Tampereen teknillinen yliopisto. Julkaisu 1116
Tampere University of Technology. Publication 1116

Arto Salminen

Mashup Ecosystems: Integrating Web Resources on Desktop and Mobile Devices

Thesis for the degree of Doctor of Science in Technology to be presented with due permission for public examination and criticism in Tietotalo Building, Auditorium TB223, at Tampere University of Technology, on the 1st of March 2013, at 12 noon.

Tampereen teknillinen yliopisto - Tampere University of Technology
Tampere 2013

ISBN 978-952-15-3022-7
ISSN 1459-2045

Abstract

The Web is increasingly used as an application platform, and recent development of it has introduced software ecosystems where different actors collaborate. This collaboration is international from day one, and it evolves and grows rapidly. In web ecosystems applications are provided as services, and interdependencies between ecosystem parts can vary from very strong and obvious to loose and recondite. Mashups – web application hybrids that combine resources from different services into an integrated system that has increased value from user perspective – are exploiting services of the Web and creating ecosystems where end-users, mashup authors, and service providers collaborate. The term "resources" is used here in a broad sense, and it can refer to user's local data, infinite content of the Web, and even executable code. This dissertation presents mashups as a new breed of web applications that are intended for parsing the web content into an easily accessed form on both regular desktop computers as well as on mobile devices.

Constantly evolving web technologies and new web services open up unforeseen possibilities for mashup development. However, developing mashups with current methods and tools for existing deployment environments is challenging. First, the Web as an application platform faces numerous shortcomings, second, web application development practices in general are still immature, and third, development of mashups has additional requirements that need to be addressed. In addition, mobility sets even more challenges for mashup authoring.

This dissertation describes and addresses numerous issues regarding mashup ecosystems and client-side mashup development. To achieve this, we have implemented technical research artifacts including mashup ecosystems and

different kinds of mashup compositions. The artifacts are developed with numerous runtime environments and tools and targeted at different end-user platforms. This has allowed us to evaluate methods, tools, and practises used during the implementation.

As result, this dissertation identifies the fundamental challenges of mashup ecosystems and describes how service providers and mashup ecosystem authors can address these challenges in practice. In addition, example implementation of a specialized multimedia mashup ecosystem for mobile devices is described. To address mashup development issues, this dissertation introduces practical guidelines and a reference architecture that can be applied when mashups are created with traditional web development tools. Moreover, environments that can be used on mobile devices to create mashups that have access to both web and local resources are introduced. Finally, a novel approach to web software development – creating software as a mashup – is introduced, and a realization of such concept is described.

Preface

This thesis would not have been possible to finish without support of numerous colleagues, friends, and family members.

First of all I thank my supervisor, Professor Tommi Mikkonen, for support, advice, and ideas. Naturally, I thank the pre-examiners of this thesis, Professor Pasi Tyrväinen from University of Jyväskylä, Finland, and Associate Professor Muhammad Ali Babar from IT University of Copenhagen, Denmark, for reviewing the manuscript, and Professor Mehdi Jazayeri from University of Lugano, Switzerland, for being my opponent at the public examination. In addition, I thank co-authors Mikko Hartikainen, Jarno Kallio, Feetu Nyrhinen, and Antero Taivalsaari for their contributions.

Furthermore, I would like to gratefully acknowledge the organizations that supported my thesis. I have been enjoying a priviledge to work at the Department of Software Systems, nowadays known as the Department of Pervasive Computing at Tampere University of Technology. The projects where the work was carried out were funded by the Finnish Funding Agency for Technology and Innovation (Tekes), the Strategic Centre for Science, Technology and Innovation in the Field of ICT (TiViT), and the Academy of Finland. In addition, the work has been funded by Graduate School on Software and Systems Engineering (SoSE) and the Nokia Foundation.

Finally, I would like to express my gratitude to my wife, Pia, for her love and patience throughout the project and while travelling.

Arto Salminen
Tampere, February 18, 2013

Contents

List of Included Publications

I A. Salminen and T. Mikkonen, "Mashups – Software ecosystems for the web era". In Proceedings of ICSOB'2012 4th International Workshop on Software Ecosystems (IWSECO'2012, Boston, MA, USA, June 18–20, 2012). Sun SITE Central Europe CEUR, Aachen, Germany.

II A. Salminen, J. Kallio and T. Mikkonen, "Towards mobile multimedia mashup ecosystem". In Proceedings of IEEE ICC 2011 Workshop on Advances in Mobile Networking – "Towards a Next Generation Mobile Core Network" (ICC'2011, Kyoto, Japan, June 5–9, 2011). IEEE Computer Society, Los Alamitos, CA, USA.

III M. Hartikainen, A. Salminen and J. Kallio, "Towards mobile multimedia mashup architecture". In Proceedings of 38th Euromicro Conference on Software Engineering and Advanced Applications (SEAA'2012, Cesme, Izmir, Turkey, September 5–8, 2012). IEEE Computer Society, Los Alamitos, CA, USA.

IV F. Nyrhinen, A. Salminen, T. Mikkonen and A. Taivalsaari, "Lively mashups for mobile devices". In Proceedings of the First International Conference on Mobile Computing, Applications and Services (MobiCase'2009, San Diego, CA, October 26–29, 2009). Springer-Verlag, Berlin, Heidelberg.

V A. Salminen and T. Mikkonen, "Towards pervasive mashups in embedded devices: Comparing procedural and declarative approach". To appear in Special Issue on Techniques and Applications for Merging Mobile and Cloud Services, International Journal of Communication Networks and Distributed Systems (IJCNDS), Vol. 10, No. 3, 2013, Inderscience Publishers.

VI A. Salminen, F. Nyrhinen, T. Mikkonen and A. Taivalsaari, "Developing client-side mashups: Experiences, guidelines and reference architecture". Artur Lugmayr, Olli Sotamaa, Heljä Franssila, and Hannu Kärkkäinen (eds.), To appear in Special issue on Ambient and Social Media Business and Application, International Journal of Ambient Computing and Intelligence (IJACI), Vol. 5, No. 1, January-March 2013, IGI Publishing.

VII A. Salminen, "Mashups in Web 3.0". In Proceedings of 8th International Conference on Web Information Systems and Technologies (WebIST'2012, Porto, Portugal, April 18–21, 2012). Science and Technology Publications (SciTePress).

VIII T. Mikkonen and A. Salminen, "Implementing mobile mashware architecture: Downloadable components as on-demand services". In Proceedings of The 9th International Conference on Mobile Web Information Systems (MobiWIS'2012, Niagara Falls, Ontario, Canada, August 27–29, 2012). Procedia Computer Science.

1

Introduction

Let's first consider an example *mashup*: Google Maps, a map service that was published in February 2005 by Google. At the time of publishing, the application mashed up only Google's own services: searching and mapping [1]. For instance, one could look for "hotels near LAX" and see the result plotted as pushpins on the map. The application was found to be very useful, and it was later enhanced with new capabilities including an ability to add third party content such as encyclopedia articles and web camera images into the mashup. However, the Web (World Wide Web) has come a long way to enable access to this kind of rich applications through a web browser. Consequently, the topic of this dissertation, mashups, is about stretching the limits of the underlying technology to enable useful and even fascinating applications for the end-user.

Within 20 years the Web has transformed from a simple document presenting system into a rich application platform. When conceived, the original purpose of the Web was presenting documents called hypertext[1] containing static elements that were connected to each other with links. Hypertext documents written in domain-specific markup language called HTML [3] were later enhanced with support for images and form-based data entry. At the late 1990s web browser capabilities were expanded with dynamic HTML (DHTML) technologies that made possible to add interactive and animated elements into static documents. The so-called DHTML includes support for client-side scripting language, separate presentation definition language, and a scripting interface for web browsers. Currently, the *de facto* client-side programming language

[1]The prefix *hyper* means "over" or "beyond", and it signifies the overcoming of the old linear constraints of written text. The term "hypertext" is often used where the term "hypermedia" might seem appropriate. Both terms are coined by Ted Nelson [2].

is JavaScript [4], which is formalized in the ECMAScript language standard [5], and the presentation definition language is Cascading Style Sheets (CSS) [6]. The scripting interface in browsers is referred to as Document Object Model (DOM) [7]. Later, plug-ins, such as Adobe Flash [8] and Java Applets [9], were introduced and targeted at embedding interactive elements as well as advanced scripting and graphics into web pages.

As browser capabilities have been evolving, more complex web applications have been implemented. In the mid 1990s the most popular browser-based web applications were e-mail applications such as HoTMaiL (known today as Windows Live Hotmail, `http://mail.live.com`) and RocketMail (known today as Yahoo! Mail, `http://mail.yahoo.com`). However, before Ajax (Asynchronous JavaScript And XML) [10], a way to download and display data in a piecemeal fashion, was introduced, the whole view was reloaded when there was need to communicate with the server back-end and to update the user interface correspondingly. This was inefficient and caused the user experience of web applications to be poor compared to their desktop counterparts. Ajax, along with numerous other developments such as faster JavaScript and browser rendering engines, turned the Web into a pervasive software platform that is accessible everywhere, at anytime, for everyone. The latest additions that are introduced in HTML5 standard [11] include offline functionality and access to device peripherals, facilities that have been typical for operating systems. Therefore, it has been argued that web browsers are taking the role of operating system as platform and resource allocator for applications [12, 13]. All this has dramatically changed the way people develop, deploy, and use software. However, document-oriented origins of the web are still evident in many areas, and traditionally web application composing has been difficult without plug-in components or browser extensions.

The paradigm shift towards the Web is obvious in numerous things that people use daily. For instance newspapers and books as well as music and movie industries are currently experiencing a transition to the Web. Similarly, the software industry is currently in the middle of a paradigm shift from static binary applications into dynamic web-based applications that are executed inside a web browser or with a special runtime environment. The new capabilities of web browsers and certain advantages of web-based software have catalyzed this transition. Consequently, we now have full-fledged browser-based alternatives for e-mail clients, calendars, spreadsheet editors, word processors,

slide presentation applications, map applications, travel reservation systems, image and sound manipulation applications as well as 2D and 3D games, among others.

Applications live in the Web as services. The delivery model of the Web, on-demand software or Software-as-a-Service (SaaS), implies that installations or manual upgrades are no longer required for end-user. These services, or web applications, can be used to gain access to and perform operations with content stored in remote systems. This model of operation is often referred to as "cloud computing". In this sense, the Web is now the first global, uniform distribution channel that can be accessed in pervasive way with different terminals such as desktop computers, mobile phones, as well as other embedded devices. This capability of instant worldwide deployment makes the Web superior to conventional binary application platforms [12].

Data, content artifacts, and other resources released to the Web are available all over the world instantly after publication, similarly to web applications. Web applications can support user collaboration, and allow users to interact and share the same resources over the Web, sometimes even simultaneously. In addition, numerous web services allowing users to upload, download, store, and modify private and public resources have emerged. These resources can include personal data such as images, texts, videos, and e-mails, as well as public data such as stock quotes, weather data, and news feeds. As the amount of web services and devices used to consume data has exploded, it is difficult to handle and gain access to the relevant data. To be able to handle the situation, searching has become one of the most important services of the Web. However, searching, if the implementation is simple, can be used only for data accessing, not for analyzing it. Similarly to resources, communication has decentralized into different services including e-mail, different social media services, instant messaging services, chats, and blogs. Therefore, there is a need to develop new mechanisms to utilize resource handling and communication services of the Web. Moreover, the Web provides an opportunity for collaboration not only for end-users sharing their personal data, but for developers sharing their code as well.

Recent development of the Web has introduced software ecosystems [14] where different actors, such as platform providers, application developers, standardization organizations, users, and manufacturers of web-enabled devices, among others, collaborate. As the Web is utilized as a platform, the collaboration is international from day one, and it evolves and grows in rapid pace. In web ecosystems, interdependencies between

ecosystem parts can vary from very strong and obvious to loose and recondite. Ecosystems have become important means of collaboration and competition, too. Often users that have become firmly binded to one ecosystem will not likely commit to another. Furthermore, ecosystems can be organized, controlled, and systematically evolved as well as unorganized, unmanaged, and evolving in aimless manner. The nature of an ecosystem has a profound effect on what kind of business models can emerge within it.

A mashup can be defined as an application that combines resources – data, code, and content artifacts – from different services over the Web into an integrated experience that has increased value for end-users. Mashups can combine the content in a new, unforeseen way, thus creating entirely new web service, or they can provide new visualizations for existing services. For instance, a mashup can combine map with images that can be attached on specific locations. Another type of mashup can visualize the images in a novel fashion, for example on a timeline or as a collage.

This dissertation presents the current state of web service-based mashup ecosystem development. Numerous problems that developers encounter when developing such applications are described. Consequently, solutions and best practises are presented, along with a general reference architecture for mashups. Finally, findings of the research are applied in the concept of *mashware*, software developed as a mashup, and as a result the first mashware application framework is realized.

1.1 Motivation

This dissertation focuses on new ways to build applications with web service-based approach. An important realization is that applications built on top of the Web do not have to live by the same constraints that have characterized the evolution of conventional desktop software. The ability to dynamically combine content from numerous web sites and local resources, and the ability to instantly publish services worldwide has opened up entirely new possibilities for software development. In general, such systems are referred to as mashups, which are content aggregates that leverage the power of the Web to support instant, worldwide sharing of content. We argue that this model of building applications will be very successful, and that it will evolve into something more compelling: software created as a mashup.

As mashups access different services and enable users as well as enterprises to collaborate, it can be argued that mashup ecosystems can be formed. Described on a high level, mashup users, mashup authors, and service providers collaborate in mashup ecosystems. Mashup users can add content to services and consume the content in extended form through mashups. Mashup ecosystems are not necessarily controlled explicitly by a central authority. In contrast, even though mashup authors and service providers may have explicit service-level agreements (SLAs) and terms of services (TOS), it is common that mashups are developed without such contracts, and consequently, ecosystems are formed implicitly. This makes it possible for mashup ecosystems to evolve constantly and adapt to user requirements. Despite the increasing popularity of mashup composing, there are stumbling stones such as interface reliability, lack of tool support, and legal matters that hinder the development of mashup ecosystems.

Mashups have potential for great user experiences, as they include more functions than just adding content artifacts to a single view. Mashups can be used to filter, combine, and modify data retrieved from multiple sources over the Web. Combining web resources into mashups is an efficient way to create new services or extract relevant information from complex mixture of source data. Even unexpected innovations are possible as mashups can combine resources in unforeseen way. Furthermore, mashups are even more usable when non-technical users create them with special purpose tools and have their own views for the data. Some mashups may implement mechanisms that allow non-technical users to alter the mashup functionality. This is a very inspiring part of mashups as it allows creative users to design their own applications that are capable to do unexpected things. Allowing "do-it-yourself" mashups serve the long tail of users having diverse needs that are not fulfilled by existing applications or services. Often this kind of possibility is enabled with a specialized tool that can be used for mashup composing out of prefabricated components.

Well-build mashups have functionality for filtering source data. By having adjustable filters a mashup can provide relevant results for the user's needs. Filters can be based on much more relevant variables than manually entered limits such as the highest and the lowest price of a product. Such filters can be time of the day, location of the user, his past activity, activity of other users (trends), profile setting of users mobile device, etc. In embedded devices, mobile devices being at the forefront, mashups can benefit from accessing the user's context thus being able to combine resources

even automatically. Heavy processing, e.g. filtering images with a face detection algorithm, can be executed on the server, using MashReduce programming model [15], for instance.

Different kinds of dependability mechanisms play an important role in a mashup. At least the mashup should be implemented so that it checks whether the input data is correct. More sophisticated mashups can have fall-back mechanisms that, instead just giving up on error, try to use next best strategy to ensure even partial functionality. Furthermore, mashups can have controlling mechanisms that supervise the functionality and replace the failing parts with other ones. In addition, mashups can have capabilities to extract the result mashup to some external viewing device and change the user interface of the mashup accordingly. For instance, this allows a mashup to be created in a mobile device and the result to be shown on a bigger screen.

Typically mashups are built with combination of server- and client-side parts. Functionality between these two parts is divided according to what is suitable for the current design. In the early days, dynamic web sites were created on server-side with combination of C programs, Perl, and shell scripts using Common Gateway Interface (CGI). Today, server-side web applications are often developed with Java, server-side JavaScript, Perl, PHP, Python, or other suitable language. This kind of applications work especially well if the client device has low processing resources as heavy processing can take place at the server-end and the client just renders the result. As client-side terminals have gained more competence, it has become possible to compose mashups where the business logic resides completely on the client-side. While this dissertation touches on server-end technologies, the main focus is on the client-side implementation of mashups.

1.2 Deriving Research Questions

Since mashups by definition combine data from multiple sources, the stakeholders that provide and consume this data form an ecosystem. The mashup ecosystem can be considered to include all possible web services and mashups build on top of them all over the world. However, subsystems of this global ecosystem can be considered as well. These smaller mashup ecosystems can be explicit or implicit, depending on how they are formed. One can define an ecosystem explicitly by allowing a fixed set of services

to be used in mashups. On the other hand, mashup ecosystems need not be controlled by a central authority, and implicit ecosystems emerge when arbitrary services are used in mashups.

When a mashup ecosystem is formed, numerous implementation issues as well as legal aspects need to be taken into account. In practice, these issues are highly related to the set of services provided, mashups implemented, and requirements set by the ecosystem users and other stakeholders. For instance, a closed ecosystem with mashups developed for desktop systems has a different set of requirements than an ecosystem targeted at mobile devices with a changing set of source services. Consequently, the first research question that this dissertation addresses is:

> **RQ1.** How to design mashup ecosystems where end-users, mashup authors, and service providers collaborate?

Mashup development is not a craft free of complications. First, the Web as an application platform faces numerous shortcomings. Second, engineering and architecting support for web application development is still immature. Third, mashups that are build on the services of the Web have special requirements that need to be addressed. Furthermore, in addition to desktop systems, mashups can be implemented for mobile domain, as well. While mobility introduces additional challenges for mashup development, the dynamic nature of mashups suits well for different ways mobile terminals can be used. Mixing web resources with the capabilities of mobile devices allows to build mashups that take the user's context into account and automatically provide relevant information. This derives the second research question:

> **RQ2.** How to solve problems related to client-side mashup development on desktop and mobile devices?

Mashups are successful utilizing the Web as a platform for accessing services providing content through different types of web interfaces. However, in addition to content artifacts, executable application components can be used in mashups to create software in piecemeal fashion. This kind of software created as a mashup is originally presented in [16, 17] and referred to as *mashware*. Despite significant benefits that such system would have, mashware has not been realized even as a laboratory prototype. Therefore, the third research question of the dissertation is:

RQ3. How to realize software as a mashup?

The research questions are addressed in the included publication as follows. RQ1 is addressed in Publications I, II, and III; RQ2 is addressed in Publications IV, V, and VI; and RQ3 is addressed in Publications VII and VIII.

1.3 Research Methods

Taxonomy of research approaches for information systems development has been proposed by Järvinen [18]. The taxonomy divides research approaches into *Mathematical approaches* and *Approaches studying reality*, where the former concerns symbol systems without direct reference to objects in reality, such as formal languages or algebraic units. The information systems research concerns the latter, which is divided further to two subcategories *Researches stressing what is reality* and *Researches stressing utility of artifacts*. The former has another subcategory *Approaches for empirical studies*. Consequently, five research approaches in addition to *Mathematical approaches* can be organized into these categories as follows (research approaches are emphasized with a boldface font).

- **Mathematical approaches**
- Approaches studying reality
  - Researches stressing what is reality
    - **Conceptual-analytical approaches**
    - Approaches for empirical studies
      - **Theory-testing approaches**
      - **Theory-creating approaches**
  - Researches stressing utility of artifacts
    - **Artifacts-building approaches**
    - **Artifacts-evaluating approaches**

In this dissertation we have built software artifacts and evaluated the artifacts to determine if requirements set for the artifacts are fulfilled. Therefore, we have followed methods categorized as *Researches stressing utility of artifacts*. In the artifact building and evaluation we have carried out action design research (ADR) [19], which includes the following phases: 1) problem formulation, 2) building, intervention, and evaluation,

3) reflection and learning, and 4) formalization of learning [19]. Consequently, our problem formulation has been practise-inspired, and the building of the artifacts has been carried out at the same process as the evaluation, and we have been able to seek a solution to the problem and simultaneously study the experience of solving the problem. During the research we applied ADR in two research areas: mashup ecosystem and mashup artifact implementation research, and mashware implementation research. In the following we describe our research efforts in these areas in more detail.

- **Mashup ecosystem and mashup artifact implementation research.** To study mashup ecosystems and mashup implementations, we carried out research cycles where we examined the existing ecosystems and implemented new ones. In addition, we performed research about mashups on embedded devices and implemented research artifacts for such platforms as well. The research cycles taken are described as follows.

