FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Inter-Widget Communication for Personal Learning Environments

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Wirtschaftsingenieurwesen Informatik

eingereicht von

Bernhard Hoisl
Matrikelnummer 0252748

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: ao.Univ.-Prof. Dr. Jürgen Dorn

Wien, 3. Juni 2012

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Inter-Widget Communication for Personal Learning Environments

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Business Engineering and Computer Science

by

Bernhard Hoisl
Registration Number 0252748

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: ao.Univ.-Prof. Dr. Jürgen Dorn

Vienna, 3. Juni 2012

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Bernhard Hoisl
Thaliastraße 10/2/19, 1160 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

To all whom it may concern.

Abstract

In recent time a new trend can be recognised on the Internet in general and especially in learning environments by moving away from monolithic *one-provider-fits-all* to a combinatorial *mixing pieces-together* approach. Mashing-up stands for the re-use, re-combination, and re-organisation of small software artifacts of clearly defined functionality (so-called *widgets*). Subsequently, mashed-up learning systems describe the idea of highly customizable environments shifting substantial personalization possibilities from administrators to the end-users (learners). This shift has significant impacts on software design decisions, especially on efficient data communication strategies of learning management systems.

This thesis presents a technical solution for an inter-widget communication in mash-up personal learning environments, enabling the possibility of different data send and receive strategies, such as, cross-domain push/pull mechanisms, topic-oriented broadcast messaging, or user-centered notification settings and, thus, enabling to model learner workflows in distributed environments. It explains the technical background of the widget concept and why inter-widget communication is beneficial, especially in the area of e-learning.

The outcome of this thesis are new methods and corresponding open-source prototype software artifacts. A proof-of-concept pedagogical use-case of a lifelong learner successfully applying the inter-widget communication facilities in a widget-based learning environment validates the approach.

Kurzfassung

In jüngster Zeit ist ein Trend bei Internet-basierten Lernumgebungen zu Erkennen, der eine Abkehr von monolithischen Systemen, hin zu einem kombinatorischen Ansatz erkennen lässt. *Mashing-up* steht dabei für die Wiederverwendung, mehrmalige Kombination und freie Organisation von kleinen in sich geschlossenen Softwarebausteinen klar definierter Funktionalität (sogenannte *Widgets*). Mash-up Lernsysteme beschreiben hochgradig anpassbare Umgebungen, die die individuelle Personalisierung weg von den Administratoren von Lernsystemen hin zu den Endbenutzern (die Lernenden) verschiebt. Diese Verschiebung hat erhebliche Auswirkungen auf Softwaredesignentscheidungen, vor allem im Hinblick auf effiziente Datenkommunikationsstrategien innerhalb Internet-basierter Lernsysteme.

Diese Arbeit stellt eine technische Lösung für eine auf Widgets basierende Kommunikation für personalisierte Mash-up-Lernumgebungen vor, sodass verschiedene Sende- und Empfangsstrategien für Daten eines Lernenden realisiert werden können; zum Beispiel: Push/Pull-Methoden für unterschiedliche Lerngegenstände, themenspezifische Nachrichten an alle Lernenden oder nutzerspezifische Benachrichtigungen. Damit wird das Modellieren von Workflows von und zwischen Lernenden in verteilten Umgebungen ermöglicht. Es wird dabei der technische Hintergrund des Widget-Konzepts erklärt und warum eine Kommunikation der Lernenden durch Widgets, insbesondere im Bereich des E-Learnings, von Nutzen ist.

Das Ergebnis dieser Arbeit sind neue Methoden und entsprechende Open-Source-Softwareprototypen. Ein pädagogisches Szenario eines lebenslangen Lernenden und die darin erfolgreiche Anwendung der Widget-basierten Kommunikationsmöglichkeiten in einer ausgesuchten Lernumgebung validieren den Ansatz.

Contents

CHAPTER 1

Introduction

1.1 Motivation

A change in perspective can be certified in recent years to technology-enhanced learning (TEL) research and development: More and more learning applications on the web are putting the learner center stage, not the organisation. They empower learners with capabilities to customize and even construct their own personal learning environments (PLEs). These PLEs typically consist of distributed web-applications and services that support system-spanning collaborative and individual learning activities in formal as well as informal settings.

Technologically speaking, this shift manifests in a learning web where information is distributed across sites and activities can easily encompass the use of a greater number of pages and services offered through web-based learning applications. Mashups [8] have emerged to be the software development approach for these long-tail and perpetual-beta niche markets. Core technologies facilitating this paradigm shift are Ajax, JavaScript-based widget-collections, and micro-formats that help to glue together public web application programming interfaces (APIs) in individual services [5].

A PLE is a network of people surrounding an individual with the people in this network making use of artifacts and tools while they are involved in isolated or collaborative activities of more or less planned (co-)construction of knowledge and information. Individuals at the center of the PLE actively and passively modify this environment through actions with the intention to positively influence their social, methodological, and professional competence, i.e., changing their potentials for future action. Though the individual tries to structure the environment, they are not fully in control to design it, as characteristics and affordances of and relationships between the agents in the network (people, tools, artifacts) are not oriented towards a common goal and according to a joint plan.

Although PLEs were first conceived in opposition to virtual learning environments (VLE) and learning management systems (LMS), today they have become rather a new technique and research stream. Their focus is on allowing for more flexible recombination of learning tools, populating them consciously with content and in social networks, and mixing them with elements of the rest of the personal environments surrounding and supporting learners beyond their learning tasks so that the environment is truly *owned* by the user.

Besides empowering users to more consciously co-design their environments according to their needs, this idea is expected to bear several other advantages: it helps to blend formal, non-formal, and informal learning, education, and work; it creates opportunities for the development of rich professional competences such as digital literacy and media competency (see [42]); and it provides a way to cope with the distributed nature of educational resources in a global information society.

Whereas classical LMS had been working with a portal or window metaphor, PLEs resemble rivers, with activity flowing across the screen, moving from widget to widget. Prominent projects such as iGoogle, the Apple dashboard, Microsoft's sidebar, or Netvibes paved the way for this user interface innovation. It is clear that this decomposition of applications into use-case-sized mini-web applications is often still a challenge for usability and further innovation. However, the advantage is that it brings along an increased flexibility and recombinability of technology along very individual learning and web workflows.

Technologically, this new technique is rooted in a software development tradition called opportunistic design, which focuses on rapid application development through the maximisation of code re-use and re-appropriation of soft- and hardware components [8]. Popularly, these approaches are also called "mash-ups", "gluing", or "wiring".

PLEs now take this idea of re-use and re-appropriation onto the next level. Through the deconstruction of learning tools into use-case-sized widgets, through the provision of plug-and-play auto-configuring middleware services, through standardized data sources, and with the help of bootstrapping tools with high usability, end-users can be facilitated in building up a PLE adapted to their needs.

1.2 Problem Statement

At the present time, Internet users and lifelong learners can create their personal view on the Internet by using tools, such as, iGoogle or Netvibes and the like. These *personal environments* allow their users to add and combine different information sources of the Internet in one environment. They can do so by adding so-called *widgets* or *gadgets*—small-scale applications—to their personal environment. According to the W3C, widgets "are client-side applications that are authored using Web standards such as HTML5, but whose content can also be embedded into Web documents" [31]. These widgets en-

capsulate information from particular web-services, like offering a combined RSS feed from several blogs, a search at Wikipedia, overview over latest bookmarks form Delicious, or content sharing opportunities as provided by SlideShare. In that way, users can create a personal view of most interesting information on the Internet. Thereby, these environments inspired research on TEL to extend functionalities to support informal learning processes. These extensions are called in the literature PLEs (see [23,27]). A PLE which mixes and combines services from different providers is called mash-up personal learning environment (MUPPLE; see [10, 40]). MUPPLEs initially support non-formal learning as they require no institutional background, curriculum structure, and are mostly free of use. One objective of using widgets to create a MUPPLE is their high re-usability. Widgets must obey to the W3C recommendation [31] and are therefore designed in a standardized way. This thesis develops as a prerequisite a MUPPLE architecture ready for integration in various (pre-existing) learning and social software environments.

An unsolved problem of MUPPLEs is the ability to interchange information and data between various widgets (maybe coming from different sources) [26]. So far, single widgets have only very limited connectivity possibilities among each other. Browser restrictions and the lack of a standardized communication interface are interfering data exchange. Therefore, this thesis presents a technical solution for establishing inter-widget communication (IWC) to be able to define learning workflows (see, e.g., [7, 29])[1].

1.3 Methodological Approach

The research performed in this master thesis project is influenced by and follows the design science research paradigm [11,25]. The research will produce software artifacts which address important pedagogical and TEL problems: building an open approach for lifelong learners. The design evaluation will be demonstrated with proof-of-concept software artifacts and functional as well as structural test methods. The research contribution has been effectively communicated via scientific publications and open-source availability of accompanying software artifacts. The construction and evaluation of the design artifacts will follow accepted methods for both, TEL and software engineering research. An iterative process for finding the optimal design of research methods and software artifacts has been set up. Revisions of published work are, therefore, integral parts of ongoing research efforts. Furthermore, the pedagogical usefulness will be explained according to a scenario-based use case of a lifelong learner.

[1] Parts of this master thesis have also been published as [2,5,6,16,17,19,24,38,39].

1.4 Outline of Learner's Pedagogical Scenario

In-line with the scenario-based design methodology, the following use case—described from the viewpoint of a lifelong learner—outlines the pedagogical scenario covered throughout this thesis: "My name is Silvia, I have been on maternity leave for five years by now and will start working again next month. I worked as a web developer in a big company, but now I do not feel up-to-date anymore. Therefore, I want to refresh my knowledge of relevant web-based development technologies, especially JavaScript. Hence, I decide to use the widget-based learning environment (LE) offered by my company that will help me to find relevant content with its search system that connects a huge number of relevant resources. I enter my query and various resources are returned, such as textual materials, videos, slides, and scientific papers. Besides, a short definition of my search term is given, as well. Furthermore, the system returns as a result of my query also a fragment of an ontology which shows the relation between the terms of my query and other relevant terms. All results are displayed side-by-side and are connected with each other. When I apply an action in one result frame, the others get updated automatically. In this way, I can find additional material and discover new related resources which helps me refresh my JavaScript skills."

1.5 Surrounding Research Project

The work carried out for and presented in this master thesis was integral part of a three years small or medium-scale focused research project (STREP) entitled "Language Technologies for Lifelong Learning" (LTfLL[2], grant agreement no.: 212578) funded through the European Union's Seventh Framework Programme (FP7) for Research and Technological Development in the Information and Communication Technologies (ICT) theme.

Eleven beneficiaries were involved in the project:

- Open Universiteit Nederland, OUNL (coordinator; the Netherlands)
- Universiteit Utrecht, UU (The Netherlands)
- Eberhard Karls Universität Tübingen, UTU (Germany)
- Wirtschaftsuniversität Wien, WUW (Austria)
- Université Pierre-Mendès France, UPMF (France)
- Politehnica University of Bucharest – National Center for Information Technology, PUB-NCIT (Romania)

[2] `http://www.ltfll-project.org`

- Aurus Kennis- en Trainingssystemen BV, AURUS KTS (The Netherlands)
- The University of Manchester, UNIMAN (United Kingdom)
- Institute for Parallel Processing of the Bulgarian Academy of Sciences, IPP-BAS (Bulgaria)
- BIT MEDIA E-learning solution GMBH and CO KG, BIT MEDIA (Austria)
- The Open University, OU (KMI; United Kingdom)

The surrounding project and its state of the art research approach is sketched in Chapter 2.

1.6 Structure of the Thesis

The thesis begins with an analysis of state of the art research in Chapter 2. Therein, the surrounding project and its research approach is summarized and it is reflected on closely related work which serve as the starting point for the work presented in this thesis.

In Chapter 3 the architectural overview along with the envisioned service approach is explained. Development requirements are mentioned and the concept of widgets is introduced.

Chapter 4 explains the development steps for customizing a widget-based server to allow for IWC. Data sharing strategies are explained and two alternative implementations are shown.

A connector framework has to mediate between widgets served by a widget server and the container platform (the LE). The development of a prototypical connector framework is shown in Chapter 5.

The basic widget implementation requirements are explained in Chapter 6 . To ease the usage of IWC facilities, a supporting widget template is provided which implements corresponding data communication routines.

Chapter 7 extends the introductory pedagogical scenario from different viewpoints. The IWC approach for PLEs is exemplified with a lifelong learner use case.

At last, Chapter 8 discusses the presented approach and its requirements for adoption. Furthermore, possibilities for future work are shown.

The thesis is accompanied by four appendices: the IWC patch for Wookie (Appendix A), the Wookie plug-in installation guide for Elgg (Appendix B), the source-code of the core widget functionalities of the Wookie plug-in for Elgg (Appendix C), and the source-code of the IWC JavaScript library (Appendix D).

CHAPTER 2

State of the Art

2.1 Overview of LTfLL

The project summary according to the proposal's "Annex I – Description of Work" follows:

"The LTfLL project will create next-generation support and advice services to enhance individual and collaborative building of competences and knowledge creation in educational and organizational settings. The project makes extensive use of language technologies and cognitive models in the services.

The research activities are enveloped by activities that ensure common ground in use cases and pedagogically sound scenarios that steer the design and development of the services and guide the validation; a technical infrastructure for the creation and integration of the services and a validation structure that ensures rigorous evaluation in realistic settings, with several languages supported.

The research in the project is organized in 3 themes, each leading to particular types of services and infrastructures:

- In *theme 1* services are developed to establish the current position of the learner in a domain. Services will offer semi-automatic analysis and comparison of learner portfolios to the domain knowledge and continuous modeling and measurement of conceptual development.

- In *theme 2* support and feedback services are developed based on analysis of the interactions of students—using Natural Language Processing (NLP) and Social Network Analysis (SNA)—and textual output of students—using Latent Semantic Analysis (LSA) with contributions from NLP.

- In *theme 3* a knowledge sharing infrastructure is construed that allows comparison and sharing of private knowledge to give rise to new common knowledge and social learning. Ontologies for formal domain representation are combined with social tagging.

The services are expected to result in improved appreciation of learner requirements, leading to better recommendations on study plans and resources. Progress monitoring based on learning activities, rather than on formal assessments, will improve recommendations for further competence building and improved co-construction of knowledge in social and informal learning."

The LTfLL project develops a set of innovative loosely coupled tools that intend to improve the understanding and analysis of students' textual artifacts using language technologies. The LTfLL tools and services are built around specific pedagogic problem statements that relate to contemporary approaches in technology-enhanced teaching and learning. In response to these problem statements, solutions are designed and prototypically implemented, that provide semi-automated assistance to users, helping them to address different areas of their work. In this pursuit, language technologies like LSA and NLP are extensively explored and implemented. All tools are tested and piloted with stakeholders and end-users in a rigorous three cycle validation and feedback process. The project develops a number of applications, comprised of interconnected widgets, encompassing the areas of learner positioning, concept coverage, dialogue analysis and summarizing assistance, as well as formal and informal resource discovery. Furthermore, the project produces a number of detailed scenarios for language technologies as well as templates and methodologies for verifying and validating conceptual designs and software solutions with stakeholders and end-users. Through this method, the tools benefit learning and teaching in the following ways:

- Addressing real educational needs (e.g., qualitative/quantitative learning analytics).
- Supporting the learning process and specific text-oriented learning activities (e.g., chat, reflection, essay).
- Easing known learner/tutor struggles in TEL and mass-education.
- Portable to any LE that is open to widgets (e.g., Moodle, iGoogle, Elgg, Blackboard).
- Flexible to pick and mix to suit learner/tutor needs: use only the widgets you need.
- Interoperability of tools allows threading in support of more elaborate pedagogic patterns.

- Enhanced understanding of the usefulness of language technologies in learning analytics.
- Algorithm and pattern recognition for text linguistic analysis of learner artifacts and interactions.

Learning is seen in LTfLL as a combination of individual and social processes. Feedback from the LTfLL tools is of advisory nature in order to support tutors (and independent learners) in their respective tasks and to allow targeted intervention. By using the LTfLL applications a tutor can spend less time on repetitive supervising and assessing of students, which leaves more time for personal attendance.

Being a research project, tools are of prototypical proof-of-concept nature to demonstrate that language technologies can bring added value and benefit to teaching and learning. Further research and development after the project has been planned and is expected to happen. Other tangible outcomes include a common semantic framework to auto-enhance formal ontologies with folksonomy data, and the development of IWC. Improved algorithms and a specialist annotation tool are further results of our work. Academic results are published as research papers in corresponding journals and conferences.

2.2 Objectives of WUW

The work of WUW (Wirtschaftsuniversität Wien) forms the backbone of the project to guide and support the research and development (infrastructure workpackage, WP2). The main tasks of WP2 are to guide and facilitate the technical aspects, i.e. the development process of the language technology WPs and to assure the integration of the resulting services into the PLE so they can be used in the validation. The first task is related to the requirement to develop the various language services in the most efficient way. This means that utilities and resources that are shared (designed and developed, or selected) are made available at one central place. The second task assures that a learning infrastructure is set-up to include the services and that the services developed will comply with a standardized way to be able to add them to *any* services-based environment and to assure the further sustainability of the services. The decision which infrastructure to use will be based on a combination of technical criteria and validation site requirements. The results of this WP will be iteratively refined in line with the cycles specified.

Regarding the e-learning infrastructure, WUW will be responsible for the set-up, maintenance, technical validation, and documentation of the e-learning infrastructure. The first task of WP2 will be to coordinate the selection and set-up and if required the adaptation of the basic e-learning infrastructure and to document inline with this the technical guidelines for the services to enable their integration. Subsequent, WP2 will

take care of the adaptation, integration and the technical validation of the result of the integration of the services as they become available through the other WPs.

The integration activities deal with the requirements and educational perspective of each of the services resulting from the language activities, their integration in a learning environment, their usage and validation through which the further development and adaptation of the services can be directed. The main outcomes are the following:

1. Creating a common ground by defining and describing a set of use-cases.
2. To define the educational perspective of the services, a scenario-based design methodology is defined. The methodology specifies how the pre-pilots (showcases) and services should be developed in the project. A set of showcases is defined for the language activities.
3. To enable and prepare the services for their integration, a concept, guidelines and the basic set-up of the infrastructure is developed that allows for an efficient and effective integration of existing tools by differentiating system components to be integrated along three layers (data, services, and widgets) and by targeting mash-ups as the integration technique of choice to glue together these components.
4. To enable access to generic NLP utilities and resources, a service-oriented, web-based framework is implemented and documented. Resources, including lexicons, corpora, and grammars, are surveyed and made available.
5. The overall validation approach, validation criteria and its relationship to the adopted scenario-based design approach is defined and described. Pre-pilot (showcase) validations are planned and their validation started.

