to clean the database to its initial state. If the `initDB` is set to true, all the data stored in the database will be lost. For example, to run Wookie in standalone mode on port 9080 and to preserve database state, the following command can be executed:

```
ant run -Drun.args="port=9080 initDB=false"
```

First launch will take some time, because all dependency libraries are downloaded using Ivy.

To deploy the Wookie-OpenAjax Hub Widget, copy woah.wgt to `webapp` directory's `deploy` subdirectory. In standalone mode with default configuration it is located in project root `build/webapp/wookie/deploy`. Wookie supports widgets, hot deployment, which means all `.wgt` files dropped to the deploy directory will be deployed automatically. When widget's deployment is finished, a confirmation message will be shown in the application server's log (see Example 17) and the widget will be available for initiation.

```
 [java] 17:06:27,268   INFO ContextListener:210 - woah' - Widget was
successfully imported into the system.
```

Example 17. A log message indicating a successful widget deployment.

Additional information concerning the installation of Wookie can be found on the Wookie's homepage in Downloading and Installing Wookie section [53] or in the Frequently Asked Questions [54].

In case the test application and the Wookie are run on different application servers, thus on different ports or even on different physical servers, it is required to have a proxy to enable cross-domain requests from portal to Wookie. I have implemented a really simple proxy to make those requests possible. By default the proxy tries to connect to the Wookie server that is deployed to `http://localhost:9080/wookie`, more precisely the request is directed to `http://localhost:9080/wookie/widgetinstances`. To configure the URL, change the servlet's init-parameter "URL" in `web.xml` in `proxy.war` file. The proxy application has to be deployed to the same application server as the test application itself. Thus the JavaScripts running in the test application are allowed to make requests to the proxy, which then can proxy them to Wookie server. If the `proxy.war` is deployed under different name, modifications must be done in the beginning of the `wookieWidgetFactory.js` in `proxyUrl` variable value. A sample deployment model is illustrated by Figure 12.

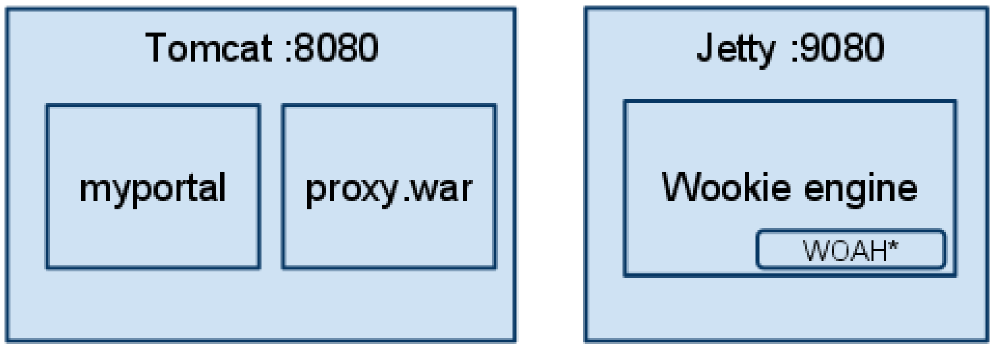


Figure 12. A sample deployment model.

The test application can be deployed by arbitrary JEE container, however, I used Tomcat 6. It is assumed, that the test application runs on `http://localhost:8080/myportal/`. If not, adjustments must be made in test application in the beginning of the `wookieWidgetFactory.js`. More precisely, `tunnelUrl` must refer to `tunnel.html`.

To be able to integrate with Facebook, host name of the application must be in accordance with the one specified in Facebook application’s settings. For example, if the microsite is hosted in mydomain.org, the Facebook application must be configured accordingly. New Facebook application can be registered at the “Create Application” page [55] in Facebook. To connect the test application to the newly created Facebook application, the new Facebook application id must be set to `FB_APP_ID` variable value in the `manager.html`. Further information about the Facebook integration can be found at Facebook Developer's Documentation [51].

To make the test application visible outside the local machine, make sure that all the necessary ports are opened in the firewall.

Appendix B. Source Code

The source code of the Wookie-OpenAjax Hub Bridge and the test application, which were discussed in this thesis, are available on the accompanying CD.