    1. The first research cycle was about studying mashup ecosystems and implementing one for a specific purpose – playing web video content on mobile devices, in this case. To gain knowledge about the current research and state of the art we first studied the existing mashup ecosystems. Then we identified the most relevant challenges we expected when a mobile mashup ecosystem would be realized. Finally, we implemented a mobile mashup ecosystem for Android mobile devices. The results of this research cycle are documented in Publications I, II, and III. The most significant results in this cycle were: 1) identifying the four levels where mashups can be supported on by service providers, 2) identifying the legal constraints of mashup ecosystems, 3) pointing out numerous implementation considerations, and 4) forming a mashup ecosystem architecture with access to web and local content.

    2. The second research cycle focused on implementing client-side mashups for desktop and embedded devices, particularly mobile phones. To implement these mashups we used JavaScript as the programming language and different runtime environments to study portability and feasibility of such applications on different types of devices. The results of the this research cycle are

documented in Publications IV, V, and VI. The most significant results in this cycle were: 1) implementing a runtime environment for cross-platform mobile and desktop mashups, 2) indentifying practical issues when implementing mashups in general and especially on mobile devices, 3) using a programmatic as well as declarative runtime to host mashup applications on embedded devices and comparison of these two, 4) thirteen guidelines for mashup developers, and 5) a reference architecture for mashups.

- **Mashware implementation research.** To study implementing mashware, we performed third research cycle where the current state of the art was reviewed and a mashware application was realized. While performing the literature review we found out that concepts of mashware are close to what has been suggested to be included under term "Web 3.0". In building and evaluation phase we applied the previous results about mashups to mashware, where software components are combined similarly to content in mashups. The results of this research cycle were described in Publications VII and VIII. In this research cycle we 1) pointed out that mashups and mashware are at the core of what is known as Web 3.0, and 2) we implemented a mashware application with architecture that allows to load software components as an on-demand service.

1.4 Main Contributions

The main contributions of this dissertation are practical solutions addressing issues on three main areas of mashup development: mashup ecosystems, mashup development, and software as a mashup. The contributions are described as follows.

- **Mashup ecosystems.** This dissertation introduces mashup ecosystems, where mashup users, mashup authors and service providers collaborate. Dissertation identifies numerous mashup ecosystem challenges and presents four levels of support that service providers can offer for mashups. In addition, we describe practical issues that are faced when a specialized mobile multimedia mashup ecosystem is implemented.

  Technical Contributions. As a technical contribution we collaborated with an industry partner in building a mashup ecosystem for aggregating videos from

multiple web video services. As the industry partner had some parts of the system (mainly on the server-side) already implemented, the solution utilized those in the final implementation, which consisted of a video client for Android mobile terminals and a server backend. This technical contribution is explained in more detail in Publication III, where the ecosystem architecture implementation is declared. Our preliminary research in preparing to build this system can be found in Publication II. There is no technical artifact directly related to Publication I, as the scope of it was to take a broader perspective and describe different goals and considerations of mashup ecosystems. However, this paper describes implementation considerations that arise from taking a retrospective view on the video mashup ecosystem project. Furthermore, the paper identifies four levels where service providers support mashup developers. These findings are based on our other efforts to build mashups described under the "technical contributions" section of the next bullet.

- **Mashup development.** Mashup development has special requirements that are not addressed with current engineering practices, implementation techniques, security models, or architectures. This dissertation introduces practical guidance that can be applied when mashups are authored with traditional web development tools. Furthermore, we describe a reference architecture for mashups that directs mashup developers towards well-defined structure in their implementation. The architecture promotes maintainability and flexibility of mashup implementations.

  Mashups can be implemented on both desktop and mobile devices, as well as other embedded devices. Creating mashups for devices with restricted capabilities has special requirements that are described and addressed in this dissertation. For embedded devices, mobile phones included, we introduce two runtime environments that can be used for accessing local resources of a device, including files and peripheral data.

  Technical Contributions. During the research we have implemented numerous mashups, all of which are not described in the included publications. In Publication IV, we describe five mashup artifacts built on a special runtime environment, and in Publication VI, three other mashup artifacts are described. Furthermore, in Publication V, we describe the same mashup artifact implemented with two

different approaches. The latter approach described in Publication V is one of the three example mashups described in Publication VI. In addition to these nine mashup artifacts, our contributions in mashup building include the video mashup client described earlier. Moreover, there are numerous other mashup artifacts build by the candidate that are not described in the publications or in this dissertation introduction.

- **Software as a mashup.** Composing mashups is not limited to content artifacts, but executable application components can be composed together as well. This dissertation summarizes the idea suggested in the literature of such application development model, or in other words, software created as a mashup. Here, we portray the mashware playing an important role in the future of the Web. In addition, we describe the requirements of such system, the most significant issues of this approach, and a proof-of-concept implementation of software created as a mashup.

  Technical Contributions. As a technical contribution we built a client-side mashware implementation described in Publication VIII that utilizes a simple repository holding meta data of software components. The repository can be used to search and download suitable components to be used in a mashware application. Our mashware artifact has a client-side mechanism to make these downloaded components available on the fly in the running software. In development of the mashware artifact we have applied the reference architecture described in Publication VI.

1.5 Organization of the Dissertation

This dissertation presents how the Web can be used as a platform for mashup ecosystems, how mashups can be composed in practice, and how software could be developed as a mashup. The following chapters of the introductory part of this dissertation are organized as follows.

Chapter 2 contains a broad background description about the Web as a mashup platform, and presents the related work. As a background information, an apporach to build service-based applications, service-oriented architectures (SOA) are described. The background includes a description of different types of mashups and approaches

that can be used for mashup development. Furthermore, different types of mashup runtimes and patterns are described.

Chapter 3 describes mashup ecosystems that can be formed implicitly or build explicitly. After providing an overview to mashup ecosystems, we point out the most relevant issues of mashup ecosystems and discuss how these issues could be addressed. Furthermore, we describe our practical implementation of a very specialized mashup ecosystem targeted at presenting multimedia content on mobile devices.

Chapter 4 includes numerous considerations about mashup composing in practice. After presenting a practical reference architecture, general design principles for mashups are discussed. Furthermore, we consider different methods that can be used to access web and local services. In addition, some specific issues that need to be taken into account when developing mobile mashups are pointed out. Security is one of the most difficult issues of mashup composing, and it is addressed as well.

Chapter 5 introduces the idea to compose software as a mashup. After providing an overview of the subject, we discuss about requirements of such systems and describe the most substantial issues that need to be overcome. Moreover, we describe our approach to develop such software along with a proof-of-concept implementation.

Finally, Chapter 6 provides conclusions of the research and revisits the main results of the work. Publications included in this dissertation are introduced, as well. In addition, future directions of mashup research are outlined.

2

Background and Related Work

In this Chapter, we provide an overview to service-based web application development as well as different types of mashups. In addition, we describe web interfaces, which are used to gain access to content – the main ingredient of mashups. Then, we introduce mashup development workflow with and without tools. Next, we describe existing mashup systems that are created as research artifacts. Finally, we present patterns and runtimes that are available for mashup development.

2.1 Hyperlinking, Portals, and Composite Applications

Mashups have developed from the basic concept of the Web, attaching hypertext documents via linking, into integrated applications [20, 21]. In the early days, we could find numerous link collections about different subjects. Today, web contains even full-featured application ecosystems with scalable server back-ends, multi-platform clients, and application stores.

Hyperlinking is one of fundamental concepts of the Web. Linking provides a simple way to navigate from one document to another and can be very easily be implemented by users. Often a hyperlink from a page to another implies that these two documents are related or that the page being linked to is recommended. Link collections, i.e., categorized lists of hyperlinks, are the first step towards accessing multiple sources of data simultaneously.

Hyperlinks provide a way to embed data from another domain into a single web document. Unfortunately, this mechanism is straightforward only with images and JavaScript code as the same origin policy restricts embedding content from arbitrary

sources [22, 23]. However, the policy has been regarded as inadequate against security threats in the Web, and consequently numerous enhancements for the same origin policy have been proposed [24, 25, 26]. In the domain of mashups the same origin policy has lead into complex ways to circumvent the restriction as described in [16].

Portals are the next step towards web applications that integrate data across domains. Portals have been used by governments, enterprises, as well as individuals to provide a starting point for web browsing. Typically, portals show information about the latest events and news, provide search functionality, and contain lists of links. Portals can be composed of "portlets", pluggable user interface software components. For instance, Web Services for Remote Portlets (WSRP) [27] is a network protocol standard for communication with remote portlets. Another portlet specification is Java Portlet Specification that defines a programming model for Java portlet developers. Java Portlet Specification version 1.0 was developed under Java Specification Request JSR-168 [28], and backward compatible version 2.0 under JSR-286 [29]. JSR-168, JSR-286, and WSRP are not competing technologies. JSR-168 may be used to define a portlet, and WSRP may be used to define a portlet's operations to remote containers. This way JSR-168 portlets and WSRP are be used together to define a portlet and to provide remote operations [30]. Apache Pluto (`http://portals.apache.org/pluto/`) is a reference implementation of JSR-286 and JSR-168, and it provides a runtime environment where portlets can be instantiated, used, and destroyed.

Mashups can be considered as a successor of portals as they often similarly combine content that is somehow related. However, it is important to make a clear distinction between portals and mashups. The difference is described in [31] and summarized in the following. In contrast to mashups portlets are isolated into their own units, the communication between portlets is little, and the aggregated content is presented side by side without overlaps. In mashups, the aggregation of content follows "melting pot" style where the user cannot distinguish data origins based on its appearance in the user interface. Usually, portal's content is aggregated exclusively in the server-side, unlike mashups that can be composed on either server- or client-side. Furthermore, mashups create something new from the information they are based on, instead of just aggregating it into a single view.

Another distinction to made is between mashups and *composite applications*. The term composite application refers to an application built by combining multiple ex-

isting components into a new application [32]. Typically composite applications use business sources of information, and they are developed in an enterprise service bus (ESB) or integrated composition environment (ICE), and therefore creating composite applications is often tool-oriented and based on special runtime environments instead of web browsers. Developing composite applications is therefore more difficult for the end-user and less flexible. Composite applications do not utilize web resources as widely as mashups that are typically build on web interfaces. However, as mashups and composite applications are so similar, both can be used as tools in enterprises, and even developed in similar fashion [33, 34, 35, 36].

2.2 Service-Oriented Architectures

Many concepts related to service-based web applications - mashups included - are introduced in an architecture style called *Service-Oriented Architecture* (SOA) [37]. Despite that term SOA is often used when discussing about web applications, the architecture concept can be used without particular implementation technology. SOA describes a set of principles and methodologies for designing and developing software as interoperable services.

SOA building elements include *service providers* and *service consumers*, as well as *interfaces* that are used by service consumers to communicate with service providers. Other modules that can be included to a SOA are *service registry* and *proxy*. The following description of SOA building elements is based on [37, 38].

Service providers. In SOA approach the software capabilities are exposed as loosely coupled services. Very important aspect is that service implementation is separated from the interface, so that service consumer does not have to care about implementation details. Therefore, SOA approach has major effect on how the software is managed during its life cycle.

Service consumers. Service consumers access services through service interfaces. It can be a client application, other service provider, or a software module. As a service is accessed through an interface, a service consumer needs to know only how to use the interface, not how the accessed service is implemented. In addition to the interface, also service location needs to be known by the service consumer to enable access.

Service interfaces. Service interfaces are central part of SOA approach. Interface acts as a contract between the service's client and the service provider. It defines request and response formats as well as other details such as data types, response times among other quality of service factors, and possible limitations.

Service registry. Service registry enables service discoverability by providing a directory of the available services. It contains meta data about service providers that service consumers can search. Web Services Description Language (WSDL) [39] and Universal Description Discovery and Integration (UDDI) [40] can be used to advertise web service's existence, and make it possible for service consumers to find and use them.

Service proxy. Service proxy can be used to simplify service consumer implementation. It works as a middleman finding a suitable service provider from the service registry. In addition, proxy can format consumer's requests to a right format and pass the request to a service provider. Furthermore, the proxy can cache information about services or provide some functionalities locally, thus reducing network traffic needed.

SOA characteristics. The most essential SOA characteristics are listed in [41] and summarized in the following.

- *Services are discoverable and dynamically bound.* Services can be discovered by searching the service registry and examining the meta data available. This can be done at runtime, and service consumers do not necessarily have to know about services at compile-time. Dynamic bounding requires also messages to be created dynamically. If a service proxy is available, the service consumer implementation is more straightforward.

- *Services are self-contained and modular.* Service modularity is a key concept in SOA. It can be divided into five fundamental requirements: decomposability, composability, understandability, continuity, and protection. Modular decomposability means that software problems are divided so independent subproblems, which can be solved separately. Modular composability ensures that software modules can be freely combined. Understandability indicates that a service module can be understood by a human without information about other modules. Continuity allows to make changes to modules without major changes to other modules. Protection refers to an ability to restrict faults within the module the error took place in.

- *Services stress interoperability.* Services that are interoperable can communicate with systems implemented with different platforms and languages. Interoperability is achieved by supporting the protocol and data formats of the services current and potential consumers.

- *Services are loosely coupled.* Loose coupling in SOA refers to a low number of dependencies between services. In contrast to tight coupling, modules do not have many unknown dependencies. This affects directly to system modifiability. By allowing as few dependencies as possible, service providers can be easily changed without need to make changes to service consumers. In loosely coupled systems a service consumer does not need much information about the service provider in order to use a service.

- *Services have a network-addressable interface.* In SOA, a service must be network addressable to enable a service consumer to access the service. This allows arbitrary services to be reused by arbitrary consumers at any time. Because parts of the system interact over the network, it is necessary to pay attention on interface performance.

- *Services have coarse-grained interfaces.* In SOA, services are coarse-grained. This means that a service implements relatively large part of the system and can be reused as part of another system. In contrast to fine-grained systems, where subsystems have a lot of dependencies, in SOA a service has relatively few dependencies. This reduces network traffic and simplifies the implementation.

- *Services are location-transparent.* Services are location-transparent, in other words a consumer does not know service location before searching the service registry. Dynamic binding and location transparency allow services to be relocated without consumers knowledge. This allows optimizing system performance by moving service producers closer to consumers.

- *Services are composable.* Modular structure of services allows unexpected compositions of services. This means that service developer can not anticipate all possible uses of the service at design time.

- *Service-oriented architecture supports self-healing.* Self-healing systems can recover internal errors independently. This is important as it simplifies developing large distributed systems.

In some cases, mashups can be viewed as applications build on SOA services, see for instance [42, 43, 44, 45]. Mashups can act as service consumers and utilize service providers as data sources, and access them by using a web enabled interface as a service interface. There is no global-scale service registry for mashups, even though ProgrammableWeb (`http://www.programmableweb.com`), a direcory of mashups and web services, lists a vast amount of interfaces. However, when a mashware system is developed, service registry is necessary to be able to locate useful mashware modules. Sometimes client-side mashups utilize a server-side proxy to perform operations that would be too heavy to be executed on a client-side device.

2.3 Different Types of Mashups

Mashups can be classified based on numerous criteria. One can classify mashups into categories based on technical implementation as follows:

- server- and client-side mashups, and
- multiple service and single service mashups.

In addition, classification can be based on the domain of the application as follows:

- commercial and non-commercial mashups,
- enterprise and consumer mashups, and
- situational mashups.

Furthermore, mashups can be classified according to the most essential service used, similarly to [46]. Some of these mashup classifications appear in the literature. Separation based on the implementation technology appears in [47] and separation to enterprise and consumer mashups in [47, 48, 49]. Situational applications are viewed as mashups in [50]. In the following, mashup categories are described and examples of different types of mashups are presented.

Server- and client-side mashups. One way to classify mashups is division between client-side and server-side mashups, based on where downloading, processing,

and generating of the web content takes place [51]. As illustrated in Figure 2.1, server-side mashup's application logic, persistent data storage, as well as accessing different web resources is implemented on the server-side. Client-side mashups (see Figure 2.2) are implemented completely on the client-side so that the processing takes place at the user's web browser. Because of historical reasons, server-side approach has been more popular in the past, but as web browsers have gained more processing power and other capabilities, client-side approach has become common as well. These two types of mashups have their advantages as well as disadvantages, and they suit for different situations. For instance a server-side mashup is not limited by browser's security model, the same origin policy, that isolates documents loaded from distinct origins from each other. Naturally, a hybrid approach combining both server- and client-side mashup techniques is possible as well, and a mashup developer can decide how to partition the functionality between the server and the client, which is the case in [52], for instance.

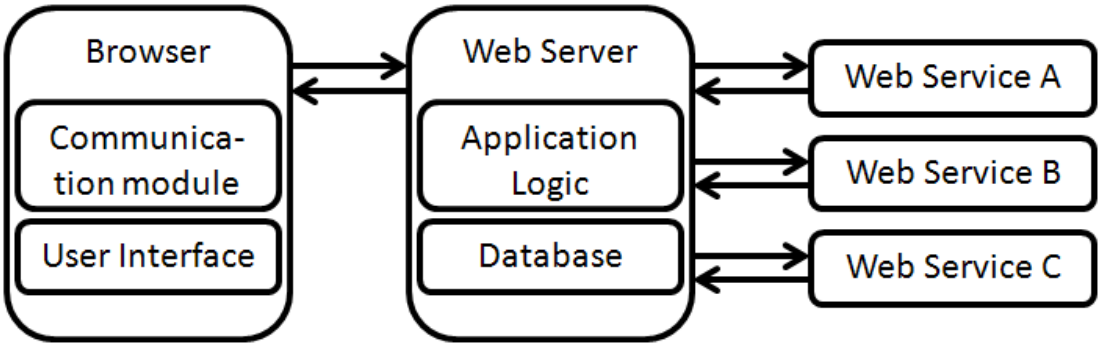


Figure 2.1: Server-side mashup architecture. Adapted from [51].

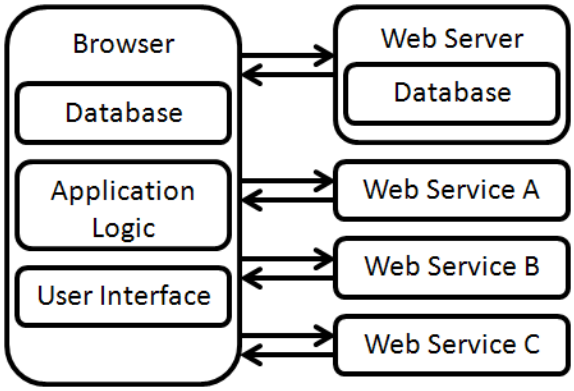


Figure 2.2: Client-side mashup architecture. Adapted from [53].

Infinite Comic (`http://infinitecomic.com/`), created by John Caruso and Paul Covello, is an example of a server-side mashup. It combines Twitter (`http://twitter.com`) messages with Flickr (`http://flickr.com`) images by generating comic strips out of them. The message and image to be combined are chosen by a keyword that user inputs. Screenshot of Infinite Comic mashup when keyword "dog" is used can be seen in Figure 2.3. In this case the application logic as well as image processing takes completely place on the mashup server and only the result comic is loaded to the client.

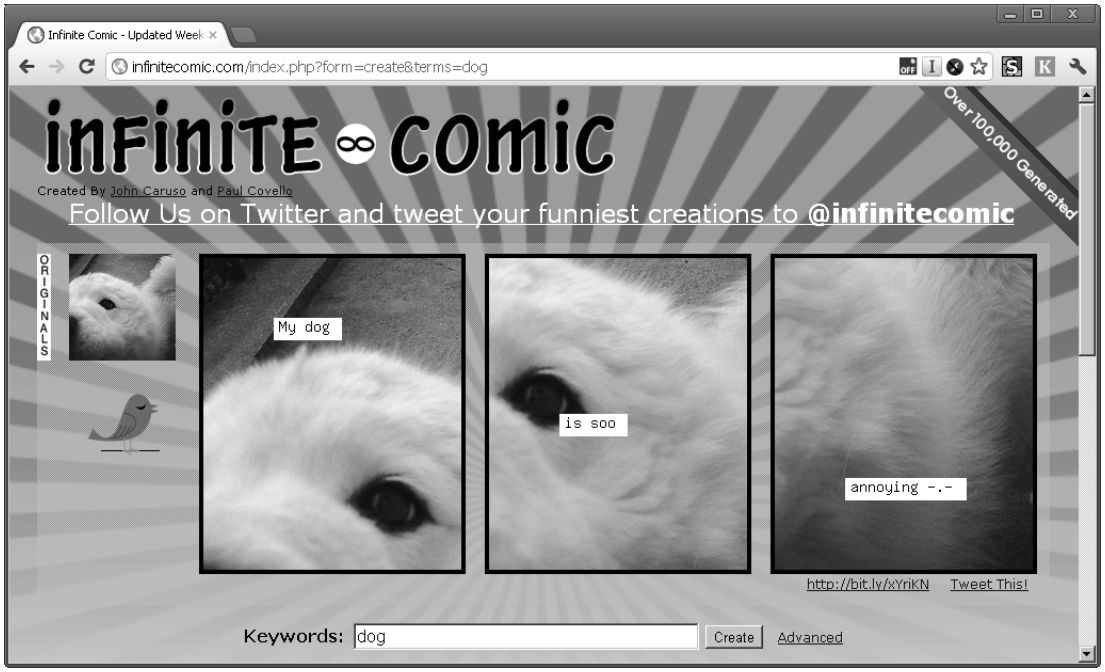


Figure 2.3: Infinite Comic mashup combines Flick images with Twitter messages.