The work presented in this thesis adds research contributions related to items number 3 and 4.

2.3 Related Work

In the PALETTE project[1] an approach has been proposed, based on a model called *3A*. The three As stand for actors, activities and assets. An actor is producing an asset being within an activity. An actor is a person, a software agent or any other intelligent object. An asset is a document or a collection of documents or items: discussion thread, wiki page, image album, and the like. An activity describes a formalization of a common objective to be achieved by a group of actors: representation of a tangible or abstract space. The structure is similar to a graph: nodes (AAA, so called entities) are connected with several directed or undirected links with a specific type and weight [1].

[1] `http://palette.ercim.org`

A different approach has been developed in the context of the iCamp project[2]. At the core of this approach stands LE design which manifests in a learner interactions scripting language (LISL) and a prototypical implementation. LISL gives end-users the possibility to directly manipulate the composition of their PLE. A simple learner interaction model has been deduced to describe the physical and social environment of learners. The activities an actor is engaged in are composed of actions that include tools, artifacts (objects), and other actors. A learning situation is represented by an activity that consists of actions that refers to objects and requires tools. With the help of LISL, learners manipulate actions, artifacts, and tools. Each action is bound to an artifact and at least one tool and produces one tangible or intangible outcome [40].

Furthermore, there are different techniques for sharing data between clients and servers in general—which are out of the scope of this thesis—and for widgets especially. Related work encompass the myWiWall portal of the PALETTE project which has taken a first step in making IWC possible and combine it with drag-and-drop facilities [26]. The portal makes use of client-side JavaScript functionalities embedded in the widget host container. However, this approach is lacking the possibility to tailoring cross-widget communication to a user-defined audience, for example, all widgets of one user or all users that have one particular widget enabled.

Google proposed a Gadget-to-Gadget communication framework [7] where a publisher widget needs to name in the manifest XML file any subscriber widget that is interested in receiving status updates. This approach is very limited and not easily extensible if other widgets should receive updates as well.

Other approaches are dependent on the W3C HTML5 working draft, defining an API for cross-document messaging [29]. Firstly, the draft specification is described more generally and not tailored specifically to IWC and, secondly, HTML5 is still no web standard and not fully reliable at the moment. Furthermore, these specifications do not focus on a user-tailored IWC and are lacking possibilities for modeling learner workflows. In this thesis, such a user-tailored IWC is presented that enables different communication styles between widgets owned by certain user groups, such as, teachers and their students, thus, enabling basic service orchestration.

2.4 Contributions to the state of the art

In this thesis it is reported on the IWC strategy to be able to link functionalities of individual tools and across different tools (threading approach) with a dedicated client-server model. On the server side, a corresponding widget engine is used; for the client side, a JavaScript library is built, which is integrated in a widget template. By instrumenting these software artifacts, IWC especially targeted towards usage scenarios for PLEs can be established. These learner-oriented scenarios focus on communication

[2] `http://www.icamp.eu`

strategies currently not covered through former approaches and not being feasible with standard technologies (see, e.g., Section 2.3). The focused IWC includes data send and receive strategies, such as, cross-domain push/pull mechanisms, topic-oriented broadcast messaging, or user-centered notification settings which cannot be realized with current technologies and are not discussed in state of the art research publications.

The IWC approach presented here is tested in different scenarios (one educational use-case is exemplarily presented in Chapter 7). A demonstrator platform for the tools and for validation is chosen, though widgets and their IWC can be deployed and used within every web-based LE. Although, for ever host LE, a dedicated connector framework has to be implemented to interface with the widget-connecting plugin. In addition, a second widget engine and connector framework for another platform is implemented to show transferability of the presented approach across different technologies.

A side outcome of these developments is the application of a single-sign-on method for widgets and its integration into the showcase platform. Therefore, the OpenID standard is utilized for authenticating a user against an identity provier. Here, the contribution is a method of passing the user identity through from the container to the widget, thus, allowing any associated widget to authenticate with the OpenID identity server. Given that users trust their LE provider, the authentication can be accepted automatically for all widgets, once a user is logged in on the identity server.

CHAPTER 3

Setting the Stage: Service Approach

3.1 Development Requirements

The approach targets a loosely-coupled widget-based integration of web-services. Certain technological decisions obviously follow therefrom, for example, the front-end presented to the learner should be rendered via a web-browser and the communication between the different layers should be done via HTTP.

Designing widgets viewable in a web-browser is ideally done with pure client-side run-able components, such as, HTML, CSS, and AJAX (but Flash or other technologies for designing rich-Internet-applications should not be neglected). Widgets are integrated into PLEs by using a container, ensuring the ability to plug-in and to communicate between widgets. Style sheet definitions (CSS) offer possibilities to configure the widgets to expose a common look and feel. Web-services are called over standardized network protocols (HTTP) with distributed software architectures (RESTful services). Data exchange between provided web-services is done using structured message formats like XML or JSON.

For integrating the services and to test their interaction possibilities, it is necessary to have a PLE platform for showcasing the developed software artifacts. Furthermore, a widget-engine needs to serve the developed widgets and these widgets need to be interfaced with the LE.

3.2 Architectural Overview

As for this thesis, different kinds of software artifacts with the help of multiple and varying technologies are developed. The integration approach chosen must allow for combining these artifacts with a high degree of individual freedom in software system

design choices. Consequently, the MUPPLE approach [40] has been selected as the integration strategy. This allows to plugin loosely-coupled software artifacts in an integrative environment, thus, generating a set of customizable services. On the one hand, the advantage of this approach from the learner's point of view is that heterogeneous software systems are plugged into a single environment: they can be arranged individually but can feel and look like one coherent software system. On the other hand, benefits for the software developer and system administrator are—beside those already mentioned above—that a modularized system like this can easily be plugged into different platforms with less effort, making it highly interoperable and reusable. Providing services with standardized interfaces brings the question of an integration strategy to a higher level, eclipsing technological decisions on programming languages or database management systems.

As can be seen in Figure 3.1, the system design is following a classical three-tier server-client architecture with its data, application, and presentation tiers. Furthermore, the architectural design makes use of an additional middleware layer connecting the application logic and the graphical user interface (GUI). For the integration of the heterogeneous services, a widget based approach was chosen. Web widgets are "client-side applications that are authored using Web standards [...], but whose content can also be embedded into Web documents" [31]. This means that we do not care what happens in the other layers, as long as data is delivered from the application tier in a standardized way and can be displayed using widgets as front-ends. In a next step, these widgets can be integrated in nearly every web-based LE[1].

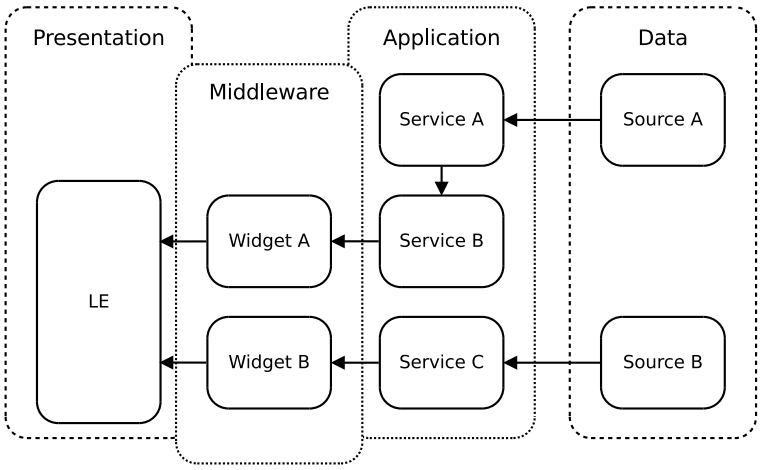

Figure 3.1: Four-tiered architectural approach.

[1]Generally, embedding widgets is not limited to the web, but can be done in desktop applications, as well. Nevertheless, in this thesis focus is set only on widgets being deployed on the web.

A simplified but illustrative example of this approach for integrating services into an existing platform can be seen in Figure 3.1. Two widgets are plugged into an existing LE where learners can use them. The widget container relies on two web-services (B and C) for generating results, which are then displayed to the learner by the widgets. Service B fetches data from a second Service A which itself is fed by data from Source A. Service C fetches data from another external source, giving back data to the Widget B. The results of both widgets A and B are rendered visible for the learners in their LEs.

This simple example can be extended: more than two widgets can be integrated in a chosen platform, communication methods and messages can be standardized, and web-service interfaces have to be defined as well as made public, for instance, through a directory service (specified via, e.g., the WSDL [28]). One advantage is immediately apparent when looking at Figure 3.1: all layers are loosely coupled, which means they are connected over a network and modularized in their own working scope. Widget integration in existing platforms is state-of-the-art in achieving PLEs. With the architectural design, a possible spatial distribution of software development contributions is taken into account. Software artifacts are not seen as static and stand-alone programs, but as interactive services offering specific problem solving mechanisms to other services. Different technological applications can, therefore, interact and can be build on top of other developed services—thereby gaining greater power.

The biggest architectural difference compared to traditional monolithic learning systems is that all components providing functionality are completely detached from the underlying software system. This having the advantage of being a very flexible solution and is a benefit in case of software reusability.

By decoupling software components and providing well-defined interfaces, high reusability and standardization can be guaranteed. At the data layer any database management system can be used as well as any data storage method and query language. In this approach the application logic relies solely on web-services. The services interface, on the one hand, with the data sink, on the other hand, provide input to the widget-based middleware. The widgets act most prominently as well-defined and encapsulated GUIs with a possibility to be used in various environments. This is due to the fact of their standardization in terms of deployment, structure, and behavior. In this thesis an approach for embedding widgets in a LE is shown, thus generating a very flexible PLE with mashed-up content and services retrieved from the interacting widgets. As these widgets are solely based on web-standards (like (X)HTML, CSS, JavaScript etc.) their look and feel can be easily adjusted according to the corresponding host environment. Furthermore, by providing a JavaScript library for inter-widget-communication (IWC) it is possible to deploy basic workflows for a single user and a group of users—even for widgets served by one widget engine and integrated in different host environments.

3.3 Introducing Widgets

Widgets are small applications that are embedded in a framework or widget engine. There are two possibilities provided: Widgets that are able to communicate directly with the server and are interactive or widgets that get their information periodically and are not interactive. To stay as flexible and as open-ended as possible the output is a standardized data format (like XML). In this way, information remains independent of any tool or widget. Furthermore, this offers the possibility to use output of one widget as input for other applications.

Most of the time, widgets are tools or some kind of help or service applications. Widgets first arose in operating systems, such as, Apple's dashboard widgets. Parallel to this development was the appearance of web widgets, mainly to serve as a container for information from any external source. In the world of web 2.0, widgets are often used to embed photos or videos, as with Flickr or YouTube.

Widgets are mostly written in HTML and JavaScript. A widget engine is needed to host a widget in an environment. There are a lot of such environments already; the natural choice being another webpage, an approach that, for example, iGoogle follows. But even modern operating systems support widgets natively, for example, Microsoft Window Vista and Apple's Mac OS X. On the Internet there are certain platforms that provide this functionality such as iGoogle, Facebook, Netvibes, Pageflakes, and others.

On the Internet, widgets serve mostly as visual interfaces to web-services in a service oriented architecture (SOA). A widget consists of a client-side programming logic and a visualization layer to view information given by a web-service. The web-service on the server only provides an API to the widget to access data or other programming logic. The web-services contain no commands to visualize any data; most of the time they only return XML data (or the like) back to the widgets.

At the moment the different types of widgets require specific widget engines. Thus, a Google widget is not deployable in Netvibes and vice versa. This is a disadvantage that the W3C consortium wants to overcome by a collection of specifications in order to achieve a common widget standard and guarantee the interoperability among the different widget engines.

3.4 Interoperability through Web Standards

The main challenge is to create interoperability along two lines of work: interoperability of the learning tools (as interconnected services) and interoperability of the learning tools for their integration in LEs. Interoperability is a property that emerges, when distinctive information systems (subsystems) cooperatively exchange data in such a way that they facilitate the successful accomplishment of an overarching task [41]. The purpose of this sought interoperability is manifold: it is to allow for integration of hetero-

geneous services; it is to allow for their flexible recombination to serve the fulfillment of learning tasks; it is to pave the way for commercial and non-commercial uptake.

Therefore, the principles of the design approach is to build a light-weight, easy to handle, multifunctional architecture focused on interoperability of services and their re-use. Hence, the primary goal is to build a scalable solution for independent deployments of software components with standardized interfaces. In-line with these requirements it was chosen to develop web-services communicating in a RESTful way. REST (Representational State Transfer) is based on and makes use of native methods of the HTTP protocol (e.g., GET, POST, DELETE etc.). Data exchange between provided web-services and the presentation layer is done using structured message formats like XML or JavaScript Object Notation (JSON).

To ensure transferability—i.e., to ensure that the developed software will run on all major systems—minimum requirements on software and hardware components have to be defined. As the developed learning tools are rendered using a web-browser, some generic guidelines can be defined. By optimizing user interfaces for Microsoft Internet Explorer, Mozilla Firefox, and Google Chrome, typically a range of clearly over 90% of all Internet users is covered [35]. Furthermore, nearly all users have JavaScript enabled [37]. If web-based software is designed for a screen resolution of 1024x768 pixels and higher over 99% of the users are covered [34]. Regarding operating systems, over 84% of the users are working on a Microsoft Windows system [36]. The XHTML and CSS standards can be validated using W3C validators [32, 33]. The tools need to be able to cooperatively exchange data in order to support the successful accomplishment of the envisioned use case.

CHAPTER 4

Customizing a Widget-based Server

4.1 Introducing Wookie

Wookie [46] is a widget engine that implements the W3C widget recommendation [31]. It is designed to provide widget run-time functionality to a wide range of applications. Web applications that integrate widgets are called *widget containers*. Widget containers take care of support activities like user management, access rights, content management and so on, while the Wookie engine supplies the functionality to add widgets to the mix.

Wookie was chosen for several reasons: it is standard compliant with the W3C widget recommendation, has a large educational community, was developed by former EU project TenCompetence, is open-source, is an Apache Incubator project, and has plugins available for different LEs.

When a widget is created, direct interactions are done only with the widget engine, and not the container—the container may sometimes set preferences a widget may use (like the user's name to be displayed in a widget), but for the most part the services offered by the widget engine are called. Developing a widget for Wookie means that it can be delivered in a range of web platforms including software like Wordpress[1], Elgg[2], Moodle[3], and so on. Wookie enables widgets to be used in these applications through the use of plugins. This means there is some code that is native to the web application's framework that can talk to Wookie and request widgets [43].

[1] `http://wordpress.org`
[2] `http://elgg.org`
[3] `http://moodle.org`

4.2 Data Sharing Strategy

The context of sharing data between widgets is a crucial one because it is targeted not only on a side-by-side widget integration, but also on the possibility to model basic workflows. Therefore, widgets must have the ability to share data between certain contexts and must also be able to trigger and listen to events invoked by different widgets. The Wookie engine uses Comet style[4] to send events and share data according to a *sibling rule*. This means that the state is shared between widgets which have (a) the same GUID (i.e., same type of widget), (b) the same API key (i.e., originating from the same system or application), and (c) the same shared data key. This is perfectly fine if, for example, it is targeted to implement a chat widget where different users can interact with each other using the same type of widget in the same application. But as further data sharing mechanisms need to be achieved, either the Wookie engine functionalities need to be extended or alternative communication techniques have to be utilized.

Different communication strategies (HTML 5 postMessage API, AJAX over service back-end, session/cookie based etc.) are investigated—every method has its advantages and disadvantages. After weighing the pros and cons, it is decided that it would be best to add needed functionalities to the Wookie engine, instead of using other external techniques. This method has the advantage that Wookie is further developed with functions where requirements for IWC have already arisen from the community some time ago. As the code changes needed for implementing the functionality in Wookie seems manageable, it is thought that this solution is also the most cost effective. Furthermore, it is a good idea to have all widget based functionality provided by only one software package.

In the scope of the provided services, data need to be shared mostly between different widgets of a single user. This means—by heading towards the option to adapt the Wookie source code—that a different sibling rule has to be implemented. Therefore, Wookie's data sharing policy has to be modified so that data sharing is restricted only to (a) the same API key, (b) the same shared data key, and (c) the same user ID (or user session). This implies replacing the restriction of the same GUID with the same user ID (or user session). Therefore, the Wookie REST calls has to be adapted to transmit the Elgg user ID as `shareddatakey` instead of Elgg's GUID of dashboard widget instances.

Wookie implements some additional features, which are extensions of the W3C widgets specification [31] and not supported in the same way by other widget engines. For IWC, Wookie's shared data API is of special interest. Its methods allow not only for storing and accessing data throughout various instances of a widget, but also to invoke specific events after a shared variable is set (like updating a widget or displaying different contents and the like). The methods `sharedDataForKey()`,

[4] Basically, a long-held HTTP request that allows a web-server to push data to a browser [3].

`setSharedDataForKey()` and `appendSharedDataForKey()` are of paramount importance. As these are asynchronous, a callback handler must be provided.

An explanation of attributes, methods, and events provided by Wookie for working with shared data follows:

- `Widget.instanceid_key()`: The read-only identifier generated by the widget engine for a widget instance.
- `Widget.sharedDataForKey(key, cb)`: Returns the value of shared data for `key`, or *undefined* if there is no match. When completed, invokes function `cb` with the return value.
- `Widget.setSharedDataForKey(key, val, cb)`: Sets the value of shared data for *key* to *val*, overriding any existing value. If there is no match, creates a new shared data entry for `key` with `val`. When completed, invokes `cb` with the return value.
- `Widget.appendSharedDataForKey(key, val, cb)`: Appends the value of shared data for `key` with `val`. If there is no matching key, creates a new shared data entry for `key` and sets it to `val`. When completed, invokes `cb` with the return value.
- `Widget.onSharedUpdate`: Called when a shared data entry is updated with the shared data key affected.