Another example mashup, created by the author of this dissertation, is using the same Twitter and Flickr services as Infinite Comic (see Figure 2.4). However, now the mashup is implemented completely on the client-side. This mashup fetches recent Twitter messages of a user and displays them on a timeline. If a message is clicked, a Flickr image related to a randomly chosen word in the message is displayed. Now the processing and accessing Twitter and Flickr services is implemented directly from the client without a proxy server. In this case, unstandardized technique called JSON [54] with Padding (JSONP) is used to circumvent the same origin policy (see `http://www.json-p.org/`).

Figure 2.4: Mashup combines Flick images with Twitter messages displayed on a timeline.

A mashup called HousingMaps (`http://www.housingmaps.com/`) is a classic example of a mashup that is implemented with both server- and client-side techniques. The mashup, developed by Paul Rademacher, presents apartments that are listed in Graigslist (`http://www.craigslist.org/`) advertisements. Apartments are displayed on a map that is created with Google Maps service (`https://developers.google.com/maps/`). Apartments can be filtered with criteria such as renting or selling price and keywords. The mashup uses hybrid approach as accessing the Graigslist data is done on the mashup server, but Google Maps is used directly from the client.

Multiple and single service mashups. Instead of combining content from multiple services, which is by definition the usual case, some applications referred to as mashups are using only one single service to create a new visualization or user interface for the service. For instance, ProgrammableWeb lists numerous applications accessing data of a single service as "mashups".

Often the user interface of this kind of mashups is simplified and added with some kind of attractive properties. Another kind of single service mashups provide more advanced ways for accessing the service's content. For instance, searching of content can be implemented in a more compelling way than the original service. This is the case

in numerous mashups that show images retrieved from the popular image service Flickr. Another examples of a single service mashups are Lexisum (`http://lexisum.com/`), a web application that creates a printer-friendly summary of a searched Wikipedia article (`http://www.wikipedia.org/`), and WikiMindMap that generates a mindmap about a keyword based on Wikipedia articles (`http://www.wikimindmap.org/`).

Commercial mashups. Commercial mashups are created to generate profit for the mashup publisher where as non-commercial mashups are provided non-profit. Typical example of a commercial mashup combines information about the product being sold with user reviews from multiple sources. Another type of commercial mashups is those including advertisements. Commercial mashups are targeted at consumers in contrast to enterprise mashups that are targeted at business users, even though both are often created by a company. It is common that a commercial mashup is provided for mobile device users as an alternative user interface for an electronic commerce.

Examples of commercial mashups are price comparison and product search mashups. For instance, there are numerous mashups offering this kind of service based on price data of Amazon (`http://www.amazon.com/`) or EBay (`http://www.ebay.com/`). Another kinds of commercial mashups help to locate a certain dealer on a map. Further example of a commercial mashup combining social network services is Scupal (`http://www.scupal.com/`), a social buying website launched in India. Scupal allows users to select a product they would be willing to purchase and then gather other interested buyers of the same product within their social networking contacts. The more there are buyers the less is the price.

Enterprise mashups. Enterprise mashups are developed to solve some particular business-related problem. They can use closed enterprise data sources and combine the information with data from the Web [48]. Enterprise mashups can be created solely by the company's IT department or a sandbox environment may be provided for non-experts to create mashups [55]. However, the more degree of freedom is allowed, the greater are the skills needed for mashup development. Often enterprise mashups are used to provide a simplified view for huge amount of data to assist with decision making [50]. Typical to enterprise mashups is that they focus is on a single presentation and target at providing a tool to help collaboration with different people working on the same project. Typically aesthetic aspects of enterprise mashups are not as important as with consumer mashups. Consumer mashups are composed with different public

web resources in contrast to enterprise mashups where own data sources are used as well.

Enterprise mashup markup language (EMML) [56] is a XML-based domain specific language for developing enterprise mashups. It is developed by the Open Mashup Alliance (OMA), a consortium dedicated to the successful use of enterprise mashup technologies, and provided for free under Creative Commons Attribution No Derivatives (CC BY-ND 3.0) license. With EMML, OMA aims at introducing a standardized, consistent, and interoperable way to develop enterprise mashups. In addition to defining the language, OMA provides a reference implementation of a runtime that processes mashup scripts written in EMML.

EMML can be used to declaratively describe the data processing flow of a mashup. It includes six types of operations: *fetch*, *mash*, *enrich*, *control*, *database* and *input/output* (IO). Operations are listed in Table 2.1. Typically XPath (XML Path Language) [57, 58] expressions are used in EMML to work with the results and variables in mashup scripts. However, other scripting languages such as JavaScript, Groovy, JRuby, and XQuery can be used as well.

Table 2.1: EMML operations [56].

Fetch	**Mash**	**Enrich**	**Control**	**Database**	**IO**
directinvoke	filter	append	if, else	sql	input
invoke	sort	construct	while	sqlupdate	output
	group	annotate	for	sqlBeginTransaction	display
	join	assign	foreach	sqlCommit	variable
	merge		parallel	sqlRollback	
	select		sequence	datasource	

Situational mashups. Term *situational application* is used about an application that is created for a narrow group of users with unique needs. In Clay Shirkys essay Situated software [59] this type of applications are described to be "designed for use by a specific social group, rather than for a generic set of 'users' ". Typically situational applications have short life span and the quality of engineering may not be first class. In addition, scaling up is often difficult with situational applications. However, Shirky remarked that as the group of users is relatively small, it is often unnecessary to implement mechanisms for user supervision. Furthermore, situational applications

are typically more personalized, and they can contain pre-entered information that is relevant only for the small group of intended users.

Some mashups are developed as situational applications [50]. As simple mashups that utilize readily available interfaces can be composed together rather quickly, the cost of implementation is relatively low. Therefore, mashups can be targeted at small, specific groups of users and be very personalized. The architecture and other engineering aspects of this kind of mashups may not be the most polished, but with the specific target group and purpose, it does not have resonance. One should bear in mind, however, that when mashups are used to address non-trivial, more complicated issues, this approach should not be used as it quickly leads to difficulties.

The most essential service. One way to establish a classification is to use the type of the most essential service as a basis to determine the mashup type. For instance, a mashup can be classified as social, news, map, image, video, audio, or search mashup in accordance to the main service it utilizes. Mashup statistics divided into categories based on the essential service used in a mashup can be collected from ProgrammableWeb, which is a web site providing statistics about consumer mashups as well as service interfaces used to create new mashups. Only those mashups that are submitted to the service are listed, and the site does not list enterprise mashups at all. However, ProgrammableWeb can be used as a source for suggestive information about consumer mashups. As can be seen in Figure 2.5, mapping mashups are the most popular type of mashups. Search, social, photo, shopping, and video mashups are roughly equally popular.

2.4 Web Interfaces

Mashups are build on one or more interfaces exposed over the Web. Interfaces can be used to access resources such as images, videos, audio, and texts. Other type of interfaces can be used to perform some operations, such as sending a message, transcoding data from a format to another, creating automatic metadata, identifying image contents, searching for arbitrary content, visualizing content, translating text, recognizing speech, transferring money, and shopping for products, among others. Furthermore, web interfaces can be used to request application components. For instance, the most popular web interface, Google Maps, can be used to create a rich map widget into a

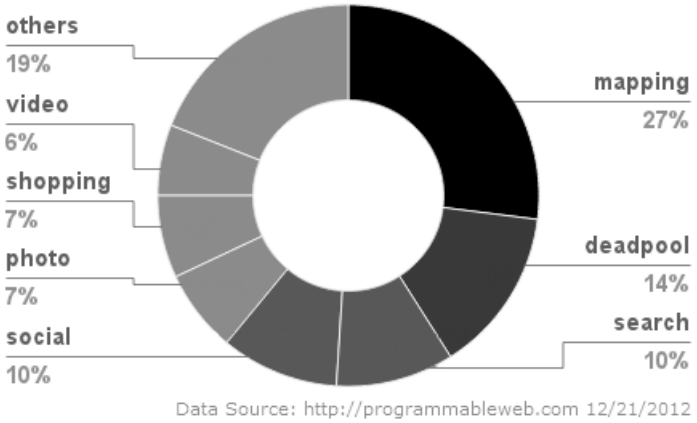


Figure 2.5: Mashup types according to ProgrammableWeb (`http://www.programmableweb.com/mashups`). 'Deadpool' refers to discontinued mashups.

web application. There are numerous web interfaces available for user interface widgets, security features (e.g. user authentication or CAPTCHA services), database accessing, and maintenance (e.g. automatic application error notification), etc. According to statistics at ProgrammableWeb (see Figure 2.6) Google Maps service is the most popular interface used in mashupping. Twitter, YouTube (`http://www.youtube.com/`), and Flickr are the next ones, with roughly equal popularity.

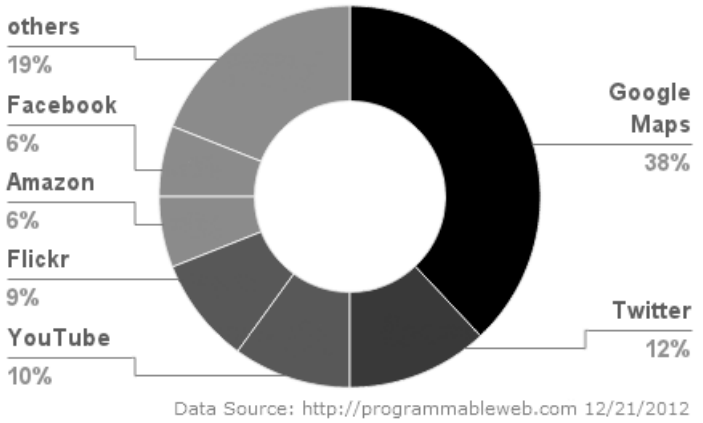


Figure 2.6: Web interfaces used in mashups according to ProgrammableWeb (`http://www.programmableweb.com/apis`).

The rationale behind exposing services to be used in arbitrary web applications can vary. Sometimes service providers charge users or the application publishers according

to the usage of the service's Application Programming Interface (API). However, more commonly a service is provided for free, and sometimes the developers are even paid for using an API in an application. Services provided for free can include banners or other types of advertisement. However, it is common that instead of advertising, service provider is interested in expanding the reach of the service and making their product easy to access via different routes.

Web interface legal terms and conditions are diverse. Commonly service providers set restrictions for those uploading content to the service, as well as those utilizing service's content through the API, including mashup developers. In the following, some typical requirements and terms that affect mashup development are explained.

- *Service-level Agreements* (SLAs) are used to provide uptime guarantee or to state that the service has no liability for downtime or unexpected changes. Sometimes the latter is available for those who use the interface for free, and the former for paying customers.

- If the interface allows accessing user created content under different licenses, terms of service require developers to strictly follow those licenses. If an application uses a cache, also the cache needs to reflect changes in content's licenses and availability. Sometimes service terms determine time limits for the cache reflecting these changes. Moreover, caching may be forbidden completely.

- If the interface enables accessing user's private data, terms of service can include restrictions about how this data can be used and stored.

- Service provider's logo or other branding may required to be explicitly available in the mashup. Other services require adding acknowledgements to application source code. Detailed terms on how the branding is presented may be set. For instance, Google Maps service requires that the Google logo should not be the largest logo in an application implementation, except as displayed in the map image itself (`https://developers.google.com/maps/terms`).

- Interface use rate can be limited to a certain amount of requests in a time period. For instance, Twitter limits unauthenticated calls to 150 requests per hour and authorized calls to 350 requests per hour (`https://dev.twitter.com/docs/rate-limiting`).

- Certain types of applications may be prohibited. For instance, Flickr terms of service deny using Flickr API for any application that replicates or attempts to replace the essential user experience of Flickr.com (`http://www.flickr.com/services/api/tos/`).

- Repeated violations of interface terms, for instance exceeding of use rates or using the API in forbidden type of application, may result service provider to terminate certain application from accessing the interface. Technically this can be achieved restricting application IP or application specific API key from accessing the service. Terms of service often contain a clause for such situation.

Web interfaces usually follow some common interface design style. Recently, interfaces following REST (representational state transfer) architectural style have become popular, but numerous other styles such as SOAP (simple object access protocol) and other RPC (remote procedure call) interfaces are still common. In addition, web services can provide programmatic JavaScript interfaces that are relying on libraries used with regular JavaScript function calls. In the following these main web interface styles are described in more detail.

REST interfaces. Server-client interface architecture based on representational state transfers (commonly referred to as RESTful services or interfaces [60]) was introduced by Roy Fielding in his doctoral dissertation [61]. Design goals of the architecture are scalability, generality of interfaces, independent deployment of resources, and allowing intermediary components. In a RESTful service, *resources* are referenced with global identifiers, and in order to manipulate these resources, *components* of the network (clients and servers) communicate using a standardized interface. Using the interface, components exchange representations of resources. For instance, a resource representing a user may return a representation that contains user's unique identification number, username, real name and age. Listing 2.1 is an example of a request to a RESTful service and a response containing representation of two users.

To achieve generality of interfaces in REST architecture, only a set of well-known standard operations is allowed. In the case of HTTP these operations are GET, POST, PUT, and DELETE, commonly referred to as CRUD (create, retrieve, update, and delete) [62]. Resources in a RESTful service are accessed through unambiguous self-descriptive addresses, which in the case of a web service are URIs, such as

Listing 2.1: Example of requesting a resource with REST interface.

```
GET /users?username=test HTTP/1.1
Host: www.example.com
Accept: application/javascript


HTTP/1.1 200 OK
Content-Type: application/javascript
Expires: Wed, 16 Jan 2013 16:00:00 GMT

[
    {
        "id":1,
        "username": "testuser1",
        "realname": "John Smith",
        "age": 42,
        "groups": [
            { "ref": "../groups/1" },
            { "ref": "../groups/4" }
        ]
    },
    {
        "id":2,
        "username": "testuser2",
        "realname": "John Doe",
        "age": 24,
        "groups": [
            { "ref": "../groups/1" }
        ]
    }
]
```

`http://www.example.com/users/1` in Listing 2.1. Moreover, resource representations contain references to other resources, for instance in Listing 2.1 users are linked to groups.

RESTful web interfaces often make use of Internet media types (originally called MIME types [63]) as well as HTTP request and response codes. In Listing 2.1 the response media type is `application/javascript`, and the HTTP status code is `200`, which means that the request was successful. Typically resources are served in CSV (Comma-Separated Values), XML, or JSON format, and often the response format can be determined in the request. In Listing 2.1 the request header `Accept` determines the response format as `application/javascript` and consequently a JSON formatted document is returned.

In RESTful services the state of a resource is preserved on the server, and the client-side is responsible of preserving the state of an application. However, to enhance scalability, the REST protocol itself is stateless, as the web server does not preserve state of the client. Therefore, if other clients need to be aware of a client state, the state needs to be transferred between them instead of storing it to the server.

Other constraints of RESTful services are cacheability and possibility to add transparent intermediaries (layers) between the client and the server. Both these constraints increase scalability of the system. Because of requirement for cacheability, resources need to define themselves as cacheable or non-cacheable to prevent clients from misusing resources. HTTP response may also contain `Expires` field that tells the client when the resource is going to expire. The ability to add transparent layers to the system makes possible to increase scalability of the system by adding load balancing servers and shared caches.

RPC interfaces. The first RPC (Remote Procedure Call) interfaces were introduced in 1970s [64], and today there are many incompatible implementations of RPC protocol. By using RPC messages, a client sends a request to a server, which executes specified procedure and sends the result back to the client. Calls to the server may be synchronous or asynchronous. With RPC model it is possible to use remote services with familiar-looking function or method calls with parameters that the server expects. Often a proxy can be used at client- and server-side to convert a function call to a network message, which makes calling remote resources convenient for developers.

RPC-based systems, however, lack support for caching. Therefore, RPC fits poorly for some large-scale systems, as caching is often critical for scaling [65].

The most common RPC protocols used in the Web are SOAP, XML-RPC and JSON-RPC. In RPC web interfaces the actual request object is transferred in the message body. Instead of utilizing standard HTTP operations, RPC style interfaces often call interface specific functions or methods directly with HTTP POST messages. Therefore, message overhead is higher with RPC interfaces than with RESTful services. This can be seen in Listing 2.2, which implements the same functionality as Listing 2.1 with a SOAP interface.

Programmatic JavaScript interfaces. Some services offer a programmatic JavaScript interface or an API to access their service. This kind of web interfaces are often targeted at browser-based applications, and typically the interface needs to be reimplemented for other types of runtime environments. Accessing a programmatic API is done by including a JavaScript library in the application, and by calling the library's functions and objects with programmatic JavaScript calls. Using the interface in this way is convenient, as it is similar to using other JavaScript interfaces of the browser.

When a web browser acts as a runtime environment, using a JavaScript library is done with a `script` element, which is included to the web application either statically in the HTML markup or dynamically with a JavaScript call. A library can reside on the same server as the application or on the service provider's server, as the `script` element does not enforce same origin policy, which allows to download the library from a foreign origin. If the service provider requires using an API key for identifying the application, the key is typically included as a parameter in the `script` element's attribute specifying the location of the library.

Interface examples. At the time of writing, Google Maps API family contains five different interfaces: Maps JavaScript API, Maps API for Flash, Google Earth API, Static Maps API and Web Services API. The Maps JavaScript API is the one used to create most map-based mashups, and an example of using this API is shown in Listing 2.3. To be able to use the interface a JavaScript file containing Google Maps API library functions needs to be added into the HTML document with a `script` element. Google Maps API requires using an API key, unique string that is passed to the service to identify applications and attach them to domain names. In addition, a `div` element needs to be added for the Google Maps API to render on. Now all API functions can be

Listing 2.2: Example of requesting a resource with SOAP interface.

```
GET /users HTTP/1.1
Host: www.example.com

<?xml version="1.0"?>
<soap:Envelope
  xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
  xmlns:m="http://www.example.com/users">
  <soap:Header>
    <m:DeveloperKey>1234</t>
  </soap:Header>
  <soap:Body>
    <m:GetUserData>
      <m:UserName>test</m:UserName>
    </m:GetUserData>
  </soap:Body>
</soap:Envelope>


HTTP/1.1 200 OK

<?xml version="1.0"?>
<soap:Envelope
  xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
  xmlns:m="http://www.example.com/users">
  <soap:Body>
    <m:GetUserDataResponse>
      <m:User>
        <m:UserId>1</m:UserId>
        <m:UserName>testuser1</m:UserName>
        <m:RealName>John Smith</m:RealName>
        <m:Age>42</m:Age>
      </m:User>
      <m:User>
        <m:UserId>2</m:UserId>
        <m:UserName>testuser2</m:UserName>
        <m:RealName>John Doe</m:RealName>
        <m:Age>24</m:Age>
      </m:User>
    </m:GetUserDataResponse>
  </soap:Body>
</soap:Envelope>
```

Listing 2.3: Example of adding a map component with Google Maps JavaScript API.

```
<script
  type="text/javascript"
  src="http://maps.googleapis.com/maps/api/js?key=API_KEY" >
</script>

<script type="text/javascript">
function initialize() {
    var myOptions = {
        center: new google.maps.LatLng(-34.397, 150.644),
        zoom: 8
    };
    var map = new google.maps.Map(
        document.getElementById("map_canvas"),
        myOptions
    );
}
</script>
<div id="map_canvas" style="width:500px; height:500px" />
```

used with JavaScript function calls. In Listing 2.3 a JavaScript function `initialize()` is defined to add a map component with fixed location and zoom level into the `div` element with ID `map_canvas`.

Flickr, a popular service intended for social image sharing, has different kind of approach for an API. Although it uses an API key similarly to Google Maps, the service is used in different fashion with requests to and responses from the interface. Flickr allows REST, XML-RPC and SOAP formatted requests. Response is also available in multiple formats including REST, XML-RPC, SOAP, JSON and PHP. Response format needs to be explicitly defined in each request.