The communication flow and update mechanisms are based on the Direct Web Remoting (DWR) engine. DWR allows for the interaction of Java code on the server (in the case of Wookie: managing data storage and distribution procedures) and JavaScript code executed within the browser. The necessary JavaScript sources are generated automatically, as well as the marshaling of data. Normally, the update notice on shared data is sent to the client as a response to the next HTTP request (*piggyback*); nonetheless, DWR provides reverse AJAX functionality (*push*) to publish updates to definable groups of clients rapidly in order to ensure an effective update mechanism. Wookie is taking advantage of the DWR feature reverse AJAX by using the polling method (the browser sends a request to the server in regular and frequent intervals to check if there are some updates) for pushing data from the server to browsers.

Revision number 1.023.765 retrieved from Wookie's SVN repository was used to implement the modifications. In the widget manifest file, the push-feature has to be switched on, which can be done by inserting the following line in the `config.xml`: `<feature name="comet" required="true"/>`.

The communication flow and JavaScript functions to implement the use of shared data among different widgets as well as event notifications are displayed in Figure 4.1.

The `init()` function is called on load and retrieves at first the shared data key for the specific widget instance, which has been defined previously during the widget initialization request. This key is necessary to detect update notifications intended for a specific widget instance. The callback function `initSharedKey()` is called with the key as parameter—it simply sets it to the globally available variable `sharedKey`.

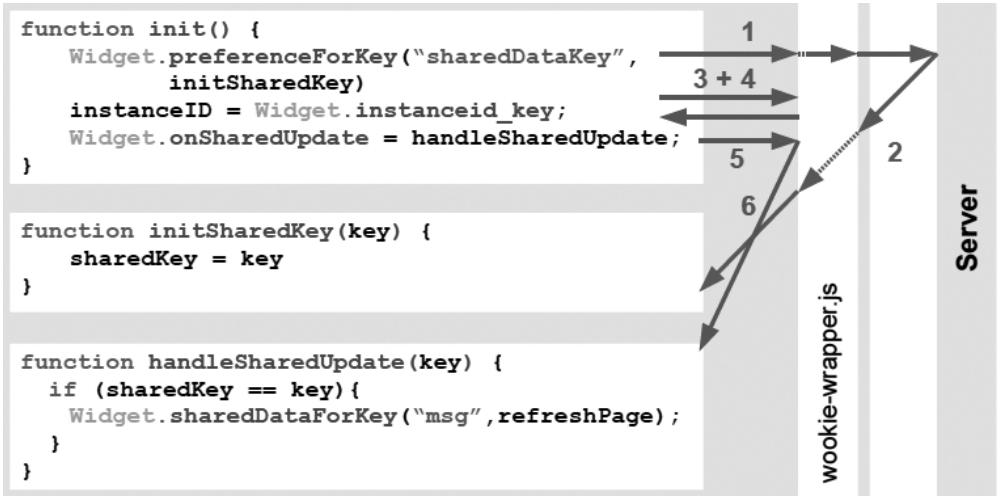

Figure 4.1: Code example of using the shared data interface [17].

Next, `instanceID`—thus, a unique widget instance identifier—is retrieved and set using the `Widget.instanceid_key` function. Finally, the update handler is registered. `Widget.onSharedUpdate()` is a function called from the server via remote execution when performing a shared update notification. In this case, the function is assigned the value of `handleSharedUpdate`, with the shared data key as parameter. Hence, if on a shared update, the update key is identical to the instance's shared data key, the corresponding if-clause in function `handleSharedUpdate()` returns true.

`Widget.sharedDataForKey()` then retrieves the shared data value for the key `msg`. At last, the callback handler `refreshPage` is invoked doing something like refreshing the page or updating its content (not displayed here). The `Widget` object (as well as helper functions for browser type and version detection) is provided by Wookie through a JavaScript wrapper file called `wookie-wrapper.js`.

If we assume to have two widgets that need to talk to each other, an exemplary workflow can be seen in Figure 4.2. There, Alice initiates the shared update by setting a new value to the shared variable `msg`. As in this example, both Alice and Bob are listening for shared updates, also both widgets are retrieving the newly set variable from Wookie. In the end, Wookie serves the variable to both clients and a call-back handler (`refreshPage()`) is invoked. In both cases, it updates the page with the data retrieved. Of course, it is also possible that just one or more than two widgets

are listening to shared updates and filtering the variables that are supposed to trigger an event. Events are defined on a per-widget basis. That means, on a shared update, different behaviors of various widgets can be realized.

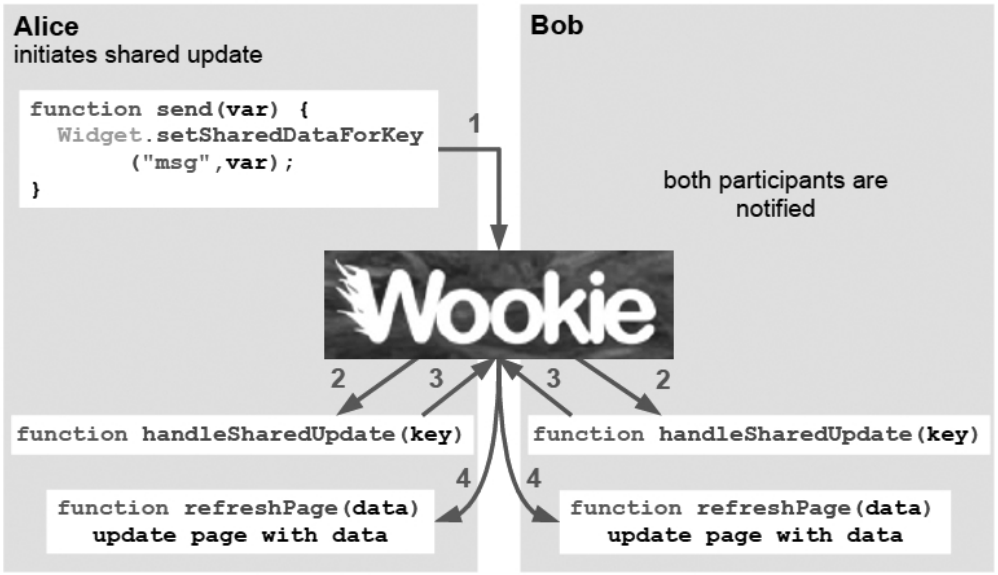

Figure 4.2: Workflow example using the shared data interface [17].

4.3 Inter-Widget Communication Facility

Wookie does not support IWC by default. Communication is limited to inter-instance data sharing. This means, data can only be shared between one specific widget instantiated by users of one widget container, but not, for example, between two or more widgets of one user or between different widgets of different users. To go beyond the current Wookie approach, the Wookie engine had to be customized by extending and adding additional IWC possibilities to overcome the current user restrictions.

A prototype for IWC facilities where developed within Wookie, but with the disadvantage of creating a branch specifically targeted on data sharing purposes and not in-line with the development trunk. Therefore, no benefits could be retrieved from new improvements, such as new functionalities or bug fixes committed from the community. This duplication was eliminated by integrating IWC specific functionalities in the head revision of the Wookie development version once again taking into account former implementations and improving them significantly. The outcome is a more generic approach of data sharing strategies.

By learning from the first approach (the prototype), the code could be simplified while making it target a wider range of scenarios. The main application—aiming at notifying all widgets in one user space—has not changed, but other strategies can be applied, as well. Therefore, a Wookie administrator is now able to couple widgets by defining matching parameters. All widgets having these variables in common are treated as belonging together, therefore, share data and are notified of updates.

In the file `widgetserver.properties` an administrator has to state all coupling IWC variables. Currently implemented are the following parameters: `sharedDataKey`, `apiKey`, `idKey`, and `userId`. With a few lines of code it is no problem to add new coupling parameters to the existing ones. Any combination can be used for binding widgets together:

- `sharedDataKey`: The key generated by an application representing a specific context (e.g., a page, post, section, group, user or other identified context).
- `apiKey`: The key issued to a particular application.
- `idKey`: An identifier representing a single widget instance.
- `userId`: An identifier (typically a hash rather than a real user ID) issued by an application representing the current viewer of the widget instance.

The purpose of our use case scenario in Chapter 7 fits a coupling of `apiKey` with `userId`. Therefore, data sharing is restricted to widgets of one user (`userId`) in one specific application (`apiKey`). The `apiKey` is specific to the Elgg installation and the `userId` represents a user's unique ID in Elgg (as defined through the Elgg plugin, [14]). Hence, all widgets belonging to a logged-in user in Elgg share data and receive notifications. Another broader linkage strategy could be to just use the `apiKey` and letting all widgets of one application share their data. If someone would like to completely avoid sharing data, the coupling parameter has to include the `IdKey`. As it identifies a single widget instance, no other widget will match this criterion.

The workflow of the IWC implementation is to firstly find out all widget instances matching the coupling statements. Then, shared data is duplicated and distributed to all instances found, combined with a notification about the updates. Therefore, modification and extension is limited to setter methods of data sharing functionalities together with notification behaviors, but could leave data fetching procedures of widget instances basically as they are.

A patch was provided for Wookie (see also Appendix A) and four test widgets using the newly development IWC functionalities (see [47]). The first two test widgets simply store and retrieve strings inserted by a user and display them. The other two widgets use the widget template implementing the IWC widget library for storing and fetching shared data using JSON (described in detail in Section 6.4). Therefore, the

developments are getting tested not only within this thesis project, but also by a wider community of interested people. The patch with its test cases is currently under review and it is expected that it is going to be integrated in the development version of Wookie, soon. Unfortunately, there have been some delays as a first official release of Wookie is planned at the moment and integration of new functionalities is generally postponed after the release. In the meantime, the modified Wookie Server capable of IWC can be obtained from [15].

4.4 OpenACS Widget Server

To test interoperability of services, and especially widgets, a prototype Widget server has been developed being able to partially replace Wookie. Therefore, core functionalities of Wookie were implemented in the Open Architecture Community System (OpenACS[5]) using eXtended Object Tcl (XOTcl[6]) as a scripting language. OpenACS is an open-source web application framework running on AOLserver[7] which is used with PostgreSQL[8] as its database management system. The OpenACS package (named *xotcl-widgets*) implementing all functionality can be obtained from [4].

The OpenACS implementation features both, a server able to deliver widgets and a connector framework serving as a plugin for OpenACS itself. The implementation is based on these three OpenACS packages: *xotcl-core*, *xowiki*, and *xowf*. Furthermore, Wookie was used as an example and conformance to its implementation was established (especially the REST API) as much as possible. However, just a prototype was developed integrating only core functionalities of Wookie. A widget upload workflow was defined for being able to deploy widgets on the server. Once available, all existing widgets can be viewed in a gallery type of style. An XML representation is provided as used, for example, by plugins to select a widget for instantiation. Therefore, a list of widgets can be generated which can be displayed, for instance, with the Elgg plugin in the same way as with Wookie (REST API calls are identical). Instantiation of a specific widget is also done with the same API calls as with Wookie, therefore, the Elgg plugin can be used without any changes. Basic support for IWC is provided by handler functions communicating with parts of the DWR library, re-implementation of internal Wookie specific methods (e.g., `setSharedDataForKey()` etc.), and interface implementations (e.g., Google Wave API).

Furthermore, the package defines also some test cases for the connector framework and IWC. Beside tests for the integration and display of widgets in OpenACS, data sharing can be tested according to the same `shared_data_key` (specified at widget

[5] `http://openacs.org`
[6] `http://www.xotcl.org`
[7] `http://www.aolserver.com`
[8] `http://www.postgresql.org`

instantiation) or the same `wiki_page_id` (specified by OpenACS) or combinations thereof.

CHAPTER 5

Developing a Connector Framework

5.1 Introducing Elgg

It was chosen to use Elgg as a showcasing platform to integrate developed widgets. The platform decision is not a particularly critical one as it is very easy to plug the widgets into other platforms at any stage in their development. The decision was made because Elgg was originally developed from an educational context perspective, has a large supporting community, is open-source, and has an existing Wookie plugin the developments are based upon. The choice of Elgg is also supported by other factors. As the focus of this thesis lies on software development for supporting lifelong learners, the underlying platform has to handle user management, access control, community networking and so on. Elgg offers a wide range of modules capable of these issues and is easily extendable.

As an identified starting point, a Wookie plugin for Elgg exists, but is outdated. Both, the Wookie engine and Elgg were further developed and the plugin interface did not match the new architecture. Therefore, the plugin has been updated based on the existing one by taking the newest Moodle plugin as a template [45] (which was by that time the most evolved). The plugin was designed to work with the newest versions of Elgg and the Wookie engine (in version 0.8). A first release of the plugin [12] was reviewed by professionals and resulted in two updated versions 2.1 [13] and 2.2 [14], respectively. In the meantime further developments were done and a new version is currently being prepared for another release. Evolvements of the plugin are reported in this thesis. Furthermore, the plugin installation instructions can be found in Appendix B.

As Figure 5.1 shows, the plugin allows for a fluent integration of widgets in Elgg. A drop-down menu was implemented that enables users to choose among all available widgets provided from the Wookie engine. The height and width of the widget to be dis-

played can be specified by the user, as well. If the dimensions are set, the values override the default manifest file definitions of the widget's configuration file (`config.xml`). Access to the widget can be restricted to logged in users only, to specific groups, or just to friends. In addition, the widget can be declared private (only visible to users themselves) or public (visible to everyone).

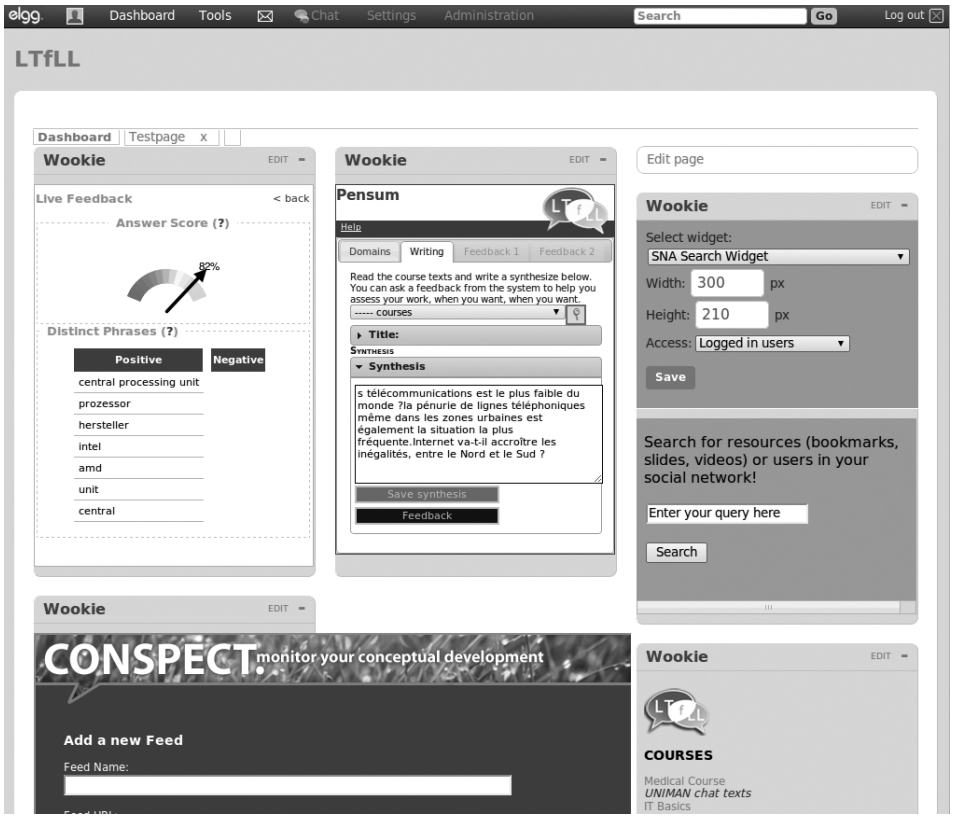

Figure 5.1: Snapshot of a (random) selection of widgets integrated in Elgg.

With the help of an additional plugin [22], multiple tabs allow to arrange selections of widgets into separate pages. In Figure 5.1, the tabs are visible at the top: *Dashboard* and *Testpage*. New tabs can be created by the user; arrangements can be changed by dropping widgets or adding new ones from the widget directory.

5.2 Instantiating a Wookie Widget in Elgg

The developed Elgg plugin for widget integration uses the Wookie REST API. The API provides functionalities such as requesting a new widget instance, getting a list of available widgets, adding participants to a widget instance, setting properties, working

with shared data, etc. To date, only the core functionalities have been implemented in the Elgg plugin (see also Appendix C). As an example, we show a request statement for a new widget instance as it is done by the function `getWidget()` in the plugin's file `functions.php` (see Listing 5.1).

```
1 http://augur.wu.ac.at:8888/wookie/WidgetServiceServlet?
2   requestid=getwidget&
3   api_key=56eFLpcT7UhhK075QjbkxVF5ZDY.eq.&
4   servicetype=&
5   widgetid=http://augur.wu.ac.at/widgets/conspect&
6   userid=163&
7   shareddatakey=wp5
```

Listing 5.1: Request a new widget instance.

Before the plugin is capable of invoking such a request, the URL to the Wookie engine as well as the API key must be set through the administrator web-interface of Elgg. The API key is a unique key identifying an individual web application and must be requested from the Wookie engine by using its administration interface. As most parameters of the above REST request are self-explanatory, only some background information not visible on first sight is provided here:

- `servicetype`: Widgets are divided in service types each serving a different purpose. This parameter is important only where it does not matter which individual widget should be displayed as long as it is of the defined service type, e.g., *chat*.
- `widgetid`: If no servicetype is defined, the unique URI of the widget (global unique identifier, GUID) must be provided instead.
- `userid`: An identifier (e.g., a user ID) issued by an application, representing the current viewer of the widget instance. In the case of Elgg, the global variable `$vars` stores application-specific values (like user ID) and is passed through to the plugin.
- `shareddatakey`: The key generated by an application representing a specific context. By default, different widgets are not able to share state information due to privacy concerns. Widgets having the same `shareddatakey` (along with some other parameters) can share data that is stored in the database of the Wookie engine (details are discussed in later sections).

By invoking this request, Wookie will return an XML answer (see Listing 5.2) consisting of the URL of the widget instance, the title of the widget, the default height and width, and optionally a *can maximize* flag according to the widget's manifest file. This information is used by the plugin to display the widget in Elgg. For further information on wookie's REST API it is referred to [44].

```
1 <widgetdata>
2   <url>
3     http://augur.wu.ac.at:8888/wookie/wservices/augur.wu.ac.at/
          widgets/conspect/index.html?idkey=U6Eyjg.pl.sJeAoUKszCitC.pl.4
          fPd5w.eq.&proxy=http://augur.wu.ac.at:8888/wookie/proxy&st=2%3
          A2%3Ahttp%253A%252F%252Faugur.wu.ac.at%252Fwidgets%252
          Fconspect%3Awookie%3A%252Fwookie%252Fwservices%252Faugur.wu.ac
          .at%252Fwidgets%252Fconspect%252Findex.html%3A0%3AU6Eyjg.pl.
          sJeOaUSkzCitC.pl.4fPd5w.eq.
4   </url>
5   <identifier>U6Eyjg.pl.sJeOaUSkzCitC.pl.4fPd5w.eq.</identifier>
6   <title>Conspect</title>
7   <height>480</height>
8   <width>640</width>
9 </widgetdata>
```

Listing 5.2: XML answer for instantiating a widget.

5.3 A Note on the Limited Space of Widgets

As already mentioned, the connector framework allows widgets served by the Wookie engine to be displayed in Elgg. As widgets are designed for small applications placed side-by-side, their display dimensions are usually quite small. By looking at Figure 5.1, one can see that especially horizontal space is limited. Of course, it depends on how much widgets and on how much columns a user wants to display, but generally widgets' GUIs have to be carefully designed.

To solve the problem of space limitations, a functionality is provided which allows for uncoupling widgets from the dashboard and enlarging them (even to full screen). This is made possible by integrating the Highslide JS JavaScript thumbnail and media viewer (see [20]) into the connector framework. Therefore, a small icon is placed in the general header of dashboard widgets, next to the edit and minimize buttons. Clicking on it uncouples the widget from the dashboard and displays it in an overlay window. This can be done with an unlimited number of widgets. The uncoupled widget is resizable and draggable. Navigation between widgets in a dashboard can be done through the next and previous arrows or by using left and right keys.

5.4 Authentication with OpenID

Digital traces left in learning obviously touch upon data of a very sensitive nature about the individual. To provide adequate protection of this desired privacy the use of OpenID[1]

[1] http://openid.net

for authentication and access control using capability-based credentials for authorization are investigated. The main advantage of OpenID, thereby, is that it provides possibilities for anonymity. Individuals are not restricted in their choice of an identity provider and they retain control of what they want to expose in their digital profile.

When logging onto a system using an OpenID, the system contacts the identity provider to generate a shared secret. Then, the user is redirected to the identity provider to validate the request (log onto the identity provider, grant access using the shared secret). Given that everything went fine and the user validated the login request, the user then enters the system—authenticated with her/his OpenID.

The Wookie plugin for Elgg described in this chapter supports basic OpenID authentication. Elgg does not support OpenID logins out-of-the-box, but a plugin exists providing these functionalities (see [21]). The implementation passes through the OpenID from the Elgg environment to the Wookie container and further on to the widget. This means that the authentication mechanism is not part of the Wookie engine core functionality, but in the hands of the widget itself. This method was much more cost effective compared to implementing the whole OpenID authentication technique in Wookie, but it has the trade-off that every widget has to integrate OpenID functionality on its own. But many open-source libraries for nearly every programming language exist.

Since the OpenID protocol does not force the identity provider to release any private information, such as, the email address, the OpenID identifier itself is the only remnant stored within Elgg. This identifier is stored as an Elgg metadata entity in the database and is read from the plugin by using the global Elgg variable `vars`. This has the benefit that the OpenID plugin remains untouched and only the Wookie plugin had to be modified. Subsequently, the Wookie plugin hands over the OpenID as a parameter `openid_identifier` in the widget calling URL. Since the widget page is HTML, a JavaScript regular expression is used to parse the URL for the `openID` parameter so that it can be handed over to the widget which is taking care of the authentication process.

CHAPTER 6

A supporting Widget Template

6.1 Wookie Widget Development

Generally, widgets are designed to work without executing server-side code directly, but by calling web-services using AJAX. This approach has many advantages, however, software developers need to have some design rules in mind while creating widgets and services.

As previously mentioned, widgets consist of web standard technologies: HTML, JavaScript functions, CSS, and can embed images. Furthermore, a widget must follow a particular file structure:

- The widget manifest file (`config.xml`), which describes the widget.
- An HTML start page, typically `index.html`.
- One or more JavaScript files that implement the widget's functionality.
- One or more style sheets (CSS) that control the appearance of the widget.
- An icon for the widget.
- A thumbnail image for the widget.
- Additional media assets, such as images.

Widgets can also support internationalization by organizing these files into localized folders (ISO two-letter language code). Each localized folder can contain files that override the defaults when the widget is deployed in a particular location.

By using JavaScript as an interaction scripting language, Wookie offers built-in functions to handle preferences, shared data, event notifications, and calls to external web-services. Therefore, it is possible to:

- Store and retrieve users' preferences, or any other settings unique to the widget.
- Maintain data shared among all instances of a widget in a common context, for example, a chat log and buddy list for a chat widget.
- Send events between widgets in the same context, for example, if a user sets their location in one widget on their profile page, the widget can send a message to other widgets on the same page.
- Handle calls to external web-services and feeds through a proxy service. This is important, as otherwise a widget cannot communicate with other services as this poses a security risk.

For all of these services, a widget can make use of the shared `Widget` object that Wookie makes available via injected JavaScript code. This means that by uploading a widget to the Wookie engine it adds lines to the HTML files loading JavaScript libraries providing these functionalities.

For working with external web-services and APIs, a widget needs to implement an AJAX request and needs to handle the response, and to do so without breaching the security restrictions of the user's browser. Wookie handles the latter aspect by providing a built-in proxy service that allows the calling of services through the same Wookie server that is serving the widget, avoiding *Same Origin Policy Violation* errors (see [30]). Any servers a widget needs to call must be listed in Wookie's whitelist (entered through the Wookie administration interface). To invoke an external service from a widget, a proxy URL needs to be constructed by calling `Widget.proxify(url)` which returns a *proxified* version of a URL.

To simplify the development process, many JavaScript frameworks and libraries exist (like jQuery[1], Prototype[2], Dojo Toolkit[3], MooTools[4], Yahoo! UI Library[5] etc.). Therefore, handling AJAX calls is convenient and easy to maintain.

Before a widget can be deployed on a Wookie server, it needs a manifest file that describes the widget and provides some configuration details that the engine can use when it renders the widget in a container application. The manifest file must be called `config.xml`, must be located at the root of the widget's file structure, and must conform to the W3C Widget Packaging and XML Configuration specification (see [31]).

[1] `http://jquery.com`
[2] `http://prototypejs.org`
[3] `http://dojotoolkit.org`
[4] `http://mootools.net`
[5] `http://developer.yahoo.com/yui`

Different language versions of the manifest file can also be provided in localized folders, as described earlier. The key things to consider for the manifest file are:

- It must have a root element called `<widget>` with attributes for the height and width of the widget. This is important, as many containers will display the widget based on this information.
- It must have a `<name>` and preferably also a `<description>`.
- If an icon should be provided for a widget, an `<icon>` element with a `src` attribute must be set to the filename of the widget's icon.
- The `<content>` element with the `src` attribute should be set to the filename of the start file; this will default to `index.html`, but it does not hurt to make this explicit.
- If the widget uses external services, `<access network="true"/>` must be included.
- Optionally, an `<author>` element can be provided as well as a `<license>` element containing copyright information.

There are many other settings which can be stated in the widget manifest file—for more details it is referred to the previously mentioned W3C specification [31].

Widgets must be packaged as ZIP archives that contain all files that make up the widget. It must be ensured that the files are in the root folder of the archive and not nested inside a subfolder; otherwise, the Wookie server would throw an error. Once an archive is created, it can be uploaded to the Wookie server by using the management interface. When the widget has been uploaded, a service type must be allocated and any services required need to be added to the proxy whitelist.

If a widget needs event notifications, it is mandatory to include the following line in the manifest file: `<feature name="comet" required="true"/>`. Event notifications are useful when locking/unlocking widgets or dealing with shared data. If some users are not allowed to make any changes to a widget, it can be locked (`Widget.lock()`) and, of course, unlocked again (`Widget.unlock()`). The shared data API is Wookie's internal interface for storing and accessing data that can be shared among all instances of a widget that share a common context (e.g., for handling collaborative and social functionality).

In principle there are three methods with which shared data can be (1) set, (2) appended, and (3) retrieved:

1. `Widget.setSharedDataForKey(key, value)`

2. `Widget.appendSharedDataForKey(key, value)`

3. `Widget.sharedDataForKey(key, callback)`

Setting and retrieving data is done asynchronously. In the case of getting data, the method specified as callback is invoked after data fetching has been finished. Whenever a shared data value is set or appended, widgets receive an event notifying them of the changes. A widget can handle notifications by setting a function as the event handler, for example: `Widget.onSharedUpdate = handleSharedUpdate`. This sets the `handleSharedUpdate` method to be invoked whenever there is an event notifying the widget that a shared data value has been updated.

If the need arises for generating an instance-specific shared data key, method `Widget.instanceid_key` returns a unique identifier for the widget instance.

Authentication mechanisms are provided by using OpenID in conjunction with the Wookie plugin for Elgg. For authenticating, the functionality of Wookie needed to be extended because at this point, ticket-based access granting is not supported in version 0.8 of the Wookie engine (but will be in version 1.0). Further details on OpenID authentication are provided in Section 5.4[6].

6.2 Localization and Corporate Design in a Distributed Environment

The widgets created for this project should share the same appearance so that their origins are easily recognized. Therefore, a strategy for realizing a common *look and feel* across the developed widgets is proposed. When putting widgets into a container platform like Elgg, the user chooses whether to include one widget or several. The user will also put in some widgets that are not related to the project, but are useful in a learning context. Therefore, it is important that, for instance, a logo is displayed in the widget. This has two drawbacks: (1) the visual impression of the container can be overwhelmed by repeated elements and (2) the already quite limited size for the actual content is further reduced. Another important part of the widget is the title of it and an easy way to find help. Widgets developed for this thesis project are not self-explanatory so this is quite crucial for the user experience.

To help creating similar widgets and to provide useful functionalities, a template system was created (and is publicly available at [18]). The parts of the widget that are common to all widgets are stored in a template directory to separate this design part (along with common functions) from the rest of the individual code. A JavaScript library was created that generates the necessary parts of the user interface, so that the

[6]For additional information regarding the development of widgets, it is referred to the Wookie Widget Developer's Guide [43].

basic HTML of the original widget is not cluttered with design parts. Apart from design decisions, IWC methods are provided through the widget template, as well. These developments are reported in Section 6.4.

The template shown in Listing 6.1 already contains everything to be a widget—but it will only display the text *place your content here*. The widget developer needs to replace line 25 with custom HTML. The JavaScript code should be placed into the corresponding files, so that the HTML is not cluttered with programming logic.

```
1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http
    ://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
2 <html xmlns="http://www.w3.org/1999/xhtml">
3   <head>
4     <meta http-equiv="PRAGMA" content="NO-CACHE"/>
5     <meta http-equiv="Content-Type" content="text/html;charset=UTF-8"
        />
6     <title>YOUR_WIDGET_TITLE</title>
7     <script type="text/javascript" src="strings.js"></script>
8     <script type="text/javascript" src="ltfll-design/common.js"></
        script>
9     <script type="text/javascript" src="ltfll-design/style.js"></
        script>
10    <script type="text/javascript" src="lib/iwc.js"></script>
11    <script type="text/javascript" src="lib/non-wookie.js"></script>
12    <script type="text/javascript" src="lib/status.js"></script>
13    <script type="text/javascript" src="lib/jquery-1.4.3.min.js"></
        script>
14    <script type="text/javascript" src="lib/json2-min.js"></script>
15    <script type="text/javascript" src="scripts/config.js"></script>
16    <script type="text/javascript" src="scripts/iwc-callbacks.js"></
        script>
17    <script type="text/javascript" src="scripts/index.js"></script>
18    <link rel="stylesheet" href="ltfll-design/style.css" type="text/
        css"/>
19    <link rel="stylesheet" href="style/common.css" type="text/css"/>
20    <link rel="stylesheet" href="style/index.css" type="text/css"/>
21  </head>
22  <body onload="init()">
23    <div id="header"></div>
24    <div id="content">
25      place your content here
26    </div>
27    <div id="footer"></div>
28  </body>
29 </html>
```

Listing 6.1: Index file of the widget template.

Widgets can easily be localized by placing files in language-specific folders. The widget container will deliver a localized file depending on the browser's language

settings—the widget developer does not need to care about that. To make the application easily translatable, it is necessary to put all strings into a single resource file. Text displayed to the user must not be mixed with the widget code. String concatenation must not be used because the word order in different languages might be different. The best possible way is to use place holders for values to be filled in. Therefore, several JavaScript functions were created to extract the string resources from the resource files and to replace the placeholders with actual values.

The strings in the default language should be placed in the file `strings.js`, which is located in the root directory of the widget. After the template initialization, `strings.js` contains only the minimal information that is needed for the widget. In the example Listing 6.2, two strings are defined: an application title and the name of the help link to be displayed in the widget.

```
1 Strings = {
2   ApplicationTitle: "place your widget title here",
3   HelpLink: "Help"
4 };
```

Listing 6.2: Localization-based strings in a widget.

With this method it is possible to separate the generated HTML code from the variables to support localization. In the following example in Listing 6.3, the HTML code is placed in `index.js`—because the code does not contain text presented to a user (then it should be placed in the file `strings.js`). The `GetString()` function expects two parameters: the string that contains the placeholders and an object with properties for each placeholder. The returned string (in the variable `html`) would contain the replaced values.

```
1 var htmlTableRow = "
2   <tr>
3     <td class=word>${Word}</td>
4     <td class=count>${Count}</td>
5     <td><div class=bar style=\"width:${Width}px\"></div></td>
6   </tr>";
7 var html = Tools.GetString(
8   this.htmlTableRow,
9   {Word: word.word, Count: word.count, Width: width});
```

Listing 6.3: Example for using localization functions.

6.3 Basic Widget Structure

The widget structure available at [18] provides a template for both, a common design and a JavaScript library for IWC (see also Appendix D). To build interconnected widgets the guidelines described in this section must be followed.

Explanation of the directory structure:

- `/help/`
  Help pages (if there are any) should go here.
- `/images/`
  Images should be put here.
- `/legal/`
  Any legal texts as well as copyright statements should be put here.
- `/lib/`
  Any general JavaScript libraries are inserted here.
- `/ltfll-design/`
  Common project design specific CSS, JavaScript files, and images reside here.
- `/scripts/`
  Any widget-specific JavaScript functions are put here.
- `/style/`
  Widget specific CSS are located here.

In order to build a widget, some widget specific files must be configured:

- `/config.xml`
  This is the main file describing a widget (name, author(s), widget ID etc.). The polling feature line should be removed if IWC will not be used because otherwise useless traffic is generated.
- `/index.html`
  The HTML start page which needs to be customized (e.g., HTML widget title and the actual content which should be displayed in the widget).
- `/strings.js`
  The widget title should be put in here to be displayed as a headline in the widget and the name of the help link (if one should be provided)—see also Section 6.2.
- `/scripts/index.js`
  The generic logo of the widget in the upper right corner, the help link, and the footer (and, therefore, status messages) can be shown or hidden by setting the following three variables to be either true or false: `showLogo`, `showHelp`, `showFooter`.
- `/help/index.html`
  If the visibility of a help link is enabled, the help page(s) are created here.

6.4 Handling Inter Widget-Communication

Basically, for setting up IWC there are two data format options: JSON or key/value pairs. It is recommended to use JSON, but for some circumstances key/value pairs might be better suited. All IWC specific JSON methods end with *JSON* (e.g., `IWCsetVarJSON()`). It is not possible to mix up JSON and non-JSON methods because this will result in data loss and erroneous behavior.

To ease the usage of IWC, a JavaScript library has been developed which provides methods to handle JSON and non-JSON data sharing capabilities (see Appendix D). For setting up a widget to work with the IWC library, it has to be configured as follows:

- `/config.xml`
  If a widget wants to receive updates on shared data, the polling feature line has to be included. If the widget just wants to set shared data, but does not want to get notified about updates happening to data, then the the line needs to be removed in order to optimize the communication (otherwise it will generate useless traffic).

- `/scribts/index.js`
  By calling `IWCinit()` it has to be specified if *JSON* or *non-JSON* should be used as data format and if already existent shared data should be initialized at widget onload. Recommended setting: `IWCinit('JSON', true)`.

- `/scripts/config.js`
  It has to be decided, if IWC related status messages in the footer of the widget should be displayed. In accordance, `IWCstatusMessages` has to be set to be either true or false. Furthermore, dependent on the choice of the IWC data format the following has to be specified:

  - `JSON`
    The variable `IWCnameSpace` needs to be set to the namespace the correpsonding widget belongs to.
  - `non-JSON`
    The variable `IWCsharedData` lists all IWC variables the widget should be notified about updates.

- `/scripts/iwc-callbacks.js`
  The callback functions which are invoked when fetching IWC updates need to be named here. They need to be written exactly as the IWC variables the widget wants to listen to, but with *IWC* in the beginning (e.g., to get notified about updates for IWC variable `keyword`, `IWCkeyword()` needs to be inserted here).

Above it is described how IWC variable updates get retrieved and a widget gets notified. Moreover, there are two further functionalities: setting and deleting shared data. Again, it has to be distinguish between JSON and non-JSON data formats:

- `JSON`
  - `IWCsetVarJSON(ns, name, value)`
    Sets a new attribute with `name` and `value` in the defined namespace `ns`.
  - `IWCdelVarJSON(name)`
    Deletes the attribute named `name` (can only be done for the namespace the widget belongs to).
- `non-JSON`
  - `IWCsetVar(name, data)`
    Sets a new variable `name` with value `data`. As with the non-JSON data format, IWC variables do not belong to a namespace, therefore, arbitrary can be set.
  - `IWCdelVar(name)`
    Deletes variable `name`. As with the non-JSON data format, IWC variables do not belong to a namespace, thus, arbitrary can be deleted.

All fetched IWC variables are locally stored in the array `IWCdata[name]` and can be addressed accordingly. Furthermore, with the function `setStatus(status)` (and `setStatusFade(status)`, respectively) IWC specific status messages are set and displayed in the footer of a widget—but only if both `showFooter` in `/scripts/index.js` and `IWCstatusMessages` in `/scripts/config.js` are set to be true. If individual status messages need to be set, these functions should be used. But `showFooter` in `/scripts/index.js` must be set true, otherwise no footer and, therefore, no status messages will be displayed. For deleting status messages, the function `delStatus()` needs to be called. If the status message should automatically fade after a couple of seconds the method `setStatusFade(status)` provides this functionality.