Listing 2.4 contains an example request that calls method `flickr.photos.search` to search images with tag `cat`. Only one image is requested and the response format is determined to be JSON. Flickr API implicitly wraps the response with a function call to `jsonFlickrApi` function, which is often convenient if the API is used with an application running in a web browser. Even though the Flickr API call looks RESTful, actually REST architectural principles do not allow an explicit method call in requests.

Listing 2.4: Example of requesting a resource with Flickr API.

```
GET /services/rest/?method=flickr.photos.search&
    format=json&tags=cat&per_page=1&api_key=API_KEY HTTP/1.1
Host: www.flickr.com

HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
Cache-Control: private
Content-Type: text/javascript

jsonFlickrApi(
    {
        "photos": {
            "page":1,
            "pages":4736416,
            "perpage":1,
            "total":"4736416",
            "photo":[
                {
                "id":"6961222925",
                "owner":"29503400@N04",
                "secret":"5b064a6536",
                "server":"7037",
                "farm":8,
                "title":"Sleeping Beauty",
                "ispublic":1,
                "isfriend":0,
                "isfamily":0
                }
            ]
        },
        "stat":"ok"
    }
)
```

2.5 Mashup Development

Mashups can be developed with conventional web programming techniques using text editor and environment with debugging capabilities [21]. However, dedicated mashup development tools can be helpful, especially when end-users are creating mashups. Typical target environment for dedicated tools is a web browser. Figure 2.7 presents technology stack adapted from [47, 49], which is used when mashups are built on web services. Most mashup applications include one or more elements from each layer of the stack, but there are mashups that do not use some layers at all. For instance, a client-side mashup can interact with the data directly without a web service as a backend, or a mashup implemented as a library may lack components from the user interface layer. The elements in the stack can be combined freely, for instance a mashup can use HTML content as a source data, have a backend written in Pyhon, and use REST interface to communicate with client applications implemented in JavaScript and native UI widgets. In the following, we provide an overview about manual mashup development and mashup development with dedicated tools with a web browser as a target environment.

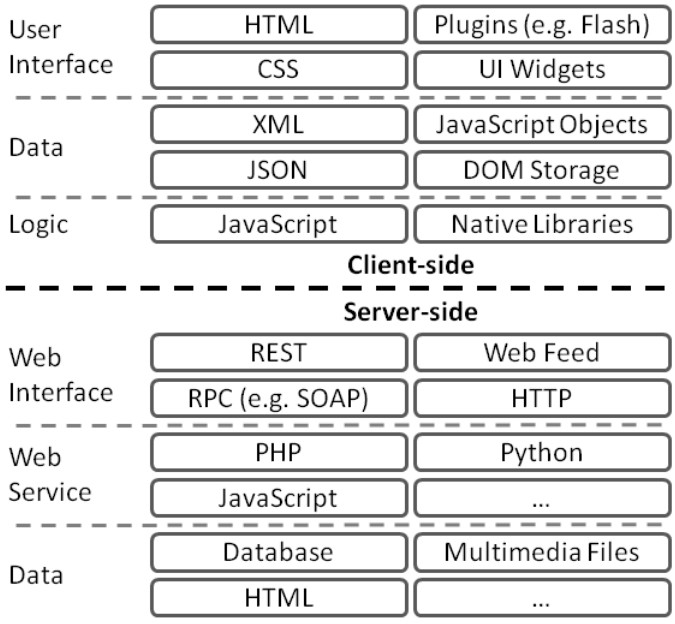


Figure 2.7: Technology stack used in mashups, where stack layers are separated with dotted lines. Adapted from [47, 49].

2.5.1 Manual Mashup Development

Mashup development starts by specifying the core functionality of the mashup. In typical mashup development process, following phases can be identified (similar mashup development scenarios and processes have been explained in [66] and [21]):

- **Selecting data sources.** Finding suitable data sources can be cumbersome as the quality of web services is varying. The selection can be based on multiple criteria such as price, reliability, available request and response formats, availability of documentations, API usage limitations, and other terms of service.

- **Accessing data sources and data handling.** Mashup needs to be able to access the data sources with HTTP interfaces, which can follow different programming models. Moreover, when accessing local resources such as files or device peripherals, interfaces for such services need to be provided by the runtime environment. Furthermore, data needs to be parsed, filtered, and possibly converted into a common format as well as stored into mashup's internal data storage.

- **Creating core mashup functionality.** After the source data is in suitable format, actual mashup creation is usually straightforward. In this phase the actual application logic of the mashup is defined and programmed.

- **Debugging and testing.** Development style of dynamic mashups is typically different from developing traditional static applications. A mashup needs to be constantly tested during the development process, and tests are necessary with different browsers and browser versions.

Client-side web applications can be constructed using a combination of JavaScript, HTML, and CSS. Usually browser-based mashups are built with some JavaScript framework, which can be used when creating the user interface, interfacing with web services, and performing animations. Commonly used JavaScript frameworks include the following (in alphabetical order):

- Dojo (`http://dojotoolkit.org`),
- jQuery (`http://jquery.com`),
- Prototype (`http://www.prototypejs.org`), and
- Scriptaculous (`http://script.aculo.us`).

There are also numerous specialized frameworks targeted at handheld platforms. These include the following:

- jQuery Mobile (`http://jquerymobile.com`),
- jQuery Touch (`http://jqtouch.com`),
- Sencha Touch (`http://www.sencha.com/products/touch`), and
- Titanium Mobile (`http://www.appcelerator.com/platform`).

Some of the frameworks are relatively recent and in beta or alpha stages. Therefore, a developer has to deal with a number of issues related to unexpected behavior of frameworks. Furthermore, documentation quality of the frameworks varies. Some of the frameworks have complete examples and thorough documentation, but others are more or less insufficient in these areas. JavaScript does not inherently support modules, and using two JavaScript frameworks together may lead to namespace conflicts. For instance, some popular libraries use the dollar sign (`$`) as a shortcut syntax for function calls. This has been taken into account in some frameworks, but others are difficult to mix.

2.5.2 Mashup Development with Tools

End-user programming of mashups is a widely researched topic. As mashups by definition add value from the end-user perspective, enabling end-users to compose their own mashups would be a great benefit. However, creating mashups is not straightforward without web development skills. Therefore, number of mashup development tools have been emerging to help end-users to create their own mashups without knowledge about programming languages, data formats, and hosting environments. The most popular mashup tools include (in alphabetical order) IBM Mashup Center, Intel Mash Maker, and Yahoo! Pipes. Numerous tools have been discontinued, and these include products made by major companies such as Google's Mashup Editor and Microsoft's Popfly. Some of these tools have also been evaluated from end-user perspective [67], and Taivalsaari introduces a number of tools in [16]. Academic efforts in developing tools for mashup creation are DashMash [68], Marmite [69], and Vegemite [70]. In principle, any web enabled platform can be used to create mashups. Even spreadsheet applications such as Microsoft Excel and OpenOffice Calc have been used as mashup environment among other table-based systems, and the reader is referred to

[71, 72, 73, 74, 75, 76, 77]. Furthermore, Cao et al. have studied mashup tools from debugging perspective in [78].

IBM Mashup center (`https://greenhouse.lotus.com/`) is a commercial browser-based enterprise solution for presenting and combining data. It has widgets for displaying, requesting, and transforming data. Mashup composing can be done using a visual browser-based tool where widgets are connected to form a system that acquires, handles, transforms, and combines data. Furthermore, the Mashup Center includes a catalog for sharing and discovering mashup components. Components can be tagged, rated, and commented by users. In addition to web resources, local files can be used as source data.

Intel Mash Maker [79] (`http://mashmaker.intel.com/`) is a Firefox browser extension that allows showing information from other web pages related to content of current web page browsed. In contrast to IBM Mashup Center and Yahoo! Pipes, the mashups created with Mash Maker are implemented as client-side mashups. In Mash Maker mashups, information from other pages is shown inside widgets or windows that appear on top of the original content. Therefore, the result is not very integrated and the widgets block the original content. Intel Mash Maker also includes a community maintained database of *extractors* that can be used to extract structured data out of unstructured sources, such as arbitrary web sites.

Yahoo! Pipes (`http://pipes.yahoo.com/pipes/`), shown in Figure 2.8, is another browser-based visual mashup tool that introduces a way to combine data sources via wiring pre-configured modules together. Modules can fetch the data they use from an external source or alternatively take user filled text and numbers as input data. Plug-and-play modules for popular services such as Flickr and Yahoo! Local (`http://local.yahoo.com/`) are available to create a direct hook-up to the service. Mashups created with Yahoo! Pipes are hosted on the Yahoo! servers. In a research by Stolee et al. [80], different ways to refactor mashups created with Yahoo! Pipes are identified and discussed.

As for more academic tools, DashMash is a tool developed by Cappiello et al. [68] for composing server-side enterprise mashups. DashMash has a visual editor to compose the mashup out of components that can be used to retrieve, filter, and render content. In the visual editor, components are added to the system by dragging them from a component menu into mashup workspace. The system includes a recommendation

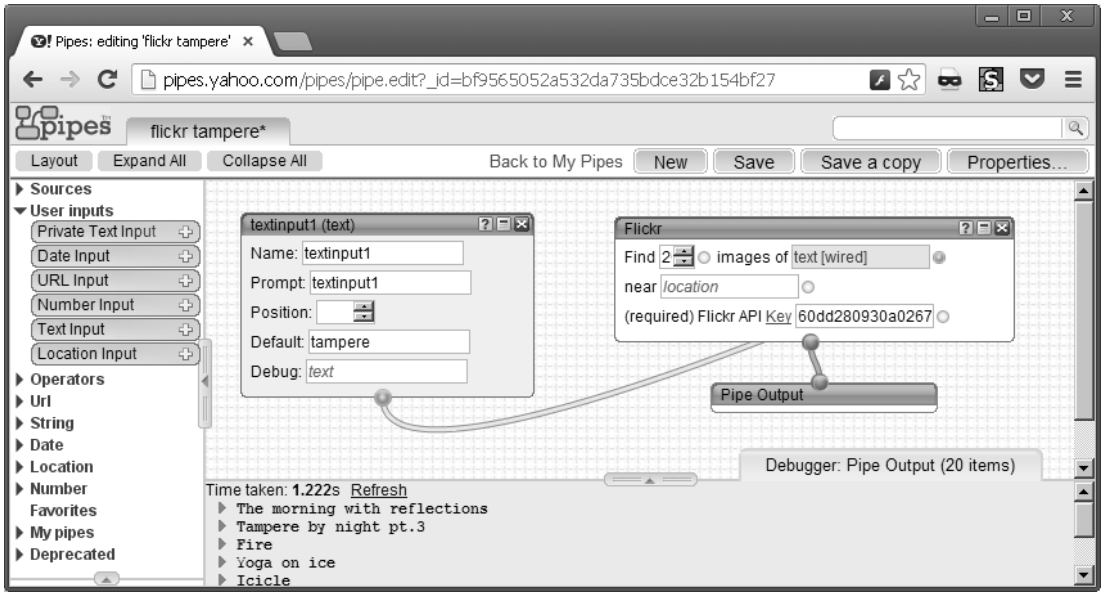


Figure 2.8: Yahoo! Pipes example Pipe fetches Flickr images around user specified location.

mechanisms that tries to help users composing their mashups. In addition to the ability to determine custom bindings, components have predefined default bindings that determine how the components work in co-operation. Default bindings ensure that some functionality is achieved just by adding components into the mashup composition.

Marmite is an end-user programming tool for creating mashups [69]. The system is implemented as a Firefox browser extension. Marmite supports extracting content from web pages, processing it in data flow manner, and combining it with other content. The result can be map, text file, or web page. The programming model of Marmite is similar to Yahoo! Pipes, and extracting, manipulating, and displaying data is done by connecting ready-made modules together and by determining different parameters for these modules.

Vegemite is another academic effort to enable end-users to create mashups [70]. It is an extension for CoScripter end-user programming system [81], which is implemented as a Firefox browser extension. The programming model of Vegemite is based on a data table that is augmented and manipulated with CoScripter scripts. However, in the user study it was found out that using scripts to determine the mashup functionality was very challenging for users without programming experience [70].

2.5.3 Mashup Systems for Cross-Domain Communications

Some research efforts have been focusing on creating mashup frameworks to ensure mashup security while accessing resources in separate origins. Some of these frameworks are targeted at unmodified web browsers, while others require a modified browser or browser extensions to be used.

CompoWeb [82] is a system where mashups can be composed of gadgets that are isolated from each other. Gadgets are developed with a combination of HTML and JavaScript. The system defines a new HTML tag `<gadget>`, three new HTML meta types, and several global JavaScript objects and functions. The system is requires two browser extensions to be able to support the customized HTML.

To create a fine-grained solution for mashup security, Crockford has proposed introducing a `<module>` tag to be able to partition a web page into a collection of modules [83]. The communication between modules is permitted only through cooperating send and receive functions, and it is restricted only to JSON text. In addition, JavaScript code execution is sandboxed into the context of modules. Support for modules can be added to some browsers with an extension.

MashupOS [84] is another system where new HTML elements with different types of security levels are used to enable secure cross-domain communication. In addition, message passing is enabled with JavaScript objects, but the communication is restricted to "data-only objects" such as numbers or strings, or collections of these objects. The system is implemented to work with a customized web browser.

OMash [85] uses security levels of MashupOS as a starting point and provides a simplified version of them. OMash uses object abstractions to enable secure mashups, as is treats web documents as objects and allows documents to communicate only through their declared public interfaces. OMash can be used only with a customized web browser.

Subspace [86] is a client-side system for enabling secure mashups, and it works with unmodified web browsers as the implementation is built on `iframe` elements, which are referred to as *frames*. Frames in Subspace have different domains, but as the domain attribute of a frame can be relaxed, JavaScript message objects can be passed between frames. Consequently, data from one domain can be passed to a frame of different origin.

SMash [87] is similar high-level communication framework to mashups as Subspace, and it can be used in an unmodified browser as well. In SMash a web application is divided to components, which are implemented as `iframe` elements. SMash enables communication between the components with a channel implemented using URI fragment identifiers [88], and unlike Subspace, dynamic domain changes are not necessary.

Another security related study is a lattice-based mashup security model by Magazinius et al. [89]. The security lattice is build from the origins of mashup components so that the each level of the lattice corresponds to a set of origins. To allow a controlled release of information between mashup components, a declassification mechanism is proposed. Declassification policy defines what pieces of information can be shared between components, so that sharing between components on the same level of the lattice is unrestricted, but limited between other levels. Sharing data from security level to another can only be done if it is explicitly allowed.

Ikeda et al. [90] have designed a framework for creating flexible mashups in which the user can selectively browse trough mashup items. The framework includes data management engine for on-demand data generation and GUI widgets that can be used to browse the data. These are both implemented on client-side as well as connections to different web services. On the server-side the framework provides only configuration files for widgets and data management.

Service access control API that aims to better mashup security has been studied by Hashimoto et al. [91]. The SAXAE API provides functions for the mashup to retrieve protected, non-public resources. This allows the mashup to access users private data, for example on a social service, in secure fashion.

2.5.4 Mashup Patterns

Design patterns in software industry are general reusable solutions to some commonly occurring problem. Patterns describe how a certain type of problem can be solved in many different situations. Even though mashups are often tailored to solve a particular problem, there are repeating problems that can be identified. Well-thought mashup patterns can be used as a starting point when solving this kind of issues. Ogrinz classifies mashup patterns into five categories: harvest patterns, enhance patterns, assemble patterns, manage patterns, and test patterns [92]. Based on [92], a brief overview of the pattern categories is provided in the following.

Harvest patterns. Harvest patterns describe methods for extracting information from resources previously viewed as closed. It is common that mashups rely solely on public APIs, but for instance in enterprise context, mashups can be build on closed resources to be able to use them to solve specific business related problems. Harvesting is used to extract data from both structured and unstructured sources. Structured sources can be databases, RSS feeds, XML, and other data streams. Unstructured sources can be web sites, binary files, or free form text. Harvest patterns include mechanisms to navigate to important information and then retrieve it. For instance, Ogrinz identifies two harvest patterns referred to as API Enabler and Infinite Monkeys. The former is used to create REST or other kind of interface for previously closed resources, and the latter is used to extract larger quantities of data to be examined more closely later. Often these two can be combined or executed in parallel.

Enhance patterns. Without regular attention, applications degenerate. This happens inevitably as sooner or later something that the application is build on changes and breaks the application or makes it obsolete. This applies to mashups, too. Enhance patterns make keeping mashups running less time and money consuming. This happens in five ways: extending mashups to a wider audience, fixing bugs without touching the underlying code, making software more user friendly, improving the "findability" of data, and incorporating changing business rules. In contrast to harvesting patterns, enhance patterns show the benefit of leaving data as it is. Enhance patterns have emphasis on changing the interaction model of a mashup to keep it useful.

Assemble patterns. Assemble patterns can be used when developing the core function of mashups: adding resources together from multiple sources to create something new. Often the problem to be solved with assemble patterns is availability of source data in suitable format or converting data from a format to another. Sometimes the problem is the amount of data, which is so excessive that presenting it to the user becomes an issue.

Manage patterns. Data management is often at the core of information technology. Therefore, mechanisms for transmitting and storing data are needed. However, transitioning between data handling systems, condensing data, and securing access to data are important functionalities as well. Manage patterns can be used to include this kind of actions to mashups.

Test patterns. As mashups can be developed by users, there is also an obligation for the users acting as developers to test their mashups at least in some degree. Test patterns describe mechanisms that are easy to understand and implement in mashups.

2.6 Mashup Runtimes

Web browser is not the only execution environment for mashups. In fact, mashups can be executed on other environments that fulfill the following basic requirements, listed originally in [93] and summarized as follows:

- **Access to resources.** A mashup environment should allow liberate access to resources with asynchronous HTTP. Most web interfaces are accessed with protocols that are based on HTTP. Possibility to make asynchronous requests enables accessing web resources in a non-blocking manner. In addition, an environment should have inbuilt support for parsing data in JSON and XML formats. Most web interfaces offer their data in these formats, and it is convenient to be able to parse it automatically.

- **Support for creating user interfaces.** Support for attractive graphics is important for a mashup environment. User's perception of quality is often based on look and feel of the system. Some mashups are solely based on providing the same functionality as the original service, but with more attractive visualization. The environment should provide ready for use widgets and other components that could be used as building blocks for the mashup user interface.

- **Support for dynamic programming.** Environment support for a dynamic programming language is vital when accessing web resources with interfaces that are prone to changes. Even though static languages are traditionally used in systems with limited resources, recent development of language engines and device capabilities has enabled using dynamic languages on embedded devices as well. JavaScript is currently the most widely supported dynamic language, as it has become the industry standard in web browsers. Therefore, using it as the language of choice is often beneficial, because this enables utilizing the same libraries that are available for browser applications in mashups.

- **Access to user's context.** When JavaScript or other language is chosen, it is important to have necessary bindings for accessing device interfaces. These interfaces can include camera, Bluetooth (`http://www.bluetooth.org`), location sensors, and other peripherals. Bindings to such services enable development of context aware mashups.

- **Fine grained security model.** Fine grained security model is necessary when building mashups, and it is justified as a basic requirement as explained in [93]. Web browser's security model, the same origin policy, forces applications to function through a single domain. As mashups access multiple web interfaces on different domains, developers use proxies or other means to circumvent the same origin policy, effectively making the security model ineffectual [94]. More sophisticated security model is clearly necessary, and recent developments by W3C (see `http://www.w3.org/Security/` as well as [95]) are currently addressing the problem.

Environments that fulfill these requirements include different widget platforms and rich internet application (RIA) platforms available for desktop computers. If the target device is not a regular computer, use of a custom runtime environment is even more practical. Many mobile manufacturers have their own web application frameworks and there are cross-platform solutions, as well. In Publication V we introduce two runtime environments that can be used for mashups in mobile devices and describe how it is possible to create mashups by combining dynamic code and native binary libraries. This kind of hybrid approach can combine the flexibility of dynamic code and the performance of binary applications. Important design decisions in this kind of implementation are how the functionality is divided between static and dynamic parts of the application, and how technical issues related to combinations of scripts and binaries are solved.

New web technologies are offering promising building blocks for composing web applications. For example, WebGL [96] is enabling 3D graphics API to be used inside a web browser without plugins, and HTML5 [11] is bringing support for embedded audio and video, cross-document messaging, offline functionality, and local data storing, among others. Applying HTML5 technologies to mashup development has been studied by Aghaee and Pautasso in [97]. Third version of CSS [6] adds numerous ways to

determine the visual appearance of elements in web documents. Various effects that have been previously implemented with undocumented "hacks" can now be described in standardized manner with CSS code. These are remarkable improvements that will gear the web browser into more and more powerful platform for complex applications. From mashup development point of view these tools can enable compositions that are yet completely unforeseen.