If the widget is configured as described above, it will receive updates on IWC variables it listens to. If, by any means, a widget wants to retrieve an IWC variable which it does not listen to—because the polling feature may have been disabled—it can be achieved in the following way:

- `JSON`: All IWC variables for the namespace a widget belongs to are retrieved automatically (because it is stored as *one* JSON object) and they can be addressed using `IWCdata[name]`. If polling is disabled and the widget needs to retrieve

all IWC variables for a namespace, the method `IWCgetVarJSON()` must be called. This makes no sense when polling is enabled, because a widget with polling enabled listens on all updates happening in its namespace. Therefore, stored values in `IWCdata[name]` are up-to-date anytime.

- `non-JSON`: The method `IWCgetVar(name)` needs to be invoked in order to fetch an IWC variable a widget does not listen to. Once retrieved, the variable is also stored in `IWCdata[name]`. If an IWC callback function is defined for the variable, it will be invoked, as well.

Hence, with IWC, shared data can be set and updates can be retrieved. If there is an update for a variable a widget listens to, the corresponding IWC callback function is invoked doing something with the newly received data. For the JSON method, it is enough that the callback functions are named as described above. For the non-JSON method, all IWC variables a widget wants to listen to have to be explicitly defined (in `/scripts/config.js`, as described before). This is to minimize communication traffic for the non-JSON method because it is useless to receive updates for IWC variables a widget does not need. Every update on an IWC variable triggers a new request, therefore, it is recommended to only fetch the ones needed. By using JSON, all attributes set in a namespace are retrieved and the ones having a callback function are invoked. Network traffic is no problem here because a widget listens only to one namespace which is represented as one IWC variable consisting of a JSON object.

Furthermore, by utilizing the non-JSON method, variable names have to be unique. Therefore, it is recommended to use absolute namespaces, e.g., in the form *ns::variable*. All variables in the whole scope of IWC can be set, retrieved, and deleted. Therefore, extra caution is advisable.

With the JSON method, variables belong to a namespace and, therefore, can be uniquely identified. A widget belongs to exactly one namespace and listens on updates only on that namespace. This means, shared data can only be retrieved (and deleted) for the namespace the widget belongs to. The only exception of interfering with another namespace is the setting of shared data. If a widget of one namespace wants to communicate with another widget of a second namespace, it can set a new IWC variable in the scope of the second namespace. Therefore, the widget in the second namespace gets notified. If, for example, widget *A::1* (i.e., widget *1* in namespace *A*) wants to talk to widget *B::1*, widget *A::1* is allowed to set a new variable, e.g., *B::searchterm*. Hence, if widget *B::1* is listening on updates for this variable, it gets notified and can do something with the newly received data. This behavior acts as a data protection mechanism that one widget does not interfere with another, unless the two have negotiated how they want to do it.

Moreover, by using the template, a widget recognizes if it is served by a Wookie server or not. If the widget is deployed outside of the Wookie container or runs as a

stand-alone version, Wookie specific dummy objects are created and IWC methods are overloaded. Therefore, widgets can be used elsewhere without any code change. For example, a widget can be served by the OpenACS widget server described in Section 4.4. Wookie specific functionality is then disabled by default. But it is no problem to adapt the widget, for instance, to a potentially different API of another widget server.

CHAPTER 7

Use Case: A Lifelong Learner Scenario

7.1 Revisiting the Pedagogical Scenario

The pedagogical scenario targets a lifelong learner. Learners are free to choose if they want to use the PLE on their own or in a combination with a tutor. Either way the PLE is maintained by a learning provider. In this scenario, the learning provider is a company offering an internal continuing education service for its employees. In addition to the learner's view on the system described in Section 1.4, below the other views of participating entities are presented.

Tutor's View

When I start preparing my teaching course, I log into my PLE, by using the username and password that match my tutor's profile. I have to prepare a course on web development technologies for an internal training. First, I will use the suggested search methods to find out whether there are available materials on the subject of the course. If I do not find any available information with regard to such a course, my next step will be using the ontology-based search. The result is a visualization of the parts of the ontology including the sub-concepts related to the search terms. My next step is marking up the relevant sub-concepts and searching for corresponding materials. The system returns a list of learning objects. After browsing through the learning objects, I discover that the materials are sufficient for the course. I choose the most suitable ones for my purpose and store them on my harddrive for later usage. Furthermore, I compile a glossary and present a list of keywords for the topic for each learning object that has been selected for the course. Both are based on the concepts from the domain ontology. It is a semi-automatic process. I will make not only all learning objects, but also the glossary and keywords available to my students in order to help them find additional material [9].

IT Staff's View

It is my responsibility to ensure the proper functioning of the PLE system and its IWC functionality and to provide help for tutors and learners willing to use the new system. Besides the wizards I have prepared some tutorials that are profile dependent and, thus, aim at different user groups (tutors or students). Of course, I anticipate that in the future I will be addressed on different matters concerning the use of the system. Apart form this I have to make sure that the relevant software parts are accessible at any time and the system is stable [9].

System's View

After having been installed on a server, I have access to a number of resources, such as domain ontologies, learning objects, etc. I have modules that can manage these resources, can search in them, etc. When a learner/tutor uses me for the creation of a new course the two main activities in their work are: search for available materials and storage of learning objects. The search starts with a formulation of a query. This query has the form of a list of words similar to the query used in other full text search systems. The learner/tutor can either write the query directly in the search box or select appropriate concepts from the domain ontology which is loaded from the repository. Both functionalities are connected through the IWC API. After I receive the query, I process it in several steps. First, I recognize which concepts are mentioned in the query. Then, I consult the ontology for some related concepts in order to make the query complete. Then I search in the linked document repositories for the relevant resources. When I receive the list of appropriate resources, I order this list with respect to my judgment how well they match the original query. Then I show the list to the learner/tutor. She/He can browse over the list and open the corresponding resource. As the resource will be external to me (for example, a PDF document or a web page), I call an appropriate external program to open the resource for the learner/tutor. In this way I can present to the learner/tutor relevant learning objects for her/his query [9].

Company's View

With the every increasing speed of new technologies emerging, the requirements for our employees have changed. We have to provide more continuing education for each individual person. To optimize the required amount of time for the training, we use the new PLE system. With this system we can provide internal trainings at low costs [9].

7.2 Interconnected Widgets: the Use Case Exemplified

This section explains the developed system by example. Therefore, we make use of the pedagogical scenario and describe the functionalities of the implemented system by showing a selection of screenshots. In our use case, we employ Elgg (see Section 5.1) as widget container software. But as the widgets' functionalities are independent from the container, any other widget-supporting container software can be used to integrate our PLE use case[1].

As the PLE and its IWC functionalities build on web-technologies, it can be accessed via any web-browser. In our use case, Silvia has to log into the system by opening her preferred web-browser and navigating to the front website of the PLE system (see Figure 7.1). Two login possibilities exist: she can either enter her system-specific credentials or provide her OpenID (see Section 5.4) from a selection of OpenID service providers (Figure 7.1 shows an example OpenID). Internally, OpenID logins are mapped to system-specific credentials at the first successful login.

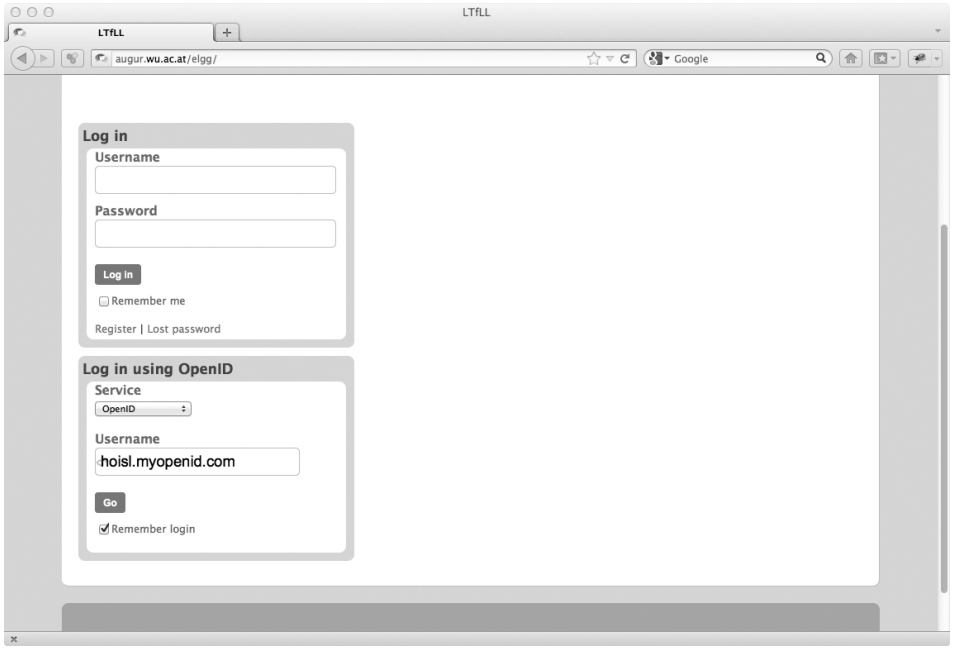

Figure 7.1: Login screen of the PLE.

To distinguish between different roles of a user, an own group exists for each role. Tutors can invite other users to join, for instance, the *tutors* group (invitation only). With the acceptance to join a group, the specific user inherits all privileges from the group

[1] It is also possible to use various widget containers interacting and communicating in one scenario.

settings, for example, to be able to add restricted widgets to one's own dashboard or to join closed discussions only visible to a certain group etc.

Silvia logs into the system for the very first time. After a successful login, she sees an empty dashboard (Figure 7.2) without any widgets on it. There exists also the possibility that a tutor can provide an initial set of widgets to a user. This is beneficial, to help the user getting accustomed to the—probably new—system or if certain widgets should always be displayed. But this is not the case for Silvia (see Figure 7.2) and, thus, she has to add widgets on her own depending on her individual learning goals. Therefore, Silvia clicks on *Edit page*, which will load the dashboard editing website displayed in Figure 7.3.

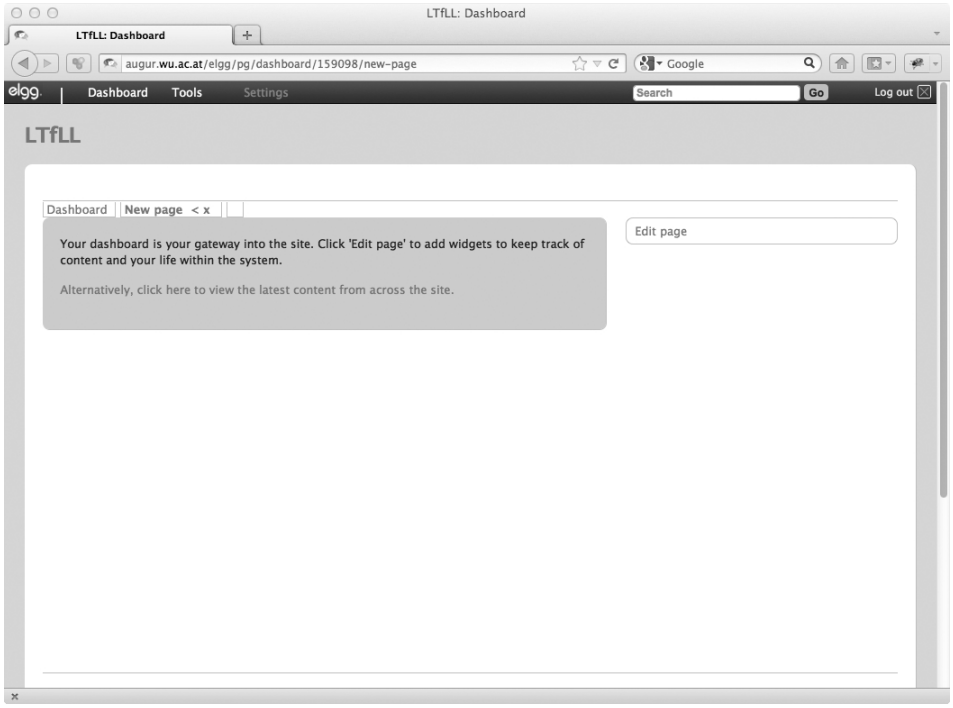

Figure 7.2: Empty dashboard at first login.

By creating a new dashboard, Silvia has to enter a name and an access-level for it (Figure 7.3). A dashboard can be visible only to oneself (private), to friends, to all logged in users, for the whole public, or for specific groups (e.g., *tutors*). In our use case, Elgg provides Silvia with a three column dashboard (configurable). To add widgets to the dashboard she has to drag and drop one or more *wookie* widgets to the corresponding left, middle, or right dashboard columns. Silvia is completely free in constructing her PLE and can add as many widgets, in any place, and in any order she would like her widgets to appear.

Figure 7.3: Adding individual widgets to one's own dashboard.

Figure 7.4 shows the combination of widgets Silvia has chosen for her PLE. The widget combination consists of:

- *Search widget*: A free text search widget, wherein Silvia can enter a number of relevant search terms.
- *Definition widget*: According to the term entered as query in the search widget, a snippet of a Wikipedia definition is given here. By clicking on the *Read more...* link Silvia is redirected to the corresponding Wikipedia article.
- *Graph visualization widget*: Besides the full text search widget, Silvia can navigate through an existing information-technology-based ontology to search for relevant as well as connected concepts.
- *Videos widget*: This widget searches for relevant videos found on the video sharing website YouTube. By clicking on a link, Silvia is redirected to the corresponding YouTube video.
- *Slides widget*: This widget displays a number of found presentations uploaded to the slide hosting service SlideShare. Again, by clicking on a displayed link, Silvia navigates to the corresponding SlideShare presentation.

- *Scientific papers widget*: The scientific papers widget displays a list of relevant research papers socially bookmarked on BibSonomy. The link takes Silvia to the corresponding BibSonomy entry.

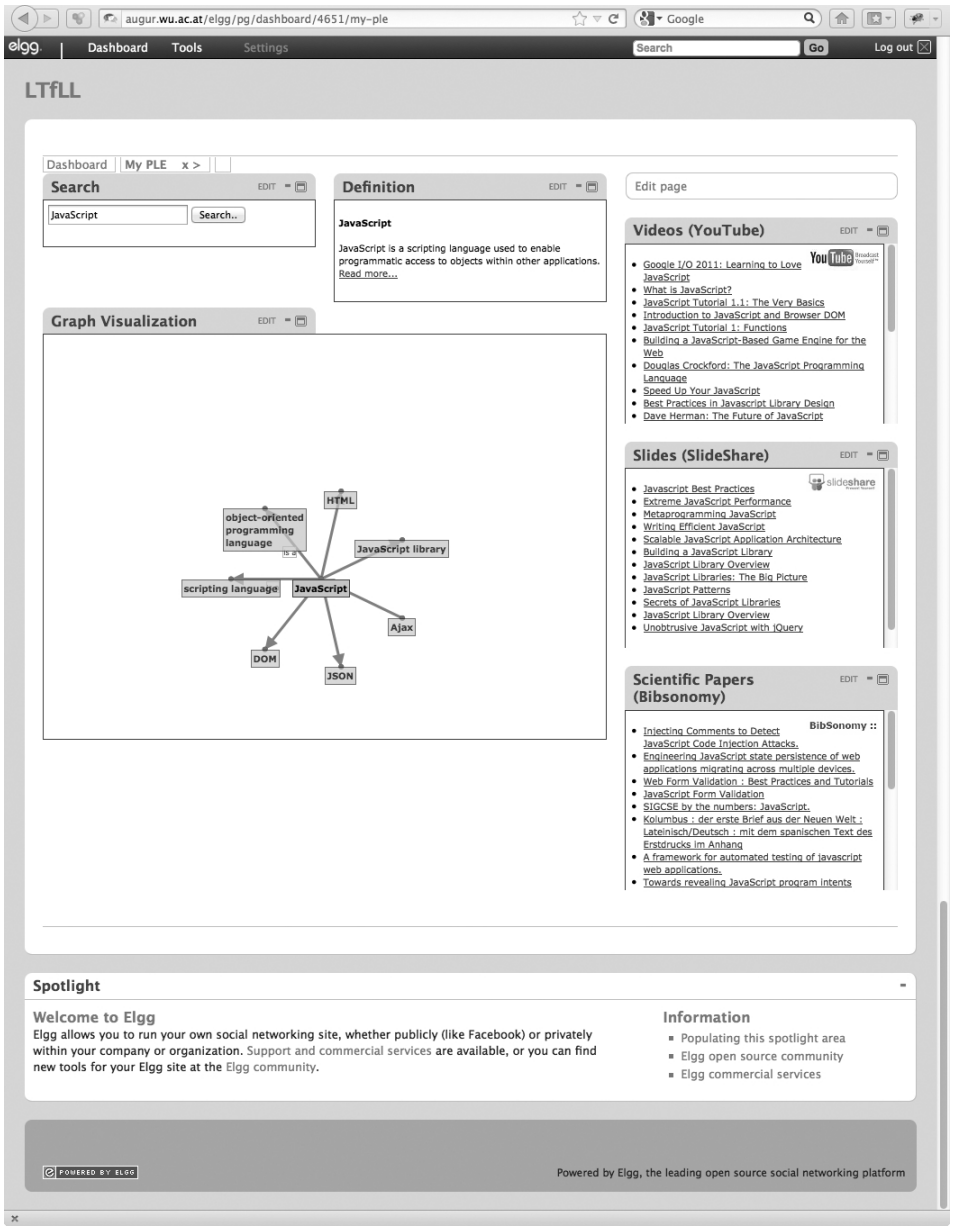

Figure 7.4: An example combination of widgets.

In our use case, Silvia—a lifelong learner—wants to refresh her knowledge of web-based development technologies, especially JavaScript. Therefore, she queries her PLE

for the term *JavaScript* by entering it as a term in the full text search widget. After submitting her query, the other widgets in her PLE get updated automatically (through the IWC API) and display the results. As output she gets a definition of the term together with interesting videos, presentations, and scientific research papers. Furthermore, in the graph visualization widget, she sees other relevant concepts retrieved from the ontology. If she clicks on a concept, the visualization is updated with the clicked concept centered, surrounded by further important and connected concepts. Simultaneously, the clicked concept is submitted as a new search query and all the other widgets in her PLE are automatically updated and display the results (because of the IWC). With her PLE, Silvia can easily find not only specific learning resources, but also other related and relevant learning materials.