3

Mashup Ecosystems

In this Chapter, we consider mashup ecosystems that are formed by users, mashup authors, and service providers, and where mashups and services are connected through web interfaces as has been visualized in Figure 3.1. Here, we have considered only those actors that interact in the ecosystem by producing, altering, or consuming the content. In other words, mashup ecosystems could include other actors as well, including, but not limited to, tool providers, researchers, regulators, and so forth.

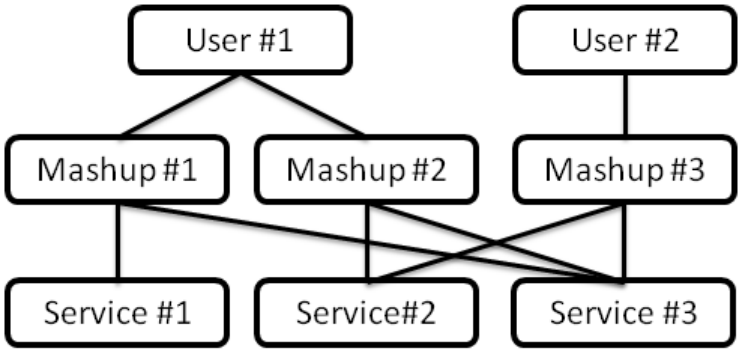


Figure 3.1: Mashup ecosystem built on three services and containing three mashups.

We start by providing background information about mashup ecosystems. Then we describe three perspectives to mashup ecosystems and introduce four levels of support that service providers can offer for mashup developers. In addition, we discuss about characteristics of explicit and implicit mashup ecosystems. Furthermore, we introduce challenges that mashup ecosystems encounter, as well as present our practical implementation of a mobile multimedia mashup ecosystem. The background Section is

mainly based on the literature, while other Sections in this Chapter are based on our own research, if not stated otherwise.

3.1 Background and Related Work

According to definition by Bosch in [98], "A Software Ecosystem consists of the set of software solutions that enable, support and automate the activities and transactions by the actors in the associated social or business ecosystem and the organizations that provide these solutions". Since mashups by definition combine data from multiple sources, the stakeholders that provide this data form an ecosystem [14] – in other words a set of entities that act as a single unit instead of each participating business acting separately. In [98] Bosch reviews mashup ecosystem from end-user programming point of view. Furthermore, the same article poins out two success factors as well as two challenges that this ecosystem has. The two success factors are, first, the value that end-users gain by designing their own applications, and second, sharing of applications among users. The two challenges are enabling the end-user programming for inexperienced developers and minimizing ecosystem maintenance efforts. Moreover, Bosch identifies so called "undirected developers" that are able to use the platform in unforeseen ways and provide significant innovations for the overall ecosystem.

A mashup ecosystem can be considered to include all possible web interfaces and mashups build on top of them, which is the case in Yu's and Woodard's research about mashup ecosystem characteristics [99]. Yu and Woodard describe the mashup ecosystem by using the ProgrammableWeb mashup indexing service's data as the source. They investigate the structure and dynamics of the Web 2.0 ecosystem by analyzing the data available about mashups and APIs. The first finding was that at the time of the study services were organized into three tiers, which were 1) the most popular service (Google Maps), 2) popular services (many services used for social services and searching), and 3) less popular services (services often used for blogging, online retail, music, videos, and feeds). The second finding was that mashups are often composed by combining APIs across tiers. This highlights the central role of the most popular APIs, but also reveals the importance of less popular APIs in dilution of the ecosystem. Many of the third tier APIs bring together novel combinations of functionality. Another interesting finding is that in contrast to what has been suggested in [49], there

is no long tail of services that would form a basis for significant number of mashups. Instead, Yu and Woodart noticed that 95 % of mashups are build on 20 % of services, which is much more than in the famous Pareto's 80-20 rule. Moreover, they noted that 51 % of services were not used by mashups at all. However, one should bear in mind that Yu's and Woodard's data source, ProgrammableWeb, does not list services and mashups other than that have been added to it by developers. Therefore, there are services and mashups that are not included in the source data.

Another interesting topic concerns the way mashup ecosystem grows. For instance, hypothesis in [100] is that mashup developers create new mashups by copying existing ones. Simulations suggest that this would be true, as it is in line with the reports about mashup ecosystem growing [99]. However, the hypothesis of [100] has not been tested empirically.

3.2 Three Perspectives on Mashup Ecosystems

Subsets of the global mashup ecosystem described by Yu and Woodart in [99], i.e. ecosystems where an ecosystem provider determines the available interfaces, tools, and platforms that can be used to build mashups, can be considered as well. The ecosystem provider may be authoring either the services or the mashup, or both. Some ecosystems allow end-users to create own mashups on top of the ecosystem services, but this is not always the case. In the following, we describe these smaller mashup ecosystems from three perspectives of the ecosystem actors: mashup user's, mashup author's and service provider's. Mashup author's and service provider's view on mashup ecosystems are originally presented in Publication I, and in pursuit of completeness, we present the view of a user here, as well.

Using a mashup. By definition, mashups have increased value from user's perspective. This value comes from mashup's ability to access numerous services simultaneously and to combine information into a single integrated view [21]. In addition, some mashups may allow to users contribute content into a service. Consequently, typical mashup user desires to gain access to a set of services and looks for a suitable mashup to be executed on a specific device, or, if such option is available, the user can create a mashup by himself using tools that the ecosystem provides. The services that are desired to be used can be closely related, for instance when using a mashup that

accesses multiple instant messaging services. However, there are numerous examples of mashups that access completely different types of services and create something unexpected. For instance, different types of information can be arranged geographically and attached to a map.

User's perception of the mashup ecosystem quality is based almost completely on the look-and-feel of the mashup implementation. This makes user interface issues really important, as, even if the mashup ecosystem would include well-designed services, a poorly implemented mashup that is used to access them may drive potential users away.

Authoring a mashup. One motivation of authoring a mashup is providing a better user experience for users of particular application, for instance a video player mashup that is capable of using video content from multiple services is more compelling than one using just single video service's content. Consequently, to be able to create a mashup that becomes popular, the ecosystem has to provide access to services that are desired to be used [101]. In addition, there is a need to gain access to other services that the users are typically not aware of, but that can be used to create something unexpected and interesting [102]. Furthermore, accessing these services should be as convenient as possible, preferably supported by helpful code libraries or application frameworks.

Moreover, legal issues related to mashup ecosystem are numerous (see `http://blog.programmableweb.com/category/law/`). Service terms are often incompatible and hard to follow in complex mashups. The situation is even more complex when a mashup is hosted on a third party platform, such as mashup tool provider's servers. Mashups can be required to follow some content related limitations as well. For instance, some content can be freely available in U.S. but restricted from accessing in U.K. Some service providers may restrict their interfaces to be used only on desktop and prohibit using them on mobile devices. In addition, libraries and frameworks used in mashups may have conflicting licenses. Furthermore, protecting a client-side mashup from copying is often difficult as the executable code needs to be transferred to the client.

Mashup authors encounter numerous challenges, for instance web service reliability and complexity of integrating high number of services [21]. Web service reliability can be addressed by adding fallback mechanisms, but this strategy will make the implementation more complex. While adding more services to the mashup can be attractive for

users, it makes the implementation more complex and increases vulnerability to service breakouts. Furthermore, client-side device capabilities may be limited and operational expenses can be an issue, especially with mobile devices. These and numerous other mashup authoring challenges are addressed in Chapter 4.

Providing a service for mashups. The rationale behind offering services for mashups can be getting a wider audience for certain platform, product, or content accessed through the service. Moreover, opening a service can lead to numerous clients created by third party developers to emerge on different platforms for diverse user needs. Less common are non-free web services that have some specialized, hard to implement, functionality. Services that are funded with advertisements can push advertising messages into mashups along with the content.

Service providers support mashup ecosystems in four identifiable levels, which are originally presented in Publication I. A brief summary of the four levels of support is given in the following:

1. **No support for mashups.** Some web content authors do not support mashups at all and provide their content solely as regular web documents. This kind of content is still accessible with "screen scraping", but such accessing is typically error prone, and it often is illegitimate. Some services even have implemented technical measures to prevent scraping. Furthermore, even if reusing the content in mashups would be allowed, the web content author does not have control on what parts of the content is reused, and it is difficult to build a business model around such approach towards mashups. In addition, it is likely that accessing the content is very inefficient and cumbersome from mashup author's point of view.

2. **Access through a web feed.** It is common that regularly updated sites, such as blogs or news sites, provide their content through RSS, Atom, or other type of web feed. A web feed is easy to set up and maintain, particularly if a publishing system is used. The feed is intended mainly for users to subscribe with a feed reader application, but at the same time the data becomes accessible for mashups, too. Control over the content is still rather coarse, and use cases of web feeds are limited to accessing the content as a whole. Utilizing web feeds in mashups is typically straightforward as helpful libraries and tools for such task

are available on most platforms. Some dedicated mashup tools support only web feeds if content from an arbitrary service is desired to be included into a mashup.

3. **Access through a web interface.** Providing a service with a web interface, typically following either REST or RPC architecture style, enables using the service in mashups. Use cases of such interface allows not just data accessing but other types of services as well. For instance, a service can provide means for social communication, authentication, database accessing, or specialized functions such as reverse geocoding or music identifying. Setting up a web service with REST or RPC interface requires careful planning and implementation, especially when sensitive data is handled. However, such system allows fine-grained control over the content as well as applications using the interface, and it enables different kinds of business models. Service load can be handled as well by limiting requests made in a time period, even individually for each application.

4. **Access through a programmatic interface.** Establishing a programmatic JavaScript API allows to integrate the service tightly with arbitrary web applications and mashup ecosystems. Such interface is used by including a JavaScript library into the application, which makes it possible to use the service with regular JavaScript function calls. Typically the JavaScript library is downloaded from the service provider's server instead of having a copy on the server hosting the mashup, which makes possible to always use the most recent version of the library. Setting up a programmatic JavaScript interface requires careful engineering, but it enables superior control over the content and applications. Diverse business models are possible, and the content can be provided with different terms and licenses for individual clients. Considerable downside of the programmatic interfaces is that, because of technical reasons, numerous versions of the interface need to be provided in parallel. Consequently, bug fixes need to be performed on all the versions, which makes maintaining the interface more laborious. Another downside is that if a programmatic interface is desired to be used on other runtime environments than a web browser, more parallel versions need to be provided.

Authors of [102] point out that when a new service is introduced, it benefits from having a similar API as existing APIs that could possibly form an ecosystem together

with the new service. In addition, authors note that "complementing the existing API, the new API will also benefit the provider of the existing API by providing additional contexts of use for the API and increasing its potential share of mashups that use it" [102]. In other words, when the potential client applications of a service are mashups, the API design should reflect other services that could be combined with the new service.

Until recently most of the services have been provided for free, with the exception of some very specialized services such as image content recognition services. However, in October 2011 Google announced that the most popular mashup service, Google Maps API will be provided in two different versions: free and non-free one called Google Maps for Business. The one with a prize tag provides more advantageous features such as higher request limitations and technical support. Even thought this is the first example of this kind of development, it is an interesting change and may be a beginning of new kind of business model to be common.

3.3 Explicit and Implicit Mashup Ecosystems

Mashup ecosystem does not need to be controlled by a central authority. In contrast, a mashup ecosystem can be formed implicitly without limitations for services or content utilized in mashups. For instance, one can build a mashup on top of services freely available in the Web with liberal enough licenses. In a broad sense, any web document author can be considered as a service provider, as it is common that content is gathered from web sites by technique called "screen scraping" or "web scraping", where the source data is parsed from HTML pages aimed at human readers. The possibility to create implicit mashup ecosystems makes mashups attractive, as it allows to create unforeseen solutions that can even exceed the user's expectations. Furthermore, mashup ecosystems can be targeted at specific platforms, for instance mobile devices with certain operating system. In the following, we summarize our categorization of mashup ecosystems, originally presented in Publication I.

Mashup ecosystems can be categorized roughly to explicit and implicit ecosystems. However, an ecosystem having both types of interactions is possible as well. We define these two types of ecosystems as follows. In an explicit ecosystem it is not possible to add arbitrary services into a mashup, whereas in an implicit ecosystem adding and

removing interfaces is not restricted. In general, explicit mashup ecosystems are based on specific contracts between ecosystem actors, and implicit ecosystems are formed in more ad hoc manner.

Commercial and enterprise mashup ecosystems can often be regarded as explicit [49]. This kind of ecosystems are often closed because of high security, reliability, and availability requirements. Commercial ecosystems are created to generate profit for mashup authors or service providers, and to be able to utilize such system in a stabilized business, mashup authors and service providers coordinate and use either specific contracts or terms of service agreements. Enterprise mashups are typically explicit as well. Because of enterprise mashups rely more on closed enterprise data sources, the agreements between service providers and mashup developers can be company's internal interface specifications in addition to legislation [103]. In general, explicit ecosystems typically require more attention on engineering of the implementation as well as legal matters.

Most other kind of mashup ecosystems can be regarded more or less as implicit. Often these mashups are targeted at consumers and provided for free, and therefore the requirements set for such ecosystems are not as high as in commercial or enterprise setting. In implicit ecosystems even "screen scraping" may be used to gain access to specific content. The most extreme type of implicit ecosystems are those built around situational mashups that highly rely on the fact that the target group of users is very limited [59]. Simple mashups that utilize readily available interfaces can be composed together rather quickly, and consequently the cost of implementation is relatively low. This kind of implicit ecosystem may have rather lightweight security, moderation, and authentication features, and the architecture and other engineering aspects of the mashups may not be the most polished. Implicit mashup ecosystems can emerge swiftly, but sometimes lifespans of such ecosystems are shorter, too.

Sometimes an ecosystem is build around relatively few central services – often just one or two – and, in addition to those, it can contain both implicit and explicit interactions with secondary services. If this is the case, the ecosystem often has an interface that can be used to add arbitrary services to the ecosystem. Typically, mashup tools have such possibilities. For instance, Yahoo! Pipes allows to add arbitrary data sources and combine the data with predefined services that are supported by the tool.

Another example of a system including explicit and implicit parts is Google Maps service, where support for Weather Channel (`http://www.weather.com/`) and Panoramio (`http://www.panoramio.com/`) services is in-built with the map service, and arbitrary services can be included into the ecosystem without restrictions with JavaScript programming.

3.4 Implementing Mashup Ecosystems

Mashup ecosystem implementations are highly dependent on the application domain. For instance, an open ecosystem targeted at desktop clients forms a very different kind of ecosystem compared to a closed one targeted at mobile devices. In addition to choosing whether or not the ecosystem is expandable with arbitrary services, and what kind of devices it will be used on, one has to determine numerous other details as well. For instance, security and issues related to legislation are typically areas where attention is needed. Another fundamental decision is whether to implemented mashup logic on the server-side or the client-side. In the following, we summarize these considerations, originally presented in Publications I, II, and III.

Services in a mashup ecosystem need to provide clearly defined interfaces for mashups to utilize. The interfaces should be standardized web interfaces or programmatic JavaScript libraries to allow different actors to implement compatible subsystems. When services are implemented, portability needs to be taken into account. For instance, while web interfaces following REST or RPC style can be typically used on most web enabled application environments, programmatic JavaScript APIs may be targeted at specific runtime environments.

Security of services provided in a mashup ecosystem should be considered carefully. Security model should be liberating enough to allow access to different services and user's data but eliminate possibility to misuse the system. The model should be easy to understand and implement without adding unnecessary complexity to mashup development. There are numerous existing systems for user authentication, for instance OAuth [104] is a widely used open service for secure interface authentication. In addition, there are numerous research efforts that try to solve mashup authentication issues (see [105, 106, 107]), but applying them still requires careful evaluation.

Legislation is another fundamental challenge of mashup ecosystems. Often the content in mashups is created by different third parties and accessed through services with varying terms and licenses. Content within a service might be provided with different licenses, for instance Flickr image hosting service API can be used to access both images marked with a Creative Commons license and with "all rights reserved" notices. In addition, some services are provided with stricter terms when used in commercial purposes. For instance, a separate agreement and charges may be applied when a service is used in commercial setting, even when the same service is provided for free for non-commercial use.

Targeting a mashup ecosystem for mobile devices sets restrictions for the client-side mashup implementation and may affect on service implementation as well, as explained in Publications II and IV. The impact on the client-side is more significant, as mobile devices do not necessarily have the same runtime environments available for mashups as desktop systems, and sometimes separate versions of the mashup need to be implemented for mobile devices with different operating systems. In addition, rich mashups typically require a reliable network connection, which is not always available in mobile devices. Therefore, support for offline operation is sometimes included into the mashup. Furthermore, the user of mobile mashup ecosystem is typically required to have an unlimited data plan to avoid unexpected operational expenses.

As described in Section 2.3, mashups can be implemented as client- or server-side applications, depending on where handling and combining of the content takes place. Mashup ecosystems can be built around both types of mashups. In addition, a hybrid model, which combines both approaches, is possible. For instance, a mashup ecosystem can consist of a server-side implementation that aggregates different services, and client-side mashup that utilizes this aggregator server.

In addition to issues mentioned above, mashup ecosystems face also other difficulties, such as addressability of mashup ecosystem endpoints as well as transparency and protocol support of the network used. Before IPv6 [108] gains ground it might be necessary to use higher level methods to address network endpoints. Addressing mashup clients and services in too high level can derive scalability and performance issues. Another problem is caused by non-transparent network nodes that may cause some parts of the ecosystem become unattainable. Furthermore, lack of protocol support for other than HTTP can cause video and audio streams fail to operate, for instance.

3.5 Example Implementation

To provide a hands-on view of the topic, we briefly describe an implementation of a mobile multimedia mashup ecosystem originally presented in Publication III. The goal of the ecosystem is ambitious. It targets at enabling liberal mashupping of all relevant web content available in the most popular Web 2.0 multimedia services as well as user's own local area network. The architecture is designed to hide service interface complexity and to enable coherent user experience across services. It should be possible to access the mashup content with different mobile client devices including ones with limited capabilities. The client-side mashup should take user preferences, interpreted on the basis of past actions, into account and automatically adapt to user needs.

In order to address the ecosystem design goals described above, we have chosen to follow a hybrid approach that consists of server- and client-side parts. The architecture is presented in Figure 3.2, where the server-side is referred to as *Mashup Aggregator Server* and the client-side mashup implementation as *Mashup Client*. This hybrid approach enables to run the mashup even on devices which do not support all multimedia technologies. For instance, most of the video content in the Web is available in Flash video format (`http://www.adobe.com/devnet/f4v.html`), which is supported poorly in some mobile devices. Furthermore, clients that are able to take user context into account can be implemented using our approach without need for extensive data transferring with the server.

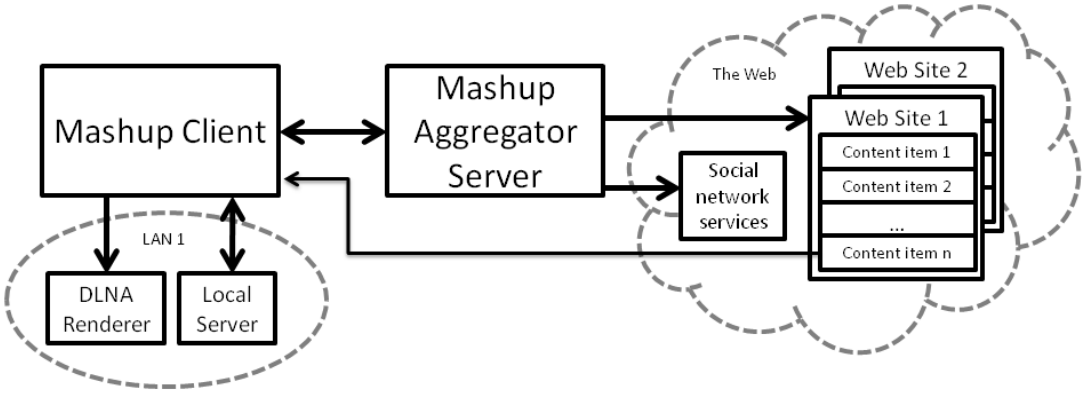


Figure 3.2: Example architecture of a mashup ecosystem. The figure is simplified from the one appearing originally in Publication III.

The Mashup Aggregator Server is implemented as a cloud service. It provides a uniform interface that client-side mashups can use to present videos from multiple sources. The aggregator server provides mechanisms for searching content across services and authenticating users. The backend provides content with different qualities to ensure that devices with low bandwidth network connection can access it. In addition, it connects to different social networking services and can publish public content in those, too.