The strength of a PLE is its high ability to be customized for each individual user. Therefore, the arrangement of widgets can be adjusted as well as each particular widget. Figure 7.5 shows the display possibilities implemented in the connector framework for Elgg (see Chapter 5). When a widget is placed on the dashboard, Silvia can adjust it according to her needs. If a widget should be replaced by another one, this can be done by selecting the appropriate widget from the drop-down list. Furthermore, access to a widget can be granted only to the user itself (private), to friends, to logged in users, to the general public, or to specific groups. These settings can be saved for each individual widget, hence, constructing one's own PLE.

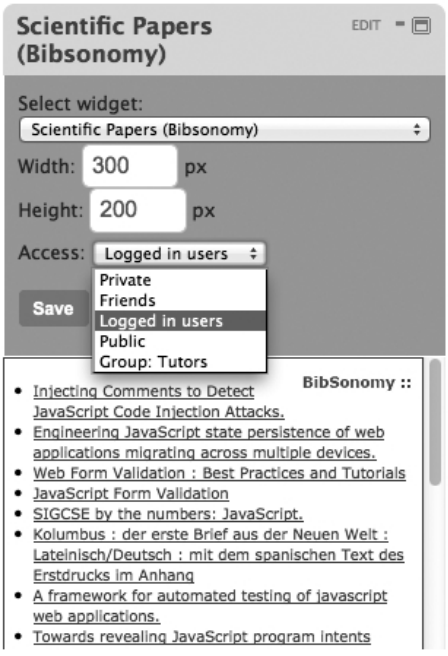

Figure 7.5: Configuring a widget.

Besides being able to rearrange and customize the displayed widgets on Silvia's dashboard, further options can be set. Figure 7.6 shows the functionality of minimizing widgets. Every widget has in its header a minus sign when it is expanded, and a plus sign when it is minimized. Clicking on a minus sign hides the widget until the user clicks on the plus sign again. Figure 7.6 shows two minimized widgets (videos and slides). With this option, Silvia can temporary hide uninteresting widgets to be displayed again at a later stage without losing any widget-specific configuration options (as shown, for instance, in Figure 7.5).

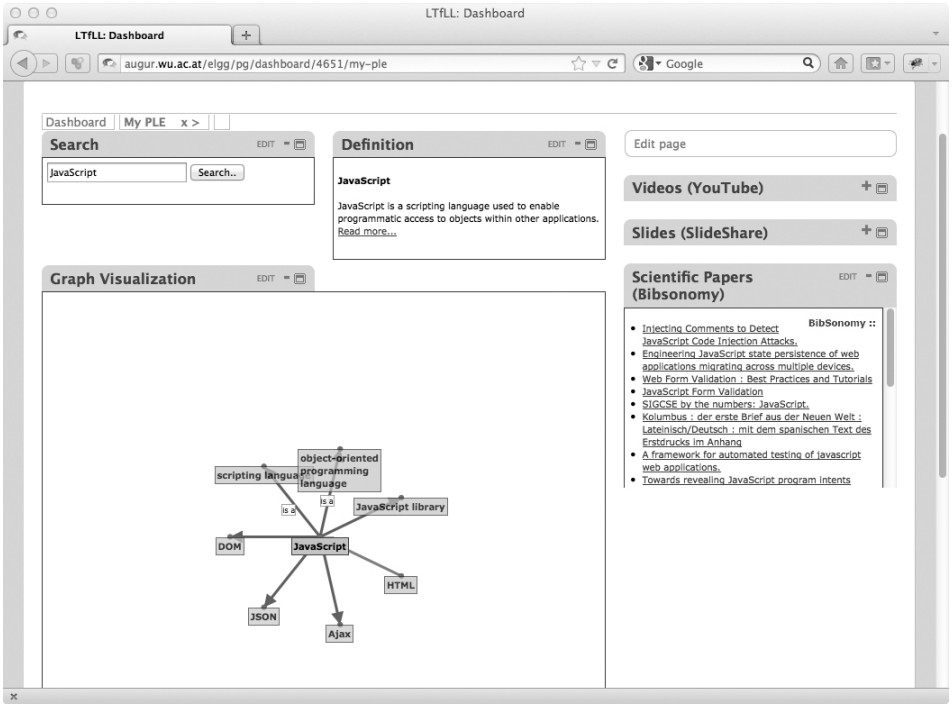

Figure 7.6: Minimizing widgets.

Another user-interface-specific customization option is the possibility to uncouple selected widgets from the dashboard (Figure 7.7). By clicking on the window icon displayed on the rightmost of every widget header, the specific widget gets uncoupled from Silvia's dashboard. This means, the widget is completely customizable in its size and position. It can be resized from tiny to fullscreen and moved to any place in the browser's window. There is no limitation on the number of decoupled widgets. Silvia can arrange decoupled widgets by herself. As can be seen from Figure 7.7, decoupled widgets are displayed in front of dashboard widgets and can overlay them. If a widget is uncoupled from the dashboard, the widget-instance on the dashboard gets grayed out and does not respond to user inputs (it is deactivated). Decoupled widgets act the same as dashboard widgets. If any user interaction is invoked in any widget, all other widgets

are updated—no matter if they are on the dashboard or decoupled from it (because of the IWC API). Therefore, dashboard and decoupled widgets are fully interoperable and can be used in combination without any limitations.

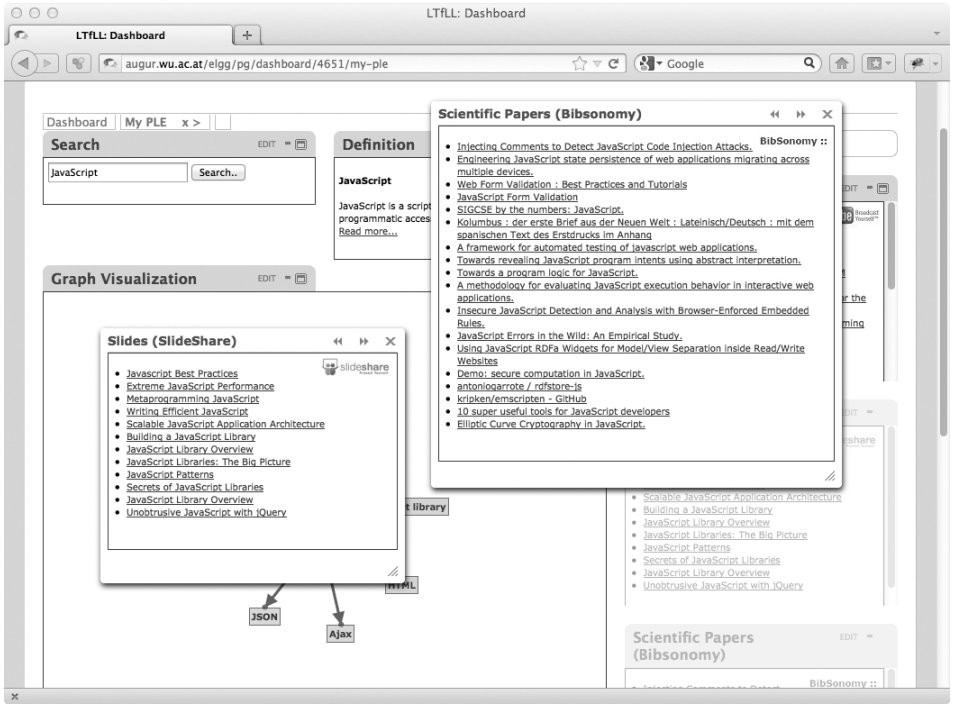

Figure 7.7: Decouple widgets from the dashboard.

In Elgg, Silvia can add as many dashboards as she likes, thus, constructing a variety of widget combinations in her PLE. As mentioned before, widgets can be rearranged freely. Figure 7.8 shows how to transfer a widget from the right to the middle column of Silvia's dashboard by using drag and drop. In order to add widgets, Silvia must drag a new *wookie* widget from the widget gallery and drop it into the corresponding dashboard column. There is no technical limitation on the number of widgets displayed. Silvia is free in constructing her PLE, but experience shows that four to seven widgets are in most cases sufficient for a user to accomplish a task and do not overstress a user's perception.

As discussed in the former paragraph, Silvia is free in the arrangement of widgets displayed on her dashboards. From the initial widget combination shown in Figure 7.4, she decides to require only a fraction of widgets (Figure 7.9). Therefore, she can either construct a new dashboard and add widgets to it or rearrange an existing one (as shown in Figure 7.8). Silvia decides that she only needs three widgets: the ontology graph visualization widget, the videos widget, and the scientific papers widget. She drags these three widgets onto her dashboard and customizes their appearance accordingly

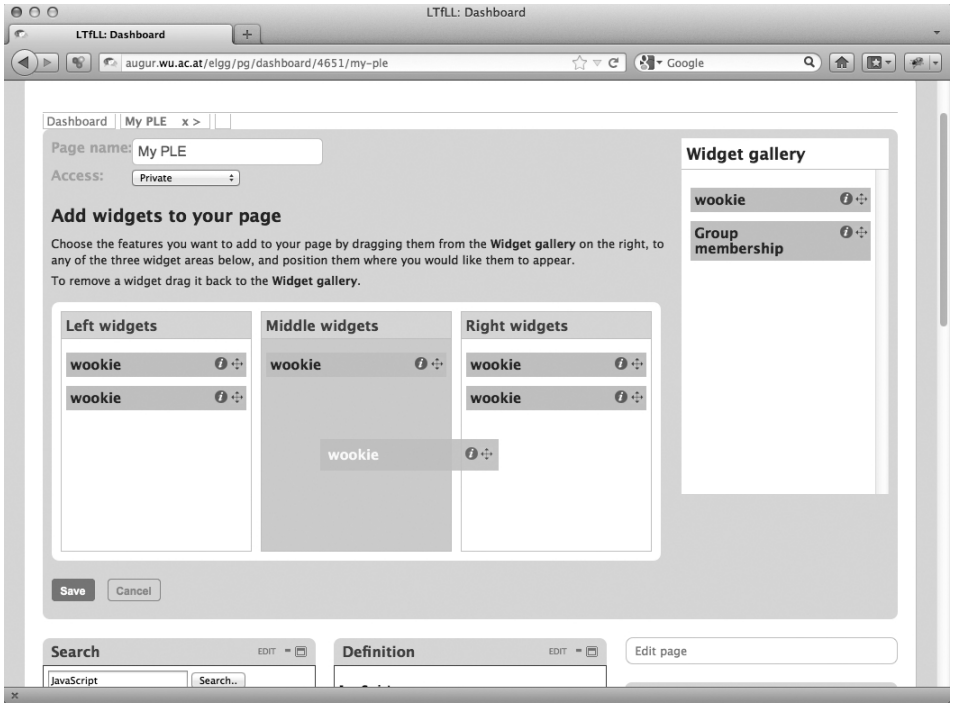

Figure 7.8: Rearrange and add widgets to the dashboard.

(Figure 7.9). Now she can navigate through the concepts with the graph visualization and gets relevant videos and research papers displayed on the left side of her dashboard. Of course, she can again modify the widgets or their arrangement at any time. Therefore, she is able to adjust her PLE to reflect her changing learning goals and IWC works for all possible usage scenarios.

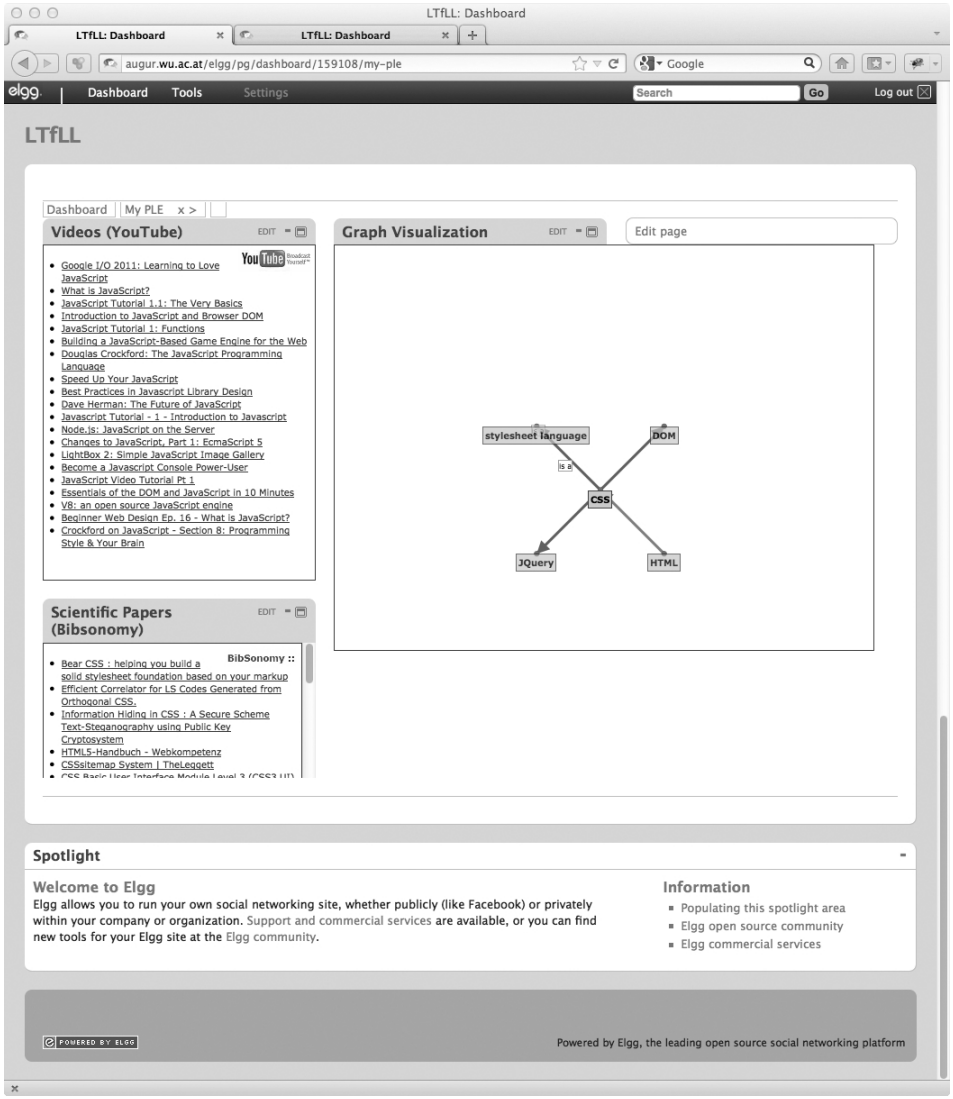

Figure 7.9: A new widget arrangement.

CHAPTER 8

Conclusion

8.1 Discussion

In this thesis, a concrete IWC prototype for a novel approach of developing MUPPLEs has been presented. A four-tier layered and loosely-coupled system was explained gaining power from its high flexibility: on the one hand, for learners and tutors by rearranging and recombining components and, on the other hand, for developers by reusing software artifacts. Further benefits consist of a high scalability of the proposed solution in conjunction with the possibility to independently deploy software components and being able to integrate them in a single or in multiple container applications.

Widgets are a novel way of designing MUPPLEs. With the increasing amount of widgets the need for communication among them arise. In this thesis, a new communication method was proposed which should overcome current restrictions in IWC techniques. The approach focus on user-tailored IWC. That means, communication is explicitly controlled through the widget engine and configured to meet users' needs. This is achieved by monitoring data flow of widgets to enable data sharing between widgets of one user or by widgets that belong to a group of users. The software artifacts were explicitly designed to work with the Wookie widget engine, although in principle the communication method could be implemented in any other widget server, as well. This was prototypically validated by also implementing a basic widget engine and connector framework in OpenACS. This work tries to bridge the gap between a highly individual MUPPLE approach and the possibilities for orchestrating services through IWC. It is believed that IWC will contribute to more personalized LEs, which are tailored to the functionalities users really require for their personal learning goals.

Mashing-up small encapsulated software artifacts means each of them must have self-contained functionalities. As this approach has many advantages over traditional architectures of LEs, it limits itself through the increasing difficulty of software arti-

facts involved. It is easier to specifically develop software runable exclusively on one platform than to claim that one solution will work for any environment. Therefore, software developers have to carefully think before developing MUPPLEs if the benefits of reusability and interoperability outperforms the lock-in effect on one environment.

As in this approach, widgets, services, and data storage facilities can be distributed across a heterogeneous set of web applications and servers, advantages emerge over traditional systems:

- Services can be hosted in one place or on multiple servers.
- The output can be displayed in widgets embedded in a frontend or in any standalone solution.
- Data can be stored in a central database or can be distributed.
- Simple data exchange between components can be realized through interfaces (e.g., IWC).
- Complex data exchange is encapsulated in services.

These benefits manifest in high reusability, interoperability, and transferability of the various software components developed within the approach presented in this thesis.

The distributed architectural decision taken ensures the scalability of the proposed approach. PLEs, web-services, databases, and computational algorithms can be distributed on separate physical machines allowing for balance loading and optimal response times. The customizable system proposed in this thesis can easily be used in a wide range of LEs. It only has to be assured that one service can take the output from another as its input—this is ensured by the definition of connection standards (e.g., an XML definition).

However, trade-offs of technological and architectural decisions are, for example, an overhead in communication messages and their size. This is acceptable because of the generated surplus in interoperability and reuse, and the preferred loosely-coupled design of the services. Calling services over networks also have the disadvantages of message delays (i.e., latency) and connection problems. These have to be kept in mind when orchestrating different services into a workflow. As the Internet of today is a reliable infrastructure this is not a major problem, but network error handling routines have to be considered. Therefore, web-service results have to be displayed asynchronously.

By using widgets, it is possible to develop software artifacts independent from the underlying platform, but the depth of widgets arrangement is limited to the functionality of the host LE. As the current widget container is insufficient in considering authorization mechanisms, the platform's own authentication mechanisms together with third-party services are used. The evidence collection of a learner is technically a mediation

service (created by linking different services together), with the drawback of a poorer level of performance in comparison to aggregated data.

Widgets are designed to provide use-case sized functionality within the boundaries of a restricted space on screen. In case of desktop computers, this typically means that the available screen space is restricted horizontally or vertically by a two or three column lay-out. A widget's user interface has to accommodate these restrictions. For some use-cases, this may not be sufficient and a fullscreen option must be provided (as by the Elgg connector framework) or the widget has to be deconstructed into several smaller ones.