The Mashup Client, installed into user's mobile terminal, composes the mashup from content items. It communicates with the aggregator server and gains access directly to the web content. In addition to aggregated content, it can browse content from local servers and use it as well. The third source of content are servers in other domains, provided that the Mashup Aggregator Server has necessary information about them. Naturally, if the client has access to local filesystem, local files can be used. If a network connection is not available, the mashup can use only local video content. In the current implementation the client can use any DLNA compatible renderer to play mashupped videos. The client is targeted at mobile devices with Android operating system, and it is programmed with Java.

Interfaces. The ecosystem interface between the aggregator server and mashup clients relies loosely on REST principles [61], and it could be categorized as a REST-RPC hybrid architecture [60]. The interface supports data exchange in both XML and JSON formats, and the format can be selected in the client implementation. This is convenient as parsers for XML are often readily available in runtime environments. However, using data-oriented JSON has certain advantages in contrast to XML which is more document-oriented. One benefit that JSON has is smaller data exchange overhead.

Security. Mashup ecosystem security is based on requiring authentication for clients invariably when a request is made to the aggregator server. We used HTTP digest access authentication [109], which is relatively light-weight solution. This authentication method is less secure when compared to strong authentication protocols such as public-key [110] or Kerberos [111]. However, it is more secure than traditional digest authentication schemes such as basic access authentication [109] or CRAM-MD5 [112]. Digest authentication encrypts user's password and leaves content of a message uncrypted, unlike stronger protocols, which encrypt also the content of the message. This is sufficient solution in basic functionality, which does not require information

about the user. However, when sensitive data is transferred between the client and servers, stronger security mechanisms are necessary to be implemented, as while the HTTP digest access authentication provides a way to negotiate credentials, it does not provide a secure channel to transfer data.

4

Composing Mashups

In this Chapter, we present numerous practical considerations that mashup developers need to take into account. These include architecture, general design principles, interfacing with web and local services, considerations about mashups on mobile devices, and security. Sometimes different kinds of tools targeted at end-users are proposed to make mashup composing easier. Our staring point, however, is that mashups are created without such special purpose tools that often limit the mashup developer's possibilities. Instead, we discuss about how traditional web development tools, methods, and practices can be applied to client-side mashup development along with special web runtime environments, which may be necessary in certain situations where using a regular web browser is not possible.

Mashups can be constructed in numerous ways with a plethora of tools, and there still are major practical problems related to mashup composing and security. For instance, tools introduced are lacking behind, using dynamic languages for large applications is an unknown territory for many developers, and the web browser security model is too restricting for mashups. The field of web programming is constantly changing as new interfaces, technologies, and frameworks build upon novel technologies emerge. The amount of different, constantly evolving APIs with different legal terms is overwhelming. When developing large-scale mashups utilizing numerous web services, the situation turns out even more problematic. Some of the issues are common for both mobile and desktop environment, but naturally mobile mashups have their own specific things to handle as well.

While implementing client-side mashups on mobile terminals sets restrictions on applications and application development, it is a great opportunity from mashup de-

velopment point of view. The dynamic nature of mashups suits well for different ways mobile terminals can be used. Often, the information needed on the fly is related to the user's context, which can be available for applications to access automatically. This opens up opportunities to provide advantageous user experiences, as mashups can dynamically present eligible information – possibly even automatically without requiring specific user action. However, as mobile device capabilities are limited, extending mashups to mobile domain is not trivial, and special solutions are sometimes necessary.

There are situations when composition of a mashup is not possible without use of native code. For example, applications that require a lot of computation power or access to interfaces that are not available for dynamic code have to be constructed with both dynamic and native code. Therefore, offering an interface for mixing web technologies with the capabilities of native software components is sometimes necessary. On the other hand, utilizing hybrid technology allows one to combine the best of both worlds: performance and eye candy of traditional, installed binary applications and pervasiveness and seemingly infinite resources of the Web.

Because of issues and opportunities described above, we first provide an overview about the current state of the art in mashup development. Then we describe how to solve problems related to client-side mashup development in desktop and in mobile devices. We address the research question RQ2 by introducing a reference architecture and a set of general guidelines for mashup development. In addition, we describe two runtime environments that can be used to enable context-aware mashups on embedded devices and point out some considerations about mashups on mobile devices. Finally, we discuss about mashup security, which has a very important role in mashup composing, even though the topic is not addressed in the included Publications. While the background Section and the discussion about mashup security are based on the literature, other Sections are based on our findings described in Publications IV, V, and VI, if not stated otherwise.

4.1 Background and Related Work

Mashup design has been considered opportunistic activity where methods described as "hacking, mashing and gluing" have been used [113]. However, following established software engineering principles in mashup composing is vital for maintainability and

modifiability, as pointed out in [52, 114]. In addition, ability to reuse mashup components requires a well-engineered architecture and careful implementation, as described in [55] and [115].

Composing mashups is different from developing conventional software as pointed out in [16, 45, 116] and summarized in the following.

- Mashup development focuses on reusing the *content*, not the *implementation*. There are numerous widely accepted standards for formatting the content, which makes reusing it in mashups possible. However, reusing mashup implementation – or any other web application's implementation – in another context is difficult. Some mashups manage to reuse the *visual representation* of sites at least partially (e.g. map component), but in others, only the content is reused.

- Mashups are more dynamic than conventional compiled binary software in the sense of accessing content of different types. Because mashups access content that often changes using dynamic interfaces, they can not be built easily using static programming languages with advance compilation and static type checking, as pointed out in [116].

- Mashup developers often do not have formal training or background in software engineering. Those developing mashups may have some kind of media or data mining background. This is another reason, why mashup implementations typically focus on content rather than implementation techniques.

- Because of using the Web as a platform that has great distribution and sharing power, it is easy to implement mashups where reusing content happens in unforeseen scale. In principle, anything that is released to the Web is instantly available for anybody anywhere in the world without limits. Therefore, the party that made a piece of content available may not be aware that it is used in other context as well.

Even though mashup desigers have common goals, architecting mashups is often done by trial-and-error or intuitively. There are few guidelines for developers on mashup architectures, and the design of mashups is regarded as ad hoc acitivity [113]. Moreover, current approaches to mashup development are tool oriented and disregard software engineering principles, such as decomposition and modifiability [16, 45].

Server-side enterprise mashup architecture has been studied by Lopez et al. [117]. Their architecture consists of four layers: *source access*, *data mashup*, *widgets*, and *widget assembly*. In addition, the architecture includes *common services*, which provides general functionalities and can be used from any layer. The result mashup developed using this architecture is somewhat similar to a web portal with the exception that the widgets are connected.

In [118] Bader et al. identify numerous security aspects of enterprise mashup architectures. They point out that mashups handling sensitive data of an organization, should store data strictly on organization's own storage services as using an external storage service in a cloud might cause unwanted information disclosure. One should bear in mind that even non-sensitive data can expose delicate information when aggregated. Moreover, mashup robustness and stability are especially important in an enterprise setting. Therefore, SOA-based approach to creating mashups is recommended. Downside of SOA-based approach is complexity for unexperienced users that may not be able to build mashups on top of a complex SOA stack. Trustworthiness of mashups is significant as well, when they are used in a business setting. Unfortunately, when mashups utilize external web resources, confirming resource trustworthiness is typically difficult or even impossible. Therefore, authors of [118] suggest utilizing a central service governance, which would control services that are used through mashups. Furthermore, when mashups are used to access organization's private data, authentication of mashup users and restricting unauthorized users from accessing sensitive data is vital.

4.2 Designing Mashup Architecture

ISO/IEC/IEEE 42010 standard for architecture descriptions gives the following definition: "Architecture is the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution" [119]. Software architecture can include numerous views describing different levels of abstraction of the sofware. Following a reference architecture in client-side mashup composing will result in flexible implementations that are still straightforward to maintain. In the following we first identify mashup architecture requirements and then describe our reference architecture for mashups. The topic

is discussed in more detail in Publication VI including the descriptions about mashup implementations that our research is based on.

4.2.1 Mashup Architecture Requirements

Architecture requirements for mashups can be derived from common goals of different mashup systems. Mashups are build on top of independent services and aim at providing superior user experience while preserving quality attributes, such as performance and modifiability. In addition, choosing the right architecture model for a mashup is vital for security. If a mashup has access to user's private data, but utilizes untrusted dynamic code or content as well, providing architecture level security features is important.

Content accessing. Service interfacing is an integral part of mashup architecture. Mashups are vulnerable to changes in interfaces, and mashup architecture should support either automatic or manual reacting when such changes occur [45]. Furthermore, the overlapping nature of the Web provides an interesting challenge for mashup architecture. It is common that several web services provide roughly the same functionality. There are, for instance, numerous mapping services – Google Maps, Yahoo! Maps, Bing Maps and OpenStreetMap, for instance – that could be used to create a map component within a mashup. Unfortunately legal terms, availability of services, features of services, and response times differ accross services, which makes creating an architecture that allows interchangeability of services more difficult.

Data handling, converting and exchanging. Data handling, converting, and exchanging are typically a central part in mashups, because of data downloaded from different services is inevitably in diverse formats [16, 45]. Before data can be mashed up, it needs to be converted into a common format that is straightforward, if possible, for the application to handle. This is important to be able to create a flexibility adding abstraction layer between services that are used and the mashup composition functions. While this is true when established web serviced are accessed through well-documented interfaces, it is especially vital when a mashup extracts source data from an HTML page, where the data scraping and parsing is extremely error-prone if the source HTML page changes.

Mashup creation. Application logic is the processing that actually creates the mashup [21]. It utilizes the data that has been converted into a common format and

combines it in novel fashion. How the logic is determined is sometimes dependent on the tools that are used. For instance, with some tools, such as Yahoo! Pipes, the composition is generated graphically. If the mashup is created manually, the application logic is typically determined using JavaScript or other dynamic language, and implemented by writing a suitable program code.

Creating user interface. User interaction is included in practically all web applications targeted at end-users, and mashups are not an exception. User interface implementation should be separated from the implementation for numerous reasons [120, 121]. For instance, this allows the widget set to be changed to another according to target device type. In addition, need for internationalization as well as customization of the user interface motivates separating it into an independent module. However, in some cases the underlying service may restrict the user interface implementation, map-based applications being a classic example of such construction.

4.2.2 Reference Architecture for Mashups

In the following, we summarize the reference architecture for client-side mashups, which has been described in detail in Publication VI. The reference architecture (see Figure 4.1) can be used as a fundamendal structure when composing client-side mashups using manual development methods. Naturally, in practical implementations, some components of the reference architecture can be merged with others or omitted completely. We describe the reference architecture modules in the following.

Content providers. Content providers are interfacing modules that access web service interfaces, local files, or device peripherals to retrieve content for the mashup. These interfacing capabilities are, by definition, an integral part of any mashup application, even though they are not part of mashup architecture *per se*. Accessing content, whether it is available in the device itself or already existing in the Web, is carried out by content provider modules that can be easily modified, replaced, or removed as standalone components.

Context extractors and formatters. Data model of the mashup reference architecture consists of content extractors and content formatters. These two types of components acquire, manage, and maintain the data that is used by other modules of the mashup. Data model includes operations that accesses the required data and extracts it from different sources by utilizing content providers. Furthermore, data model

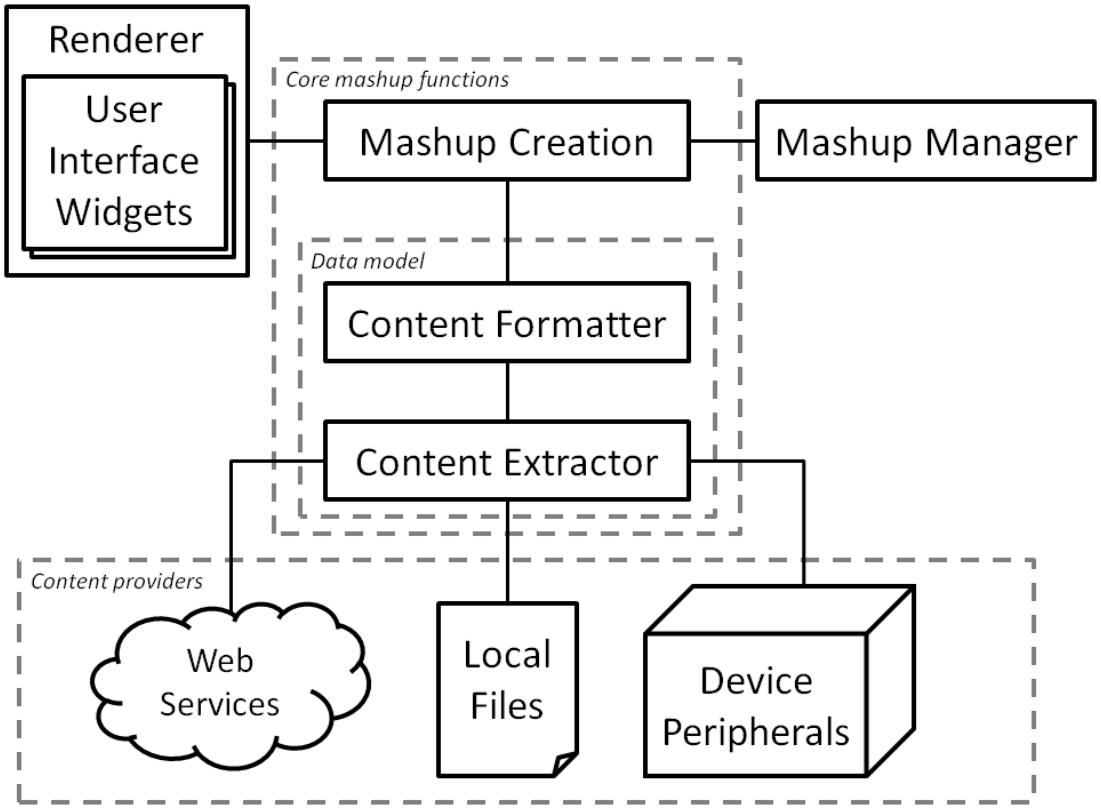


Figure 4.1: Mashup reference architecture.

includes functionalities, i.e. content formatters, to alter the data by parsing, filtering, and converting it. These functionalities can be composed together in layered fashion. Furthermore, data model of a mashup can maintain local cache or database where the data is stored for faster accessing or offline use.

Mashup logic. In order to compose a mashup, a module referred to as mashup logic draws the content together and performs the actual mashup related functions. Different mashups may be built on the same content providers, extractors, and formatters, while the result is different as the mashup logic is unique. Naturally, methods used to determine the mashup logic may depend on the tools used and environment where the mashup is created. In most systems the logic is described in program code, but in others, a graphical tool is used to generate the logic.

User interface. Implementing mashup's user interface as a separate module has evident benefits. Most mashup designs rely on an existing set of user interface widgets,

and often it is desirable that replacing a set with another is straightforward. Depending on the selected widget framework, it may be necessary to merge mashup logic partially with the user interface, as in some widget libraries the data is binded with the graphical presentation. In some cases the result mashup may even be rendered on completely different device, such as a TV set or a public display, for instance.

Mashup manager. To overcome the fragile nature of mashups, a mashup manager is included into the architecture. It is capable of making updates to other mashup components, for instance to content providers when service providers change their interfaces. In addition, mashup manager is responsible of changing a component completely into another, as is benefical when some part of the mashup is failing and a fall-back module is available and ready to be taken in use. Furthermore, mashup manager can make operations that affect more than one mashup component. For instance, it can determine whether the mashup should work on online or offline mode.

4.3 General Design Principles

In the following we point out considerations that should be taken into account as general design principles when composing mashups. These considerations are explained further in Publication VI along with the example mashup implementations that our findings are based on.

Design applications according to software engineering principles. When designing mashups, adhere to established software engineering principles, and follow an architecture that takes special features of mashup development into account. For instance, our mashup reference architecture that can be used as a starting point is described in Section 4.2. Utilizing suitable JavaScript libraries benefits mashup authoring with traditional web development methods. Such libraries are available for JSON and XML parsing as well as for making requests over the Web. In addition, libraries are useful for creating user interfaces or enhancing cross-platform compatibility between different web browsers. However, selecting suitable libraries should be done carefully. Sometimes the selection has side effects on selected application architecture model, for instance some libraries suit better for 3-tier architecture [122] and others for MVC model [123, 124]. Typical further issue is that libraries are often incompatible, or have incompatible libraries that they depend on.

Follow standards and recommendations. Following standards and recommendations on mashup implementation improves changes to develop mashups that are easy to maintain. Open standards, established data formats, such as JSON, RSS, and XML, are less likely to change over time than proprietary standards and formats. In addition, following recommendations regarding JavaScript programming such as guidelines, patterns, and best practices (see [125, 126]), can improve quality, security, and performance of a mashup implementation.

Consider using quality evaluation frameworks. Quality evaluation can be done with suitable frameworks. For instance, Cappiello et al. have evaluated mashup components and data used in mashups from quality point of view in [127] and [128]. They have introduced a quality model that can be used when evaluating quality of interfaces, libraries, and data used in a mashup. In addition to tangible quality measures, stability of mashup components should not be overlooked either.

Test carefully. Careful testing is essential when developing dynamic JavaScript applications [129]. Tests should be done with a weak network connection and in the offline mode, too. Tools and libraries make testing easier (see, for instance, tools introduced in [129, 130], and Selenium, `http://seleniumhq.org/`, as well as JSLint [125], `http://www.jslint.com/`). Unit testing is possible with special purpose JavaScript libraries and well designed debugging tools are built-in web browsers or available as extensions [131]. Testing should be done with different browsers and browser versions. If a mashup is built on experimental technologies, support for these technologies may be discontinued or changed when browser version changes. As some browsers update automatically, such mashups may stop working unexpectedly.

Be visually impressive and keep usability in mind. Combine data and other content in an innovative, visually impressive way. Use innovative data visualization techniques to help the user to handle complex information (see [132] for qualitative study about mashups). Most users "see only what they see", and their perception of quality is based on attractiveness of the interface they see [133, 134]. Therefore, user interface issues are really important. In addition to look-and-feel of the user interface, user experience of a mashup has other aspects as well as described in [132]. For instance, taking user context into account in the implementation, and using it to provide relevant information, can be a great benefit. Edward Tuftes books on data visualization (e.g., [135]) can serve as a helpful starting point.

4.4 Accessing Web Services

Issues that are faced when using technologies available in web browser to access web services include interface changes, timing problems, as well as lack of documentation, tutorials, and examples among other difficulties that make the implementation of mashups more cumbersome. These issues are explained further in Publications IV and VI, and the following remarks are based on our own experiences in developing mashups mentioned in these publications.

Changes in interfaces are naturally difficult for mashups to handle automatically. Typically changes are more common in new service interfaces that are in their beta or alpha stages, in contrast to established services. However, changes can occur even in established services and, therefore, a flexible model to access services is necessary to be able to reflect such changes [136, 137]. Utilizing open services and wikis as resource providers for a mashup decreases dependence on single service providers and is therefore worth considering. Community provided data should be used carefully, though, as accuracy of the information provided is not always sure. Some services, such as Google Maps, have enabled accessing their services with old versions of the interface as well. This is achieved by determining the exact version number of the JavaScript API in the request URL.

"Screen scraping" or "web scraping" – in other words, parsing source data from HTML pages aimed at human readers – should be avoided altogether. Using such approach to obtain data may seem tempting, because of it enables accessing desired data that is not available through proper web interfaces [113]. Many services, however, deny accessing their content in such manner with a legal disclaimer. In addition to legal issues of using screen scraping for harvesting content, this technique is especially prone to failing when changes occur in the representation of the service. Furthermore, some web services have implemented technical measures in order to prevent scraping.

Adapting new technologies and services is often risky in terms of stability and maintenance. Documentation and tutorials of new frameworks and interfaces may be nonexistent or obsolete. Sometimes only examples are provided to developers to explore. These factors slow the development down and make it more difficult. In addition, experimental interfaces may be discontinued, causing a need to reimplement the mashup.

Building the mashup in manner that is easy to modify when changes occur is beneficial. Anticipating changes includes possibility to change a resource interface to another providing similar service as the original [137]. Therefore, binding the application modules too tightly with a certain data format or accessing method should be avoided. Instead, layered architecture and loose coupling with services should be used to enable modifiability, especially when building on experimental services.

Fall-back mechanisms can be built in order to ensure mashup functionality in degraded level when a service becomes unreachable. For instance, a mashup using information about user's location can revert to IP-based or manually entered location if GPS data is not available. Another example of such fall-back mechanism would be utilizing services with different quality for reverse geocoding, where an address or a place name is resolved from a point location, i.e. latitude and longitude values. In some cases a mashup may even be able to adapt on the fly by changing a service to another providing roughly equal functionality. For instance, a map component can be aquired from numerous overlapping services. Moreover, caching the data locally may provide a way to ensure undisturbed mashup operation.