Utilizing (usage) data from different sources carries with it the risk of running into privacy issues. Learners have to be assured that their private data are only used as intended and are not—in any form—accessible by third parties. Furthermore, a learner has to have as much control as possible over the amount of information visible to others. Therefore, privacy policies must be enforced explaining exactly what information the service will collect and how it might be used. It has also to be assured that, by externalizing information to the public domain, no inference can be drawn to the individual (unless a learner is intending otherwise). Authentication is ensured by password-protected logins, while challenge-based access control handles authorization issues. Finally, learners need to have the option to delete their private data at any time and unsubscribe from the community.

8.2 Requirements for Adoption

Despite the advantages described in former sections some open questions remain. One of it being the issue of authenticating web-services. An initial OpenID based authentication system was developed (see Section 5.4). After having successfully granted access to Elgg, the OpenID identifier is passed to the widget which itself must take care of the authentication process. This solution has the benefit of being relatively simple to implement, but the disadvantage that every widget has to integrate OpenID functionalities on its own. Hence, a token based authorization approach implemented centrally in Wookie and available to all widgets should be investigated. Therefore, a time limited hash value has to be generated at logging-in to the container application. Web-services will then only be accessed by calling them with the specific token and checking against the Wookie REST API for validity. Access levels should be modeled for authorization purposes. In the case of Elgg, for example, being member of a certain group will be equivalent of having a certain access level, thus, being able to access certain resources.

Stating to have a solution for heterogeneous systems communicating with each other has the challenge of introducing interfaces. Service A needs to know how to call service B and what to expect from it and vice versa. Therefore, APIs have to be well-defined and documentation has to be written and consistently updated.

Another challenge is that the proposed approach is optimized for reusability. This means plugging-in any widget in any LE should work out-of-the-box. It is kind of obvious that in practice such a perfect solution is hard to achieve. Although any widget will work with any widget engine following the W3C recommendations and with any widget container having a plug-in for a widget engine. A program's application logic should not need specific adaptations (or only minor once) but the IWC strategy and the look and feel of the GUI will need adjustments to match the style of the integrating container application. Efforts can significantly be minimized by using style definitions.

Currently, Wookie is in a development-state. A first official release will be made once it has been assured that the developments presented here are integrated in the release. An official release will significantly improve impact and uptake.

An enjoyable user experience is of utmost importance. An intuitive usage of the services explained in this thesis is essential. In general, embedded instructional or self-instructional support mechanisms must be integrated. Guidelines should be provided to answer frequently asked questions, to avoid mistakes, and to support users in learning the use of the software while they are experiencing it.

8.3 Future work

Future planned developments for the end-users (learners/tutors) in terms of the connector framework are an alternative view of choosing widgets not only by a drop-down list but also by a widget gallery displaying the widgets' names as well as an icon and a short description (the Wookie interface already exists). As the view displayed in Figure 5.1 is a good way for the role of a tutor to select widgets, a learner most probably will not change a widget in a certain context. A tutor has to decide for which exercise which widget is appropriate and the learner may change only the appearance of the specific widget.

Further developments regarding the connector framework's internal logic and its technical aspects will also be done by implementing the full set of functionalities the Wookie engine provides via its RESTful interface. Another extension would be integrating widgets from different Wookie engines. This would mean restructuring the internal plugin logic to handle URLs of different Wookie engines on a per widget basis.

Workflows between widgets could be improved by allowing the possibility of modeling dependencies. Also different event types could be introduced in order to allow for "unplanned" communication. Tracking of user, system, and message flows could also be envisaged. In order to remain W3C widget standard compliant, dependencies need to be modeled as features. For error handling and conflict resolution, this will require parse extensions and further modifications, both, in the connector framework and in the runtime environment.

Moreover, development work will also head in the direction of optimizing the IWC

methods between the Wookie server and the client widgets and to ease service orchestration by providing a workflow modeling library. As an example, the current implementation allows only to set the IWC policy for the whole Wookie engine. Currently, it is worked on developing a per widget-based communication policy so that the user can explicitly specify which information should be shared with others. Further improvements will target the notification system, as well, so that users, for example, are notified if an administrator switches from one shared data policy to another.

APPENDIX A

IWC Patch for Wookie

Listing A.1 shows the IWC patch for Wookie (available at [47]) against revision 1.023.765 to provide methods for coupling widgets.

```
Index: widgetserver.properties
===================================================================
--- widgetserver.properties (revision 1023765)
+++ widgetserver.properties (working copy)
@@ -52,4 +52,7 @@
 widget.persistence.manager.user=@REPOSITORY_USER@
 widget.persistence.manager.password=@REPOSITORY_PASSWORD@
 widget.persistence.manager.rootpath=@REPOSITORY_ROOTPATH@
-widget.persistence.manager.workspace=@REPOSITORY_WORKSPACE@
\ No newline at end of file
+widget.persistence.manager.workspace=@REPOSITORY_WORKSPACE@
+#####################################################################
+# Coupling widgets (Inter-Widget Communication)
+#widget.iwc.coupling = apiKey, sharedDataKey
Index: org/apache/wookie/ajaxmodel/impl/WidgetAPIImpl.java
===================================================================
--- org/apache/wookie/ajaxmodel/impl/WidgetAPIImpl.java (revision 1023765)
+++ org/apache/wookie/ajaxmodel/impl/WidgetAPIImpl.java (working copy)
@@ -21,6 +21,7 @@

 import javax.servlet.http.HttpServletRequest;

+import org.apache.commons.configuration.Configuration;
 import org.apache.log4j.Logger;
 import org.apache.wookie.Messages;
 import org.apache.wookie.ajaxmodel.IWidgetAPI;
@@ -164,13 +165,30 @@
    public String setSharedDataForKey(String id_key, String key, String value)
        {
      HttpServletRequest request = WebContextFactory.get().
         getHttpServletRequest();
     Messages localizedMessages = LocaleHandler.localizeMessages(request);
-        IPersistenceManager persistenceManager = PersistenceManagerFactory.
    getPersistenceManager();
```

```
32 -         IWidgetInstance widgetInstance = persistenceManager.
     findWidgetInstanceByIdKey(id_key);
33 +
34 +     IPersistenceManager persistenceManager = PersistenceManagerFactory.
     getPersistenceManager();
35 +     IWidgetInstance widgetInstance = persistenceManager.
     findWidgetInstanceByIdKey(id_key);
36 +
37       if(widgetInstance == null) return localizedMessages.getString("
             WidgetAPIImpl.0");
38       if(widgetInstance.isLocked()) return localizedMessages.getString("
             WidgetAPIImpl.2");
39 -     //
40 -     PropertiesController.updateSharedDataEntry(widgetInstance, key, value,
     false);
41 -     Notifier.notifySiblings(widgetInstance);
42 +
43 +     Map<String, Object> couplingWidgetAttributes =
     getCouplingWidgetAttributes(widgetInstance, request);
44 +
45 +     if (!couplingWidgetAttributes.isEmpty()) {
46 +       IWidgetInstance[] couplingWidgetInstances = persistenceManager.
     findByValues(IWidgetInstance.class, couplingWidgetAttributes);
47 +
48 +       for (IWidgetInstance instance : couplingWidgetInstances) {
49 +         if(instance == null) return localizedMessages.getString("
     WidgetAPIImpl.0");
50 +         if(instance.isLocked()) return localizedMessages.getString("
     WidgetAPIImpl.2");
51 +
52 +         PropertiesController.updateSharedDataEntry(instance, key, value,
     false);
53 +         Notifier.notifySiblings(instance);
54 +       }
55 +     } else {
56 +       PropertiesController.updateSharedDataEntry(widgetInstance, key, value,
      false);
57 +       Notifier.notifySiblings(widgetInstance);
58 +     }
59 +
60       return "okay"; //$NON-NLS-1$
61     }
62
63 @@ -181,16 +199,58 @@
64     public String appendSharedDataForKey(String id_key, String key, String
         value) {
65       HttpServletRequest request = WebContextFactory.get().
             getHttpServletRequest();
66       Messages localizedMessages = LocaleHandler.localizeMessages(request);
67 -         IPersistenceManager persistenceManager = PersistenceManagerFactory.
     getPersistenceManager();
68 -         IWidgetInstance widgetInstance = persistenceManager.
     findWidgetInstanceByIdKey(id_key);
69 +
70 +     IPersistenceManager persistenceManager = PersistenceManagerFactory.
     getPersistenceManager();
71 +     IWidgetInstance widgetInstance = persistenceManager.
     findWidgetInstanceByIdKey(id_key);
```

```
72 +
73       if(widgetInstance == null) return localizedMessages.getString("
             WidgetAPIImpl.0");
74       if(widgetInstance.isLocked()) return localizedMessages.getString("
             WidgetAPIImpl.2");
75 -     //
76 -     PropertiesController.updateSharedDataEntry(widgetInstance, key, value,
     true);
77 -     Notifier.notifySiblings(widgetInstance);
78 +
79 +     Map<String, Object> couplingWidgetAttributes =
     getCouplingWidgetAttributes(widgetInstance, request);
80 +
81 +     if (!couplingWidgetAttributes.isEmpty()) {
82 +       IWidgetInstance[] couplingWidgetInstances = persistenceManager.
     findByValues(IWidgetInstance.class, couplingWidgetAttributes);
83 +
84 +       for (IWidgetInstance instance : couplingWidgetInstances) {
85 +         if(instance == null) return localizedMessages.getString("
     WidgetAPIImpl.0");
86 +         if(instance.isLocked()) return localizedMessages.getString("
     WidgetAPIImpl.2");
87 +
88 +         PropertiesController.updateSharedDataEntry(instance, key, value,
     true);
89 +         Notifier.notifySiblings(instance);
90 +       }
91 +     } else {
92 +       PropertiesController.updateSharedDataEntry(widgetInstance, key, value,
      true);
93 +       Notifier.notifySiblings(widgetInstance);
94 +     }
95 +
96       return "okay"; //$NON-NLS-1$
97     }
98
99 +         /*
100 +          * (non-Javadoc)
101 +          * @see org.apache.wookie.ajaxmodel.IWidgetAPI#
     getCouplingWidgetAttributes(java.lang.Object, java.lang.Object)
102 +          */
103 +   public Map<String, Object> getCouplingWidgetAttributes(IWidgetInstance
     widgetInstance, HttpServletRequest request) {
104 +     Configuration properties = (Configuration) request.getSession().
     getServletContext().getAttribute("properties");
105 +     String[] couplingAttributes = properties.getStringArray("widget.iwc.
     coupling");
106 +
107 +     Map<String, Object> couplingWidgetAttributes = new HashMap<String,
     Object>();
108 +     for (String attribute : couplingAttributes) {
109 +       if (attribute.equals("sharedDataKey")) {
110 +         couplingWidgetAttributes.put(attribute, widgetInstance.
     getSharedDataKey());
111 +       } else if (attribute.equals("apiKey")) {
112 +         couplingWidgetAttributes.put(attribute, widgetInstance.getApiKey());
113 +       } else if (attribute.equals("idKey")) {
114 +         couplingWidgetAttributes.put(attribute, widgetInstance.getIdKey());
```

```
115 +      } else if (attribute.equals("userId")) {
116 +        couplingWidgetAttributes.put(attribute, widgetInstance.getUserId());
117 +      }
118 +      // add more criteria if needed
119 +    }
120 +
121 +    return couplingWidgetAttributes;
122 +  }
123 +
124    /*
125     * (non-Javadoc)
126     * @see org.apache.wookie.ajaxmodel.IWidgetAPI#lock(java.lang.String)
127 @@ -306,4 +366,4 @@
128          return ""; //$NON-NLS-1$
129    }
130
131 -}
132 \ No newline at end of file
133 +}
134 Index: org/apache/wookie/ajaxmodel/IWidgetAPI.java
135 ===================================================================
136 --- org/apache/wookie/ajaxmodel/IWidgetAPI.java  (revision 1023765)
137 +++ org/apache/wookie/ajaxmodel/IWidgetAPI.java  (working copy)
138 @@ -17,9 +17,11 @@
139  import java.util.Map;
140  import java.util.List;
141
142 +import javax.servlet.http.HttpServletRequest;
143 +
144  import org.apache.wookie.beans.IPreference;
145 +import org.apache.wookie.beans.IWidgetInstance;
146
147 -
148  /**
149   * Definition of the widget API.
150   * @author Paul Sharples
151 @@ -161,5 +163,15 @@
152     * @return
153     */
154    public String userPropertyForKey(String id_key, String key);
155 +
156 +  /**
157 +   * Returns a map containing widget coupling attributes which
158 +   * can be used to find matching widget instances. Set of
159 +   * attributes must be specified in widgetserver.properties.
160 +   * @param widgetInstance - coupling widgets are searched based on this
     widget instance
161 +   * @param request - HTTP request object
162 +   * @return - a map containing attributes and their values
163 +   */
164 +  public Map<String, Object> getCouplingWidgetAttributes(IWidgetInstance
     widgetInstance, HttpServletRequest request);
165
166  }
```

Listing A.1: IWC patch for Wookie.

APPENDIX B

Installing the Wookie Plug-in for Elgg

These are the installation instructions for the Wookie connector framework for Elgg (available at [14]). For the plugin to work, it is assumed that Elgg and the Wookie engine are configured and running. Then, follow these steps:

- Download the plugin and place it in the folder `[elggDir]/mod/`.
- Enable the plugin via the tool administration panel in Elgg.
- In the tool administration panel, configure the plugin by setting the URL to the Wookie engine (e.g., `http://augur.wu.ac.at:8080/wookie/`) and insert the API key obtained from the Wookie engine (each individual web application needs its own API key which can be generated from Wookie's administration interface).
- Now you should be able to add Wookie widgets to your dashboard. This can be done by choosing the features you want to add to your page by dragging them from the widget gallery to any of the three widget areas of your dashboard and position them where you would like them to appear.

APPENDIX C

Core Widget Functionalities of the Wookie Plug-in for Elgg

Listing C.1 shows the core widget-related methods of the Wookie connector framework for Elgg (available at [14]).

```
1  <?php
2
3  require_once('wookie_xml.php');
4
5  class elggWookieWidget {
6
7    ## widget instance
8    private $widget;
9
10   ## servicetype
11   private $type;
12
13   ## elgg user variables
14   private $userID = '';
15   private $username = '';
16   private $name = '';
17   private $src;
18
19   ## global vars
20   private $vars;
21
22
23   function __construct($vars) {
24     $this->vars = $vars;
25
26     ## url to wookie engine (as set in tool administration)
27     $this->vars['entity']->wookie_url = get_plugin_setting('wookie_url', '
           wookie');
28     if ($this->vars['entity']->wookie_url && strrchr($this->vars['entity']->
           wookie_url, '/') != '/') {
29       $this->vars['entity']->wookie_url .= '/';
```

```
30      }
31
32      ## the requested api key (as set in tool administration)
33      $this->vars['entity']->wookie_api_key = get_plugin_setting('
            wookie_api_key', 'wookie');
34
35      $user = $_SESSION['user'];
36      if (!is_null($user)){
37        $this->userID = $user->getGUID();
38        if (!is_null($this->userID)){
39          $this->username = get_user($this->userID)->username;
40          $this->name = get_user($this->userID)->name;
41          $this->src = get_user($this->userID)->getIcon();
42        }
43      }
44
45      ## instantiate widget
46      $this->widget = $this->getWidget();
47
48      ## add openid identifier to widget url
49      if ($vars['user']->alias && $this->widget['widgetdata']['url']) {
50        $this->widget['widgetdata']['url'] .= '&openid_identifier=' . $vars['
              user']->alias;
51      }
52
53      $this->addParticipant();
54
55      $this->setPersonalProperty('username', urlencode($this->username));
56      $this->setPersonalProperty('name', urlencode($this->name));
57
58      ## need to do access check here
59      if (page_owner() == $this->userID){
60        $this->setPersonalProperty('moderator', 'true');
61      }
62    }
63
64
65    ## parameters: url, title, height, width, maximize
66    public function getWidgetParameter($param) {
67      $vars_param = "wookie_widget_$param";
68
69      if ($this->vars['entity']->$vars_param) {
70        return $this->vars['entity']->$vars_param;
71      }
72
73      return $this->widget['widgetdata'][$param];
74    }
75
76
77    public function getWidget() {
78      $request = $this->vars['entity']->wookie_url;
79      $request.= 'WidgetServiceServlet?';
80      $request.= 'requestid=getwidget';
81      $request.= '&api_key='.$this->vars['entity']->wookie_api_key;
82      $request.= '&servicetype='.$this->type;
83      $request.= '&widgetid='.$this->vars['entity']->wookie_widget_guid;
84      $request.= '&userid='.$this->userID;
85      $request.= '&shareddatakey='.$this->userID;
```

```
86
87      $response = file_get_contents($request);
88
89      return XML_unserialize($response);
90    }
91
92
93    public function setPersonalProperty($name, $value) {
94      $request = $this->vars['entity']->wookie_url;
95      $request.= 'WidgetServiceServlet?';
96      $request.= 'requestid=setpersonalproperty';
97      $request.= '&api_key='.$this->vars['entity']->wookie_api_key;
98      $request.= '&servicetype='.$this->type;
99      $request.= '&widgetid='.$this->vars['entity']->wookie_widget_guid;
100     $request.= '&userid='.$this->userID;
101     $request.= '&shareddatakey='.$this->userID;
102     $request.= '&propertyname='.$name;
103     $request.= '&propertyvalue='.$value;
104
105     return file_get_contents($request);
106   }
107
108
109   public function addParticipant() {
110     $request = $this->vars['entity']->wookie_url;
111     $request.= 'WidgetServiceServlet?';
112     $request.= 'requestid=addparticipant';
113     $request.= '&api_key='.$this->vars['entity']->wookie_api_key;
114     $request.= '&servicetype='.$this->type;
115     $request.= '&widgetid='.$this->vars['entity']->wookie_widget_guid;
116     $request.= '&userid='.$this->userID;
117     $request.= '&shareddatakey='.$this->userID;
118     $request.= '&participant_id='.$this->userID;
119     $request.= '&participant_display_name='.$this->username;
120     $request.= '&participant_thumbnail_url='.$this->src;
121
122     return file_get_contents($request);
123   }
124
125
126   ## adapted from the moodle plugin
127   function showGallery() {
128     $widgets = $this->getWidgets();
129
130     $gallery = 'Select widget: <select name="params[wookie_widget_guid]">';
131     $gallery .= '<option value="">[No widget selected]</option>';
132
133     foreach ($widgets as $widget){
134       unset($selected);
135       if ($this->vars['entity']->wookie_widget_guid == $widget->id) {
136         $selected = " selected";
137       }
138       $gallery .= "<option value='$widget->id'" . $selected . ">$widget->
              title</option>";
139     }
140
141     $gallery .= '</select><br />';
142
```

```
143     return $gallery;
144   }
145
146
147   function getWidgets() {
148     $request = $this->vars['entity']->wookie_url;
149     $request.= 'advertise?all=true';
150     $response = file_get_contents($request);
151     $xml = XML_unserialize($response);
152     $widgets = array();
153
154     foreach ($xml['widgets']['widget'] as $data){
155       ## if there's an identifier, coin a new widget
156       if ($data['identifier'] != ''){
157         $widget = new Widget();
158         $widget->id = $data['identifier'];
159       } else {
160         $widget->title = $data['title'];
161         $widget->description = $data['description'];
162         $widget->icon =  $data['icon'];
163         $widget->category =  $data['category'];
164         $widget->author =  $data['author'];
165         $widgets[] = $widget;
166       }
167     }
168
169     sort($widgets);
170
171     return $widgets;
172   }
173 }
174
175
176 class Widget {
177   public $title;
178   public $id;
179   public $description;
180   public $icon;
181   public $category;
182   public $author;
183 }
184
185 ?>
```

Listing C.1: Core widget methods of the connector framework.

APPENDIX D

IWC JavaScript Library

Listing D.1 shows the main methods of the IWC JavaScript library provided in the widget template available at [18].

```
1  IWCsharedKey = null;
2  IWCdata = new Array();
3
4
5  //########################################
6  //IWC init method
7  //########################################
8
9  function IWCinit(format, initSharedData) {
10   Widget.preferenceForKey("sharedDataKey",
11     function (key) {
12       //setting shared data key
13       IWCsharedKey = key;
14
15       //initial retrieval of already set variables
16       if (initSharedData) {
17         if (format == "JSON") {
18           IWChandleSharedUpdateJSON(key);
19         } else {
20           IWChandleSharedUpdate(key);
21         }
22       }
23     }
24   );
25
26   //bind corresponding method to shared data updates (dependent on data
         format used)
27   if (format == "JSON") {
28     Widget.onSharedUpdate = IWChandleSharedUpdateJSON;
29   } else {
30     Widget.onSharedUpdate = IWChandleSharedUpdate;
31   }
32
33   if (IWCstatusMessages) setStatusFade('IWC configured ...');
```

```
34 }
35
36
37 //########################################
38 //JSON specific methods
39 //########################################
40
41 function IWCsetVarJSON(ns, name, value) {
42   if (IWCstatusMessages) setStatusFade('sending data ...');
43   Widget.sharedDataForKey(
44     ns,
45     function (data) {
46       //if shared data exist for this namespace
47       if (data != "No matching key found" && data != "") {
48         obj = JSON.parse(data);
49         //if the new value is different from the already set one
50         if (obj[name] != value) {
51           //only store shared data locally if it belongs to the same
                 namespace as this widget
52           if (ns == IWCnameSpace) {
53             IWCdata[name] = value;
54           }
55           obj[name] = value;
56           data = JSON.stringify(obj);
57           Widget.setSharedDataForKey(ns, data);
58         }
59       } else {
60         //only store shared data locally if it belongs to the same namespace
               as this widget
61         if (ns == IWCnameSpace) {
62           IWCdata[name] = value;
63         }
64         //if no shared data exist for this namespace, begin a new entry
65         data = '{"' + name + '":"' + value + '"}';
66         Widget.setSharedDataForKey(ns, data);
67       }
68     }
69   );
70 }
71
72 function IWCdelVarJSON(name) {
73   if (IWCstatusMessages) setStatusFade('deleting data ...');
74   Widget.sharedDataForKey(
75     IWCnameSpace,
76     function(data) {
77       //only delete shared data which is already set
78       if (data != "No matching key found" && data != "") {
79         delete IWCdata[name];
80         obj = JSON.parse(data);
81         delete obj[name];
82         data = JSON.stringify(obj);
83         Widget.setSharedDataForKey(IWCnameSpace, data);
84       }
85     }
86   );
87 }
88
89 function IWCgetVarJSON() {
```

```
90    Widget.sharedDataForKey(
91      IWCnameSpace,
92      function(data) {
93        //only do something when actual data was received
94        if (data != "No matching key found" && data != "") {
95          obj = JSON.parse(data);
96          //loop over all attributes of retrieved object
97          for (var name in obj) {
98            //only do something if new retrieved value is different from
                  already set one
99            if (IWCdata[name] != obj[name]) {
100             if (IWCstatusMessages) setStatusFade('retrieving data ...');
101             //store the newly retrieved value (for next time comparison)
102             IWCdata[name] = obj[name];
103             //callIWC callback function if existent
104             if (eval("typeof " + "IWC" + name) == "function") {
105               //call correpsonding IWC function and pass value
106               eval("IWC" + name + "('" + obj[name] + "')");
107               break;
108             }
109           }
110         }
111       }
112     }
113   );
114 }
115
116 function IWChandleSharedUpdateJSON(key) {
117   //check if api key of instantiated widget is the same as the api key
          retrieved on a shared update
118   if (key == IWCsharedKey) {
119     IWCgetVarJSON();
120   }
121 }
122
123
124 //########################################
125 //methods for key:val pairs (non-JSON option)
126 //########################################
127
128 function IWCsetVar(name, data) {
129   if (IWCstatusMessages) setStatusFade('sending data ...');
130   IWCdata[name] = data;
131   Widget.setSharedDataForKey(name, escape(data));
132 }
133
134 function IWCdelVar(name) {
135   if (IWCstatusMessages) setStatusFade('deleting data ...');
136   delete IWCdata[name];
137   //setting a variable null or "null" deletes it
138   Widget.setSharedDataForKey(name, null);
139 }
140
141 function IWCgetVar(name) {
142   Widget.sharedDataForKey(
143     name,
144     function(data) {
145       data = unescape(data);
```

```
146       //if retrieved data is different from already set one and
147       //actual data was received
148       if (IWCdata[name] != data && data != "No matching key found") {
149         if (IWCstatusMessages) setStatusFade('retrieving data ...');
150         //store the data for next time comparison
151         IWCdata[name] = data;
152         //if corrsponding IWC callback function exists
153         if (eval("typeof " + "IWC" + name) == "function") {
154           //call IWC callback function and pass data
155           eval("IWC" + name + "('" + data + "')");
156         }
157       }
158     }
159   );
160 }
161
162 function IWChandleSharedUpdate(key) {
163   if (key == IWCsharedKey) {
164     //to minimize traffic we define for every widget the shared data updates
          it should listen to
165     //here: loop over the array and call data fetching method
166     for (var i = 0; i < IWCsharedData.length; i++) {
167       IWCgetVar(IWCsharedData[i]);
168     }
169   }
170 }
```

Listing D.1: IWC JavaScript library.

Bibliography

[1] Evgeny Bogdanov, Christophe Salzmann, Sandy El Helou, and Denis Gillet. Social Software Modeling and Mashup based on Actors, Activities and Assets. In Fridolin Wild, Marco Kalz, and Matthias Palmér, editors, *Proceedings of the First International Workshop on Mash-Up Personal Learning Environments*, Maastricht, The Netherlands, 2008.

[2] Gaston Burek, Dale Gerdemann, Stefan Trausan-Matu, Traian Rebedea, Mathieu Loiseau, Philippe Dessus, Benoit Lemaire, Fridolin Wild, Debra Haley, Lucas Anastasiou, Bernhard Hoisl, Thomas Markus, Eline Westerhout, and Paola Monachesi. D2.5 LTfLL Roadmap. Available at: `http://dspace.ou.nl/handle/1820/3293`, 2011. Last accessed: 2012-05-22.

[3] Dave Crane and Phil McCarthy. *Comet and Reverse Ajax: The Next-Generation Ajax 2.0*. Apress, Berkely, USA, 2008.

[4] Neophytos Demetriou and Bernhard Hoisl. OpenACS-based XOTcl Widget Server and Connector Framework. Available at: `http://ltfll.svn.sourceforge.net/viewvc/ltfll/Wp2/xotcl-widgets/`, 2010. Last accessed: 2012-05-22.

[5] Reinhard Dietl, Bernhard Hoisl, Fridolin Wild, Berit Richter, Markus Essl, and Gerhard Doppler. D2.1 Services Approach & Overview General Tools and Resources. Available at: `http://dspace.ou.nl/handle/1820/1707`, 2008. Last accessed: 2012-05-22.

[6] Reinhard Dietl, Fridolin Wild, Bernhard Hoisl, Robert Koblischke, Berit Richter, Paola Monachesi, Kiril Simov, Traian Rebedea, Sonia Mandin, and Virginie Zampa. D2.2 Existing Services – Integrated. Available at: `http://dspace.ou.nl/handle/1820/2041`, 2009. Last accessed: 2012-05-22.

[7] Google. Gadget-to-Gadget Communication (Deprecated). Available at: `http://code.google.com/apis/gadgets/docs/pubsub.html`, 2012. Last accessed: 2012-05-22.

[8] Björn Hartmann, Scott Doorley, and Scott Klemmer. Hacking, Mashing, Gluing: Understanding Opportunistic Design. *Pervasive Computing, IEEE*, 7(3):46–54, 2008.

[9] Jan Hensgens, Ellen Rusman, Jan van Bruggen, Gillian Armitt, Petya Osenova, and Kiril Simov. D3.2 Designing the LTfLL Services: Guidelines, Scenarios and Commonalities. Available at: `http://dspace.ou.nl/handle/1820/2038`, 2009. Last accessed: 2012-05-22.

[10] Henry Hermans and Steven Verjans. Developing a Sustainable, Student centred VLE: the OUNL Case. In *Proceedings of the 23rd ICDE World Conference on Open Learning and Distance Education including the 2009 EADTU Annual Conference, June, 7-10*, Maastricht, The Netherlands, 2009.

[11] Alan Hevner, Salvatore March, Jinsoo Park, and Sudha Ram. Design Science in Information Systems Research. *MIS Quarterly*, 28(1):75–105, 2004.

[12] Bernhard Hoisl. Elgg Community – Wookie Widgets 2.0. Available at: `http://community.elgg.org/pg/plugins/release/322307/developer/hoisl/wookie-widgets-20`, 2009. Last accessed: 2012-05-22.

[13] Bernhard Hoisl. Elgg Community – Wookie Widgets 2.1. Available at: `http://community.elgg.org/pg/plugins/release/323321/developer/hoisl/wookie-widgets-21`, 2009. Last accessed: 2012-05-22.

[14] Bernhard Hoisl. Elgg Community – Wookie Widgets 2.2. Available at: `http://community.elgg.org/pg/plugins/release/420305/developer/hoisl/wookie-widgets-21`, 2009. Last accessed: 2012-05-22.

[15] Bernhard Hoisl. Wookie IWC Server. Available at: `http://ltfll.svn.sourceforge.net/viewvc/ltfll/Wp2/wookie-iwc/`, 2010. Last accessed: 2012-05-22.

[16] Bernhard Hoisl. A Mash-up Architecture for Learning Environments and Knowledge Management Systems. In Ronald Maier, editor, *Proceedings of the 6th Conference on Professional Knowledge Management (WM 2011)*, pages 33–37, Innsbruck, Austria, 2011.

[17] Bernhard Hoisl, Hendrik Drachsler, and Christoph Waglechner. User-tailored Inter-Widget Communication – Extending the Shared Data Interface for the Apache Wookie Engine. In *Proceedings of the 13th International Conference*

on Interactive Computer Aided Learning (ICL2010), pages 1123–1131, Kassel, Germany, 2010. Kassel University Press.

[18] Bernhard Hoisl, Markus Essl, and Helmut Kometter. LTfLL Widget Template. Available at: `http://ltfll.svn.sourceforge.net/viewvc/ltfll/Wp2/widgets/_template/src/`, 2010. Last accessed: 2012-05-22.

[19] Bernhard Hoisl, Debra Haley, Fridolin Wild, Lucas Anastasiou, Katja Buelow, Robert Koblischke, Gaston Burek, Mathieu Loiseau, Thomas Markus, Traian Rebedea, Hendrik Drachsler, Helmut Kometter, Eline Westerhout, and Vlad Posea. Building a Personal Learning Environment with Language-Technology-based Widgets: Services v2 – Integrated Thread. Available at: `http://dspace.ou.nl/handle/1820/3076`, 2010. Last accessed: 2012-05-22.

[20] Torstein Honsi. Highslide JS – JavaScript Thumbnail and Media Viewer. Available at: `http://highslide.com`, 2012. Last accessed: 2012-05-22.

[21] Kevin Jardine. Elgg Community – OpenID Client. Available at: `http://community.elgg.org/pg/plugins/project/433999/developer/kevin/openid-client`, 2010. Last accessed: 2012-05-22.

[22] Sam Kanan. Elgg Community – Tabbed dashboard and/or profile. Available at: `http://community.elgg.org/pg/plugins/release/61851/developer/sammykanan/tabbed-dashboard-andor-profile`, 2010. Last accessed: 2012-05-22.

[23] Oleg Liber and Mark Johnson. Special Issue on Personal Learning Environments. In *Interactive Learning Environments*, volume 16. John Wiley & Sons, Ltd, 2008.

[24] LTfLL Consortium. LTfLL Consortium's Approach to Integration – Additional Report. Available at: `http://dspace.ou.nl/handle/1820/2040`, 2009. Last accessed: 2012-05-22.

[25] Ken Peffers, Tuure Tuunanen, Marcus Rothenberger, and Samir Chatterjee. A Design Science Research Methodology for Information Systems Research. *Journal of Management Information Systems*, 24(3):45–77, 2007.

[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, Spain, April 23*, pages 1115–1116. ACM New York, NY, USA, 2009.

[27] Behnam Taraghi, Martin Ebner, and Sandra Schaffert. Personal Learning Environments for Higher Education: A Mashup Based Widget Concept. In *Proceedings of the 2nd Workshop on Mash-Up Personal Learning Environments (MUPPLE'09)*, Nice, France, 2009.

[28] W3C. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language – W3C Recommendation 26 June 2007. Available at: `http://www.w3.org/TR/wsdl20/`, 2007. Last accessed: 2012-05-22.

[29] W3C. HTML 5: A Vocabulary and Associated APIs for HTML and XHTML – W3C Working Draft 12 February 2009. Available at: `http://www.w3.org/TR/2009/WD-html5-20090212/comms.html`, 2009. Last accessed: 2012-05-22.

[30] W3C. XMLHttpRequest – W3C Candidate Recommendation 3 August 2010. Available at: `http://www.w3.org/TR/XMLHttpRequest/`, 2010. Last accessed: 2012-05-22.

[31] W3C. Widget Packaging and XML Configuration – W3C Recommendation 27 September 2011. Available at: `http://www.w3.org/TR/widgets/`, 2011. Last accessed: 2012-05-22.

[32] W3C. CSS Validation Service. Available at: `http://jigsaw.w3.org/css-validator/`, 2012. Last accessed: 2012-05-22.

[33] W3C. Markup Validation Service. Available at: `http://validator.w3.org`, 2012. Last accessed: 2012-05-22.

[34] W3Schools. Browser Display Statistics. Available at: `http://www.w3schools.com/browsers/browsers_display.asp`, 2012. Last accessed: 2012-05-22.

[35] W3Schools. Browser Statistics. Available at: `http://www.w3schools.com/browsers/browsers_stats.asp`, 2012. Last accessed: 2012-05-22.

[36] W3Schools. OS Platform Statistics. Available at: `http://www.w3schools.com/browsers/browsers_os.asp`, 2012. Last accessed: 2012-05-22.

[37] WebKnow.com. Javascript Enabled Statistics. Available at: `http://www.webknow.com/scripts-enabled-statistics.asp`, 2012. Last accessed: 2012-05-22.

[38] Fridolin Wild, Katja Buelow, Bernhard Hoisl, Robert Koblischke, Markus Essl, Traian Rebedea, Vlad Posea, Thomas Markus, Eline Westerhout, Sonia Mandin,

Hendrik Drachsler, Gaston Burek, Hristo Kostov, and Alex Simov. D2.3 Services v1 Integrated. Available at: http://dspace.ou.nl/handle/1820/2345, 2010. Last accessed: 2012-05-22.

[39] Fridolin Wild, Bernhard Hoisl, and Gaston Burek. Positioning for Conceptual Development using Latent Semantic Analysis. In Roberto Basili and Marco Pennacchiotti, editors, *Proceedings of the EACL 2009 Workshop on GEMS: Geometrical Models of Natural Language Semantics*, pages 41–48, Athens, Greece, 2009. Association for Computational Linguistics.

[40] Fridolin Wild, Felix Mödritscher, and Steinn Sigurdarson. Designing for Change: Mash-Up Personal Learning Environments. *eLearning Papers*, 2008.

[41] Fridolin Wild and Stefan Sobernig. Interoperability Framework Draft for the Distributed Open Virtual Learning Environment. Available at: http://www.icamp.eu/wp-content/uploads/2007/05/d31___icamp___interoperability-framework-draft.pdf, 2006. Last accessed: 2012-05-22.

[42] Joanna Wild, Fridolin Wild, Marco Kalz, Margit Hofer, and Marcus Specht. The MUPPLE Competence Continuum. In *Proceedings of the 2nd Workshop on Mash-Up Personal Learning Environments (EC-TEL 2009)*, pages 80 – 88, Nice, France, 2009.

[43] Scott Wilson. Wookie – Widget Developer's Guide. Available at: https://svn.apache.org/repos/asf/incubator/wookie/trunk/docs/legacy/widget_dev_guide_09.doc, 2009. Last accessed: 2012-05-22.

[44] Scott Wilson and Ross Gardler. Wookie API Reference. Available at: http://incubator.apache.org/wookie/docs/api.html, 2012. Last accessed: 2012-05-22.

[45] Scott Wilson and Kris Popat. Moodle Modules and Plugins – Block: Wookie. Available at: http://moodle.org/mod/data/view.php?rid=3319, 2010. Last accessed: 2012-05-22.

[46] Scott Wilson, Kris Popat, and Ross Gardler. Welcome to Apache Wookie (Incubating). Available at: http://incubator.apache.org/wookie/, 2012. Last accessed: 2012-05-22.

[47] Scott Wilson, Ivan Zuzak, Bernhard Hoisl, and Ross Gardler. [#WOOKIE-133] Implement Inter-Widget Messaging. Available at: https://issues.

apache.org/jira/browse/WOOKIE-133, 2010. Last accessed: 2012-05-22.