Adding mechanisms that alert the mashup developers when a service is failing is one way to prepare for changes in interfaces. However, adding such mechanisms usually requires some user collaboration, especially if the application is sending some information that could be used to identify the user. Using error handling constructs of the chosen implementation language, like *try...catch* statements in JavaScript, is strongly encouraged, as mashups are more likely to suffer from unexpected errors in contrast to traditional web applications.

Some runtime environments that can be used as mashup platforms do not implement all HTTP methods [62]. This leads to difficulties when using an interface that applies unavailable methods. For instance, Google Calendar API utilizes GET, POST, PUT, DELETE, and PATCH [138] methods, and common functions such as deleting an event is not possible if DELETE method can not be used. Examples of platforms that lack support for some methods include Adobe's Flash, Oracle's Java ME (`http://www.oracle.com/technetwork/java/javame/index.html`), and Digia's Qt Quick (`http://qt.digia.com/Product/qt-quick/`). In addition, HTML 4.01 specification [3] does not support PUT and DELETE in `form` elements. The support was added to HTML5 specification [11], but removed later.

It is common that service interfaces require using an API key to identify either applications or developers, and to separate commercial and non-commercial applications. API key is a distinctive character string that is included into all requests made to the service. In the early days, services supported only server-side applications, and API keys were connected to certain URLs or domains. Today, services may provide both server- and client-side keys for different types of application as client-side keys can not always be attached to specific URLs or domains.

Accessing asynchronous web interfaces may lead to timing issues as well. Depending on the perfomance of the service, network connection, and other factors, response times to resource requests vary, and this can lead to problems if the mashup is not implemented carefully.

4.5 Accessing Local Services

Web applications can be executed on embedded devices, and especially on mobile devices that have been the forerunners in this field for obvious reasons. Moreover, network connectivity has been utilized in game consoles, set top boxes, and smart home devices, among others, to provide possibility to access different web services or to enable accessibility from the Internet. In embedded devices the network connectivity has been used for delivering software updates, providing a remote accessing through a web server, and enabling web browsing. In addition, connectivity has been utilized in multi-player games. Consequently, typical examples of applications utilizing the Web in embedded systems are either static binary ones, where the role of the Web is to provide means of sending and receiving data, or those presenting dynamic content in the form of web pages.

In addition to these use cases, implementing mashups on embedded devices is possible as well. Mashups can mix web technologies with the capabilities of web enabled embedded devices and be able to novel experience for embedded device users. This is possible when mashups combine device's local resources such as local files and sensor data with seemingly infinite resources of the Web.

Approach of this kind requires, however, a special runtime environment. Accessing local services in tandem with web services can be achieved with two methods that are different from mashup developer's point of view. The former method requires

a customized web browser or a runtime, while the latter one can be used with web applications that use a regular browser as a runtime environment.

The first method is implementing a special purpose native runtime that exposes local data to dynamic mashup code via bindings that are either generated automatically or implemented manually. The local data may be accessed through native subsystem providing peripheral data (e.g. D-BUS [139]) or directly through a proprietary scripting interface. In Publication V we have described in detail two runtime environments – one supporting procedural programming and another supporting declarative programming – that can be used as a mashup platform. Besides different approach for programming, accessing device sensors is implemented in different fashion in these two environments. The declarative environment provides a proprietary interface and the procedural one uses device's subsystems for accessing peripheral data. When peripheral data is accessed in this way, it is typically in a format that is completely different from what web interfaces provide, and converting data into another format is almost always necessary. For instance, GPS sensor data accessed through D-BUS interface is usually NMEA-0183 formatted, and converting it into XML or JSON format, which are easy to handle with JavaScript, requires a special purpose parser to be implemented.

The second method is to use a local server that exposes the data over a web interface that can be used with asynchronous requests in similar manner as any other web interface. When a native component with a local web server is used to provide peripheral data, such conversion is not necessary in the mashup implementation. Further advantange gained with this approach is better cross-platform compatibility of the mashup implementation, provided that the local server is available, as now the mashup can be executed on a regular web browser. Moreover, changing source data provider into another, for instance in a fall-back recovery situation, is more straightforward when the interfaces are similar.

4.6 Considerations of Mashups on Mobile Devices

Mashups on mobile devices can be developed using the same tools and methods that are used to create their desktop counterparts if they are targeted at a runtime platform that is available across environments. However, our experiences – further described in Publications II, III, IV, and V – raised issues because of differencies between desktop

and mobile systems in runtime platforms available, network connection technologies, energy consumption restrictions, operational expenses, and possibilities for accessing user's context. Brief summary of our findings is provided in the following.

Cross-platform compatibility. In general, mashup cross-platform compatibility relies on underlying software platforms. For instance, mashups built on web browsers have the same functionality across operating systems where the browser is available. Similar compatibility can be achieved using other runtime environments such as Adobe Air (`http://www.adobe.com/products/air.html`) or Qt (`http://qt.digia.com/`). However, this is true only on desktop and on mobile devices the situation is more difficult. The mobile device browsers are usually lacking behind their desktop counterparts in functionality. Updating a pre-installed web browser on a mobile device usually happens only when the manufacturer releases a new version of the operating system, and consequently it takes time for the latest features to become available. Even browsers based on the same browser engine do not necessarily have the same capabilities on devices from different manufacturers or with differrent operating system versions. Therefore, using an application framefork that has support for desired platforms to hide differencies in details as well as designing for the lowest common denominator is often necessary.

Need for constant access to the network. Network access with a mobile device is typically not as reliable and high quality as we are used to on desktop environment. Available bandwidth and latencies may change significantly depending on different factors such as the current cellular connection type and simultaneous traffic caused by other users in the network. This can have an effect on mashups run on mobile devices and should be taken into account at the design time. This can be done by caching, compressing the data to be transferred, or using a server-side mashup approach where most of the content retrieving can be done on the server. In addition, if it is possible, attention should be paid on interface design so that it is possible to transfer only that data which is needed at the moment, in other words, overhead needs to be weeded out. For instance, when information about one user's status is needed in the application, it would be wasteful to load the entire user database. In some cases the problem of frequent web communication arises from a need to actively poll a web interface in some specific intervals to get notifications for new content. This can be avoided with techniques that eliminate the need for active polling. These

techniques, commonly referred to as server push, include Comet [140], JSONRequest object (`http://www.json.org/JSONRequest.html`) as well as Bayeux [141] and BOSH (Bidirectional-streams Over Synchronous HTTP) [142] protocols. Furthermore, Server-Sent Events and WebSockets, both introduced in a draft of HTML5 specification [11], can be used. However, to be able to utilize this kind of techniques, an interface needs to support one, which often is not the case. Further, if the server pushes new data to the mobile device in very short intervals, the situation is not different from using active polling in sense of need for constant networking.

Battery consumption. Rich mashups may drain mobile device's battery quickly as they frequently communicate over network and typically have a graphical user interface that is used to display the results. These problems are the same with most mobile web applications, but in the case of mashups the web communication can play even bigger role. Any application that forces the mobile device to be immobilized for loading the battery in the middle of a working day is most likely unsuccessful, and therefore this aspect of a mashup requires attention.

User operational expenses. There are numerous types of service plans for a mobile internet connection, for instance: pay-per-use, unlimited data plan, limited data plan with additional data charge for exceeding the limit, and tiered data plan where the price goes up in gradual increases. Because of capacity issues, the trend has currently been moving out from unlimited data plans into other types of plans where the data has a price tag. Even though this issue is not only a concern of a mashup developer, but other parties as well, it has an effect on mashups, as they are depending on web accessing capability of the mobile device. Rich mashups with multimedia content are definitely large-scale consumers of mobile data, and currently the cellular operator's business model is not fully compatible with unlimited use of this kind of applications. Many other types of applications, such as streaming audio and video players, are affected as well, and we will propably see new kind of business models that enable the full-scale use of this kind of services.

Access to context. Possibility to access user's context is a great advantage for mobile mashups [143]. Achieving this requires access to device sensors, which is not always straightforward. In general, sensor data is not available for applications running inside a browser, with the exception of location, which is available, if the browser implements W3C's Geolocation API [144]. However, in addition to location, there

are numerous other sensors that could be utilized is mashups as well. Luckily, new APIs for accessing sensors and internal services of devices have been developed for instance by W3C's Device APIs Working Group (`http://www.w3.org/2009/dap/`). Some device manufacturers have already opened access to mobile device sensors with more or less standardized interfaces. For instance, Apple has implemented W3C's DeviceOrientation Event specification [145] in mobile version of Safari browser, and this enables utilizing device's accelerometer and gyroscope sensors in web applications.

4.7 Security

Security is a major issue with mashups, where loading arbitrary content from potentially distrusted sources opens up an opportunity for attack. Mashup security can be improved if certain practices are followed when mashups are designed, and this topic has been subject for numerous researchers as well. Different models to solve mashup security problems have been proposed as we described in Section 2.5.3 (see also [106, 146], which have been introduced more thoroughly in Publication II, as well as [147, 148, 149]). In this Section, based on literature, we provide an overview of mashup attack scenarios and security practices.

4.7.1 Attack Scenarios

Mashups can be attacked against with the same methods that can be used against any other web applications. In addition, because of the dynamic nature of mashups, they can be more prone for certain types of attacks. Therefore, accessing multiple data sources needs to be done carefully to avoid security problems. Three attack scenarios considered relevant to mashup development in [94, 150, 151] are cross-site scripting, cross-site request forgery, and JSON hijacking. In the following we provide a brief summary of the scenarios.

Cross-Site Scripting (XSS). XSS is one of the most common application level attacks [152, 153]. XSS attacks are based on injecting distrusted content into a dynamic page. This can be done for instance by exploiting unsanitized HTML form input or HTTP query parameters. With XSS, every input is a potential attack vector. If the data provided by the attacker is saved by the server, malicious scripts can be later executed automatically, which makes it more difficult to notice the attack. XSS

attacks can be used for hijacking user sessions, defacing websites, and injecting worms into user's computer, among others.

Cross-Site Request Forgery (CSRF). CSRF utilizes a malicious site that instructs a victim's browser to interact with an honest web site. Consequently, CSRF leverages victim's network connectivity and browser state while disrupting the victim's session with the honest site [154]. The attack can be executed, for instance, by including a link or script that makes the user's browser to access a site by which the user is known and authenticated. If the authentication has not been expired, the attacker may be able to successfully perform operations on the honest site by using victim's credentials [155].

JSON Hijacking. JSON hijacking (or JavaScript hijacking) is a special type of CSRF attack, and it refers to a technique that can be used to attack JavaScript application with malicious JSON formatted data [156]. JSON formatted data can hold JavaScript function calls, and if the unsanitized JSON data is converted to a JavaScript object with unsecure `eval()` function, these function calls can be used to override standard JavaScript object constructor functions in order to gain access to victim's sensitive data. This is possible as the same origin policy does not prevent including JavaScript from one web site and executing it in other web site's context. While CSRF is normally a blind attack (the attacker can not see what the honest web site sends back to the victim in response to the forged requests), CSRF against JSON calls allows the attacker to see JSON responses sent by the honest web site.

4.7.2 Security Practices

There are a number of practices that, when followed, can increase mashup security remarkably. Authors of [94] suggest developers to focus on protecting their applications against XSS and CSRF attacks.

Check input values. Input value checking is necessary to prevent XSS attacks, and it should be implemented on both client as well as server-end regardless whether a web browser is used as runtime environment or not [94]. Input value checking can be done by sanitizing the inputs from all possible malicious content by filtering it with either whitelist or blacklist technique [150]. Another way to improve security is to escape special characters in input strings [150]. In mashups, input sanitization should be applied to resources retrieved through web interfaces, as well.

Prevent CSRF attacks. Preventing CSRF attacks is more complicated. One method is to require HTTP requests to include a secret user-specific token that is passed between the client and the server when requests are made [154]. Another countermeasure is to require client to provide the actual authentication data along with HTTP requests. Method that is referred to as "double-submitted cookie" can be used to prevent CSRF as well [157]. In this method a cookie containing a secret token is used along with a token submitted in the actual HTTP request. The server accepts only those requests that have matching token in both the request and the cookie. This prevents CSRF as the attacker does not have means to modify or read the value stored in the cookie. Furthermore, web services should avoid using HTTP GET requests to initiate changes and use POST requests instead [157]. Even though the latter does not prevent all CSRF attacks, it works as protection against some types of CSRF, including some cases of JSON hijacking. Other means to make CSRF attacking more difficult, yet not impossible, are limiting lifetime of session cookies and checking HTTP `Referer` header. JSON hijacking can be prevented by using reliable JSON libraries to parse the data instead of manually using `eval()` function [94, 150]. Some browsers provide native JSON encoding and decoding, which is faster and more secure way to handle JSON formatted data when compared to solutions implemented in JavaScript.

Use vulnerability checking tools. Vulnerability checing tools can be used to find common exploits from web applications. Usually these test suites are provided for free, and they can be used to find XSS vulnerabilities, system configuration problems, etc. In addition, commercial scanning services are available.

Be careful when generating and executing code dynamically. Usually security guidelines instruct to refrain from generating and executing dynamically created code – especially with JavaScript's `eval()` function [94, 150] – as this kind of functions are typically easily exploitable. This, however, may be integral functionality in some mashups. Therefore other ways to secure the application need to be implemented. One possible solution is to create a verification mechanism, such as digital signature that ensures that the dynamically loaded code has not been tampered.

4.7.3 Accessing Interfaces with Separate Origins

Same Origin Policy is a web browser security model that distinct separate origins from each other [22, 23]. The same origin policy is ill-suited for mashups because of the

following reasons:

- The same origin policy applies only on document object model, not on JSON data or JavaScript code, for instance. Therefore, the same origin policy can not protect against CSRF attacks [94].
- Numerous ways to circumvent the same origin policy exists. Trusted domain can easily be used as a proxy for distrusted data. If this is done, as is often the case in mashups, the same origin policy does not offer any protection.
- Checking if two resources are of the same origin is not always reliable. Web browsers enforce the same origin policy by checking the domain name of the server as a string literal and ignore the possible path expression of the URL [94]. In addition, for two resources to be considered to be of the same origin, application layer protocol and port number need to be the same as well. Therefore, domain names `http://www.example.com`, `http://example.com` and corresponding IP address `192.0.43.10` are interpreted as different domains. As pointed out by Crites et al. [85], the same origin policy relies on insecure services such as the Domain Name System (DNS), and it is prone to *dynamic pharming* [25] as well as *DNS rebinding* [158] attacks.
- Some web browsers allow to override domain property (`document.domain`) in the document object model, which makes the same origin policy ineffective.

Mashups, by definition, access content from different sources. On purely client-side mashups, where using a proxy server is not an option, the same origin policy needs to be violated. Therefore, more fine-grained solution for mashup security is necessary. One such solution is *Cross-Origin Resource Sharing* (CORS) specification [95] by W3C, which defines a standardized way to access resources in trusted domains.

CORS determines how a browser and a server can interact and determine whether or not to allow cross-origin requests, and it provides means for developers to create mashups without difficulties caused by the same origin policy. CORS can be used as a more robust alternative for JSONP, which has been often used to circumvent the same origin restriction.

However, CORS sets some requirements for interfaces used as well as browsers. It requires server interface to include `Access-Control` headers to its responses, and the

web browser needs to have support for CORS as well. Luckily, even though CORS specification is still at draft stage, it is already implemented in most web browsers. Furthermore, CORS has been adapted by other specifications, such as WebGL, where textures are subject to cross-domain access control. Currently, numerous web services already support CORS in some level, but others have not implemented CORS in their interfaces.

The upcoming HTML5 specification [11] includes a technique called Web Messaging [159] that enables secure cross-document messaging. This technique is targeted at different problem than CORS, which addresses accessing content in remote servers. In contrast, Web Messaging can be used to pass messages between different browser windows and `iframe` elements that can be of different origins.

5

Towards Software as a Mashup

In this Chapter, we introduce an idea to extend mashups from composing content artifacts and social communication to composing applications dynamically out of executable code components. We argue that this kind of software will be integral part of the future web, sometimes referred to as "Web 3.0". Here we discuss requirements of such systems and describe the most substantial issues that need to be overcome. Furthermore, we introduce our proof-of-concept implementation of a system that can be used to create software as a mashup.

The Web has turned from a simple document sharing platform into a pervasive distribution platform where different types of content artifacts, such as images, videos, pieces of music, maps, books, and news items can be shared in the global scale. Social communication and messaging have moved to the Web as well. In addition to hosting data and other content, the Web is used increasingly as a software platform, and its main model of delivery, referred to as software-as-a-service (SaaS), implies that applications are available without client installations or manual upgrades. Instead, web-based software is loaded dynamically on the fly as needed. There are numerous examples of such applications in enterprise setting including customer resource management (CRM), enterprise resource management (ERP), human resource management (HRM) and content management (CM). In addition, web applications available for public end users include email, games, and office applications such as Google Docs (`http://docs.google.com/`) and Microsoft SkyDrive (`http://skydrive.com`).

Mashups are utilizing the Web as a pervasive distribution platform and combining content from multiple sources into an integrated experience. Unlimited access to data through open interfaces enables completely unforeseen mashups. All this, however,

requires careful implementation and taking numerous details into consideration, as described earlier.

When the trend towards web-based software is taken into consideration, an interesting question is how to realize mashware, or software composed as a mashup. To answer this question, we have studied the related efforts made in this area and created our own mashware implementation on top of a custom runtime environment. In the following, the background Section presents the current state of the art and is based mainly on the literature. Other Sections of this Chapter are based on work originally presented in Publications VII and VIII.

5.1 Background and Related Work

Mashing up content artifacts is not the only option for mashups. Instead of individual applications using resources, it is possible to use application components and download them in the similar manner as we are used to download complete applications and access online resources. This kind of software created as a mashup is referred to as *mashware* [16], which is enabled in unforeseen scale by the fact that for the first time we now have the Web, a global, uniform distribution channel. Consequently, a web-based software component architecture is required to be able to mix software pieces together, as explained in [16]. Similarly to a mashup ecosystems where mashup developers and service providers collaborate, a global scale ecosystem of software component providers and mashware developers – in other words a mashware ecosystem – can be created.

There are clear benefits for creating web software as a mashup. As described in [16], this kind of environment would allow software developers to collaborate on an extremely large scale. This would allow not only reuse of data, but user interface widgets, data handling modules, and wireframes, among others. Loading necessary components on-demand basis at runtime is a great benefit in terms of implementation flexibility and cross-platform compatibility. For instance, flexibility of the system increases as the system may download suitable version of the user interface after it has found out the desired screen resolution and determined whether the device has support for touch input or a regular keyboard and mouse interface. Cross-platform compatibility can be enhanced as necessary device specific interfaces can be added later on request as well.

Combining software components together is not a new idea. In software development, the possibility to reuse software modules has been a common practice for a long time. Building large software systems out of prefabricated, reusable components created by different software developers was originally proposed by McIlroy and other participants of NATO Software Engineering conference back in 1968 [160], and since then techniques for reusing software have been investigated for decades.

5.2 Component-based Software in Web 3.0

As pointed out in Publication VII, combination of technologies under Web 3.0 umbrella will change the Web into even more suitable platform for mashware. While the term "Web 3.0" has been sometimes used as a synonym for semantic web [161], our perception of Web 3.0 is similar to [162], where the term refers to new ways to use the Web, and using it in new domains. Technologies driving Web 3.0 include semantic web [163, 164], HTML5 [11] and WebGL [96], advanced security models such as Cross-Origin Resource Sharing (CORS) [95], high bandwidth network connections, and linking the physical world with the Web. Especially, the semantic web, which refers to giving well-defined structure and meaning for data stored currently within unstructured and meaningless web documents, will obviously benefit mashups that could utilize such data in unforeseen contexts easily. The other technologies – enhancing pervasiveness and cross-platform compatibility of the Web as well as enabling new kind of applications – allow implementing mashups in new domains, such as different kinds of embedded terminals, for instance.

Desire for component based software has been associated with the definition of so-called Web 3.0 as well. Interestingly, Google's CEO Eric Schmidt described ideas similar to mashware when he gave his definition for Web 3.0 at Seoul Digital Forum in 2007 [165]. He remarked that Web 3.0 applications will be pieced together, relatively small, able to be run on any device, fast and customizable, distributed via social networks, and using data stored in the cloud. Characteristics such as "pieced together", "fast and customizable" and "using data stored in the cloud" are features of mashware as well. However, mashware is not limited to small applications and distribution through social connections. Furthermore, universal-scale cross-platform compatibility is not a

requirement for mashware applications, even though this can be achieved with certain platforms.

Another interesting remark, originally presented in Publication I, is that programmatic interfaces – interfaces that are utilized in web applications by including a JavaScript library created by the service provider – have become more and more common, and this phenomenon can be construed as a step towards mashware. One reason behind success of such interfaces is that they can be used conveniently with regular JavaScript function calls, similarly to DOM and other interfaces found in web browsers. Often the library is not hosted on the same server as the mashup, but downloaded from the service provider's server when needed. Consequently, the most recent version of the interface is always used. The most successful example of this kind of interface is Google Maps API, which is also the most popular interface used in mashups [99]. However, other examples are available as well, including services for user authentication, social networking, music and video playing, and data visualization.

5.3 Implementing Software as a Mashup

Mashup development is different from conventional software in numerous ways, as described in [16, 45, 116], and reusing web software at the level of implementation has not yet reached its full potential. There is still a an impedance mismatch between web-based software development and software engineering, and development practices for web applications are far from maturity levels of traditional software, as pointed out in [17]. Especially, when taking the desire of creating web applications from software components into account, it is apparent that research is needed in numerous areas, such as development practices, security, modularity, legal aspecs, and software engineering methodologies.

Even though some JavaScript libraries and application frameworks such as jQuery or Prototype exist, they only provide some basic networking functionalities and user interface widgets and effects, and using them concurrently in a single application is cumbersome. Currently, closest thing to individual software component system is Google Maps, which is typically used for adding a map component into a web application. However, typical application using Google Maps uses the map API for implementing all other user interfacing activities as well, effectively defining the capabilities of the

whole application. This makes using Google Maps is similar to any other application framework rather than a generic downloadable software component.

When implementing web application architecture based on dynamically composed software, need for carefully designed, compatible interfaces is self-evident. Current interfaces used in web services are almost exclusively proprietary and vendor-specific. For instance, changing an application from using Google Maps to OpenStreetMaps would require major code rewrite efforts. If a component market is desired, standardized interfaces to different services need to be created and published online.

Furthermore, components need to be delivered in platform-independent format without static linking or advance binding [16]. Moreover, a security model allowing components to be added dynamically into an application is required. If the execution environment is a custom one, such as one used in Publication VIII, the security model can be self-determined, but if a standard browser is used, similar concerns as those presented in [16, 17] can be raised. We believe that ongoing work, such as W3C's Security Activity Proposal (`http://www.w3.org/2011/07/security-activity.html`), will eventually solve these issues, but before that, there will be numerous attempts to circumvent the problems.

Another problem when implementing large mashware applications in a web browser is that all dynamic code is executed in a single context. Therefore, the implementation is sensitive to naming collisions and performance degradation when the client-side codebase is extended with dynamic modules. Even though JavaScript does not inherently support modules, naming collisions can be avoided by specifying object interfaces carefully and following JavaScript namespace patterns diligently. In addition, there is an ongoing work to develop a specification for *web workers*, long running background scripts that run independently from user interface scripts [166], which can offer a partial solution for executing code in different threads. However, the specification points out that web workers are relatively heavy-weight, and are not intended to be used in large numbers, as they have a high start-up performance cost, and a high per-instance memory cost [166].

5.4 Proof-of-Concept Implementation

In our practical experiment of a mashware system, presented in Publication VIII, we have applied the mashup architecture introduced earlier in Subsection 4.2.2. As web browser security model is badly suited for this kind of applications, this prototype utilizes Qt-based framework called Lively for Qt (`http://lively.cs.tut.fi/qt/`) as a web runtime, similarly to mashups presented in Publication IV. In the following, we briefly introduce the main points of the implementation and lessons learned.

Application architecture. Architecture of the mashware application (see Figure 5.1) is in align with the reference architecture for mashups and based on separating different functions into independent components. Here, three types of downloadable components are included into the system, but the architecture can be extended with other types of components if desired (see Publication VIII for further description of selected components). The three types of components are user interface widgets, content extractors, and content formatters. The components can be requested with a mashware manager that is capable of searching suitable components form a mashware repository, a RESTful service containing component meta data. Content extractors and formatters together form the data model of the mashware application. Finally, main application containing the application logic, and actually composing the mashup by managing the data presentation with rendering components, is included.

Figure 5.2 presents a mashware application using the described architecture. The application is capable of searching and displaying images from an image hosting service. In addition to image service accessing component, the example utilizes image filtering component, user interface component library, and a component capable of accessing a social networking service. Naturally, communication between the modules has a key role in the mashware application. In our implementation, the communication is based on common module interfaces, that are used to pass structured data from a component to another.

Mashware repository. Mashware repository, which can be generalized into a set of repositories, is used for discovering components that are available. The repository contains meta data of the components, and it has a RESTful interface for requesting the data. The interface is self-descriptive, and components can be searched based on their capabilities. For instance, a rendering component capable of presenting

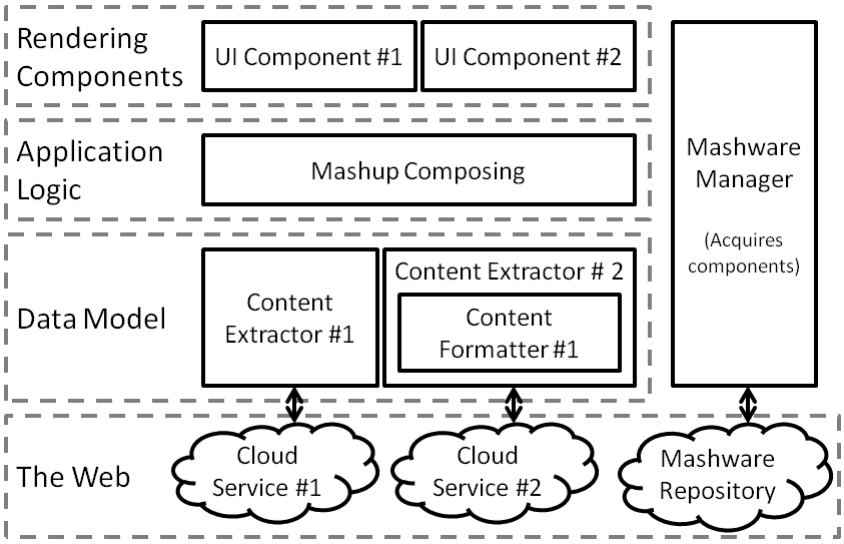


Figure 5.1: Example of mashware application architecture. Mashware manager is used to download and enable components from the Web. Mashware repository is used to locate the components.

content with `image/jpeg` MIME type can be searched with an HTTP GET request `http://example.com/renderers[contentTypes='imagejpeg']`. Furthermore, meta data entries can be linked together, which makes possible to recommend compatible components.

Mashware manager utilizes the repository to search, download, and select components to be used in the mashware application. An implementation can use more than just one manager, and the manager can be loaded dynamically similarly to any other component in the system. Furthermore, managers can cache components that are used often and provide means to extend them at the client-side, if desired. Autonomous selection of the components is possible thanks to possibility to search components based on capabilities, as well as linking compatible components.

Using the repository and components in the application logic. To illustrate how the mashware architecture is used in our implementation we provide some example code listings written in JavaScript. As the listings show, the mashware manager is responsible of providing the actual components that are selected in the application logic.

In Listing 5.1 the mashware manager is used to access the repository and fetch component meta data based on component types. Mashware manager has a function

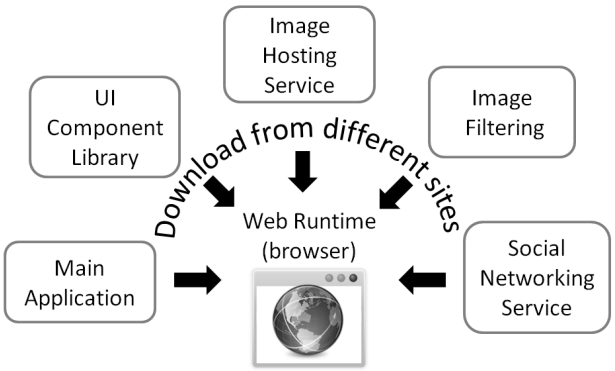


Figure 5.2: Example of mashware application. This application could be used to search and combine images with relevant content from a social networking service. Way of presenting the application structure is adapted from [16].

Listing 5.1: Example of requesting component meta data from the mashware repository.

```
// Create new mashware manager instance
this.mw = new MashwareManager(this);

// Requesting possible components with
// mashware manager from the repository.
// The manager returns an array of meta data objects.
var renderers = this.mw.getComponents('renderer');
var contentproviders = this.mw.getComponents('contentprovider');
var filters = this.mw.getComponents('filter');
```

`getComponents`, which is used to get a meta data array based on component type provided as a string.

In the current implementation, selection of suitable component is done in the application logic by using the exact name of the desired component as a key for function `selectMetadata`. In Listing 5.2 three component instances – two content providers and one filter component – are requested. The mashware manager has a function `getComponentInstanceWithMetadata` to perform this functionality. When this function is called, the instance parent is passed as a parameter for the mashware manager to be able to enable the component in the right context. The filter component is passed to be used in another component.

Finally, Listing 5.3 shows how the component instances can be used in the application logic to perform actions. In this case, a content provider component is used to make a search to a web service when a user interface button is pressed.

Listing 5.2: Example of requesting component instances using meta data as query parameter.

```
// Using meta data to request component instances.
// First parameter passes the meta data,
// second refers to component type, and
// third points to the parent of new component instance
this.visualrest = this.mw.getComponentInstanceWithMetadata(
  selectMetadata(contentproviders, 'VisualRest'),
  'contentprovider',
  this
);

// Passing the filter component to be used by another component.
this.visualrest.setFilter(
  this.mw.getComponentInstanceWithMetadata(
    selectMetadata(filters, 'OnlyTaggedImagesFilter'),
    'filter',
    this.visualrest
  )
);

this.twitter = this.mw.getComponentInstanceWithMetadata(
  selectMetadata(contentproviders, 'TwitterSearch'),
  'contentprovider',
  this
);
```

Listing 5.3: Example of using the component instances in the application logic.

```
// Function search is called when a button is
// pressed in the application.
QtMashwareClient.prototype.search = function() {
  // Set callback for visualrest component
  this.visualrest.setCallback(this, this.visualrestCallback);

  // Download content by the tag in the search field
  // See http://visualrest.cs.tut.fi/ for the API
  this.visualrest.downloadContent(
    'q[tag]=' + this.searchField.text
  );
};
```

Lessons learned. After the mashware repository is populated with a set of components that can be mixed without restrictions, creating new applications is very rapid. The actual mashup logic can be written with only few lines of JavaScript code, and modifying it is very straightforward. Moreover, the focus of implementation efforts was shifted from interfacing with web services to creating the look and feel of the mashup. Therefore, we argue that creating compelling mashware applications with our of approach is possible for inexperienced developers as well.

As in software development in general, changing the component interfaces afterwards is cumbersome, and careful planning is therefore required beforehand. This promotes lucidity of the implementation and makes is easy to understand and debug as well. Furthermore, passing data between software components should be done in a well-defined format.

Current debugging tools support dynamically loaded code poorly, which makes the development of new application components as well as testing them together more difficult. Tracing code execution or variable values is not possible, and debugging tools are not capable of pointing out the erroneous line of code. Therefore, debug logs need to be used extensively. In addition, code needs to be written with small iterations and tested constantly.

Finally, RESTful mashware repository is an important part of the implementation. Utilizing this easy-to-use interface for discovering software components is straightforward, and it enables even applications where the software components are selected to use automatically. The repository still lacks security features, and it could be further developed to support different types of components as well as parallel components with similar functionality but different runtime environment requirements.

6

Conclusions

This Chapter concludes the dissertation by summarizing and revisiting the main results of the work. Furthermore, introduction to included publications is provided and author's contributions in publications are described. Future work that can be based on the results of this work is outlined as well.

6.1 Summary

This dissertation describes and addresses numerous issues regarding mashup ecosystems and mashup development. To achieve this, different mashup ecosystems, both implicit and explicit, are described along with specialized ecosystems with considerable limitations. Furthermore, the dissertation discusses practical issues that need to be taken into consideration when developing mashups. In addition, a novel approach to web software development – creating software as a mashup – is introduced, and a realization of such concept is described.

Chapter 2 includes a broad introduction to the Web as a mashup platform, and it illustrates possibilities of mashups, as well. First, a concept referred to as SOA, which is often used when web service-based applications are developed, is introduced. Next, different types of existing mashups are briefly covered. Web resources are the main ingredients of mashups, and therefore it is described how to access these resources. Development process of mashups with and without specialized tools is described to give an overview of the typical approach used. Some mashup patterns are available, especially on enterprise setting, and these are briefly described. Furthermore, as web

browser is not the only option for a mashup runtime, general requirements for such systems are briefly summarized.

Chapter 3 describes mashup ecosystems that are formed to allow users, mashup developers, and service providers to collaborate. To gain better understanding of mashup ecosystems, we first describe mashup ecosystems from different perspectives including user's, mashup developer's, and service provider's view on the subject. In addition, four levels of support that a service provider can offer for mashups are introduced, and mashup ecosystems are roughly categorized into explicit and implicit ecosystems. Furthermore, we present practical issues that need to be taken into account when a mashup ecosystem is formed, and present our practical implementation of a specialized ecosystem targeted at mobile multimedia mashups.

Mashup composing with general purpose web development tools is the subject of Chapter 4. Being a relative immature area, mashup development suffers from the same issues as web development in general, but it has additional issues as well, mainly because of need to access numerous interfaces and pass information between them. This leads to numerous security issues, but also other difficulties in the areas of maintainability and dependability, for instance. Furthermore, in this chapter we describe a well-engineered reference architecture for mashups that can be used as a guidance when a mashup is created. As accessing different resources is central part of any mashup implementation, we have pointed out numerous considerations on accessing both web and local resources. On mobile and embedded devices, even more careful approach to mashups development is needed mainly because of hardware and network connectivity limitations, and therefore composing mashups with such devices as a target is covered, as well.

Applying the idea of mashups to software development is described in Chapter 5. This extends the lessons learned during mashup development to software development. Even though creating large applications from software components is nothing new, web development methods and tools for creating software in such style are still immature. In this Chapter, requirements, issues, and limitations of creating web applications from downloadable components are discussed. In addition, a proof-of-concept implementation of this kind of software is described as an example.

6.2 Research Questions Revisited

This dissertation addresses three research questions. This Section revisits the questions and gives a brief summary of the main results. The first research question, which is addressed in the Chapter 3, is the following.

> **RQ1.** How to design mashup ecosystems where end-users, mashup authors, and service providers collaborate?

This dissertation describes mashup ecosystems where end-users, mashup developers, and service providers collaborate. In addition to ecosystem consisting of all available web services and mashups built on top of them, subsets of this global ecosystem, which can be roughly categorized into explicit and implicit ecosystems, can be considered as well. The three ecosystem actors have different views on mashup ecosystems, and this dissertation describes these different perspectives. In addition we identified four levels of support that service providers can offer for mashup development.

Furthermore, when mashup ecosystems are formed, numerous practical issues need to be taken into account. The dissertation points out how interface compatibility, security, and legislation are the fundamental challenges of mashup ecosystems. Moreover, one has to determine whether the mashup functionality resides on the server-end or the client-end, or is a hybrid model combining both approaches followed. To elaborate how specialized mashup ecosystems can be created, we described an example implementation targeted at enabling mobile multimedia mashups. The ecosystem follows a hybrid model where an aggregator server hides the complexity of accessing numerous web services, and the mashup functionality resides on the client-end.

The second research question – addressed in Chapter 4 – is the following.

> **RQ2.** How to solve problems related to client-side mashup development on desktop and mobile devices?

Often special purpose tools are proposed to make mashup development easier, but here, we have used traditional web development tools to create mashups instead of such tools that typically limit mashup developer's possibilities. Mashup design has been often considered as opportunistic activity, but our view is different. To foster maintainability, modifiability, and reusability of mashup implementations, we describe

a reference architecture for mashups that can be used as a fundamental structure when composing client-side mashups with manual development methods. Instead of considering mashup development as an ad hoc activity, here it is regarded as serious engineering effort that benefits from careful design and implementation. Consequently, we describe a set of general design principles that should be followed in the process of mashup creation. Accessing different interfaces play an important role in mashups, and in this dissertation we describe what should be taken into account when interfacing with web and local services. Special requirements that are faced when mashups are developed for mobile devices are described, too. In addition, to address security related issues of mashups, we point out mashup attack scenarios and security practices that should be followed to increase mashup security.

The third research question, which is addressed in the Chapter 5, is the following.

RQ3. How to realize software as a mashup?

The last research question considers realizing software composed out of executable code components. We pointed out that this approach to software development – sometimes referred to as mashware – fits very well for the current way web is used, and the combination of emerging technologies included into Web 3.0 definition will change the Web even more suitable platform for mashware. Despite reusing software components in traditional application development has been investigated for decades, there have not been widespread systems that would enable mashware. Consequently, research is needed in numerous areas of mashware engineering. This dissertation describes the most relevant issues, but further work to solve these issues is left for future research efforts. To illustrate possibilities of mashware development, we describe a proof-of-concept implementation where the reference mashup architecture is applied. However, a special purpose runtime environment is used in the implementation, and therefore numerous problems that would be faced when a web browser would be used, are avoided.

6.3 Research Contributions Revisited

The contributions of this dissertation were briefly summarized in Section 1.4, and here we compare the main results to the existing work presented in the literature. The

main results about mashup ecosystems, mashup composing, and software created as a mashup are revisited in the following.

Mashup Ecosystems. In [99] Yu and Woodart have considered mashup ecosystem that consists of all available mashups and services indexed in their data source ProgrammableWeb, and growth of this ecosystem is modeled by Weiss and Gangadharan in [100]. Instead of considering this kind of global ecosystem, we have focused on smaller subsystems consisting only few selected services and mashups build on top of them. This approach is similar to Bosch's [98], where software ecosystems are considered as controllable entities and categorized. Bosch mentions mashup ecosystems under "End-User Programming Software Ecosystems" category, where ecosystems build with mashup tools such as Yahoo! Pipes are discussed.

Our work focuses on addressing practical issues that mashup ecosystem authors need to deal with when such systems are created either explicitly or implicitly. We describe mashup ecosystems from perspectives of mashup user, mashup author, and service provider. We identify four levels of support that a service provider can offer for mashup developers. Instead of considering only tool-oriented mashup ecosystems, we have addressed ecosystems that are build with general purpose web development tools and methods. As a technical contribution and an example ecosystem, we present a mashup ecosystem for aggregating videos from multiple web video services.

Mashup development. Mashup development has been widely researched, and numerous tools for mashup composing have been created and extensively reported in [68, 69, 70], for instance. Mashup achitecture has been studied by López et al. [117], and their layered server-side approach is targeted at creating mashups in enterprise setting. The example mashup presented in [117] resembles a portal with connected portlets. Requirement to follow established software engineering principles in mashup development is mentioned in [114], which is a case study pointing out the need for careful engineering when developing mashups. Similarly, [52] identifies numerous mashup development challenges and addresses them with a mashup architecture and pipeline-based tool for mashup composing. Furthermore, Liu et al. have studied component-based approach for thematic mashups in [115].

Instead of focusing on a tailored solution for mashup composing, our work uses general purpose web development tools to create mashups. To aid the development of mashups, we describe a reference architecture for client-side mashups and introduce

a set of general design principles for mashup composing. Furthermore, we address numerous issues about communicating with web services and accessing local resources in embedded devices. In purpose of fostering development of mashups on mobile devices, we listed a number of considerations to be taken into account.

Software as a mashup. The idea of software developed as a mashup has been introduced by Taivalsaari in [16], where mashware is described having a role in the future web. Numerous areas where further research is necessary are listed by Mikkonen and Taivalsaari in [17]. Our work reviews technologies included under term "Web 3.0" and describes how those will promote role of software developed as a mashup in the future web. As a technical contribution, we have developed the first laboratory prototype of a mashware application.

6.4 Introduction to the Included Publications

This thesis consists of the introduction part and eight included publications. The publications are described below.

- **Publication I.** Publication I discusses software ecosystems that are created when mashups are developed. The paper points out that these ecosystems can be implicit or explicit. General ecosystem model consisting of mashup users, service providers, and mashup authors is described, and four levels of support that service providers can offer for mashups are identified. Furthermore, the paper discusses the different breeds and types of mashup ecosystems that have been introduced. In addition, the publication identifies challenges that mashup ecosystem confronts in the areas of cloud infrastructure, web services, legal issues, and tool support.

- **Publication II.** Publication II includes a literature review about mashup ecosystems, mashup security, and end-user programming solutions. Moreover, the publication describes numerous fundamental challenges that are encountered when a mobile multimedia mashup ecosystem is realized. The ecosystem targets at providing consistent user experience on different end user devices. This constitutes a basis for the work that is described in Publication III.

- **Publication III.** Publication III presents an aggregator-based mashup ecosystem architecture that is a realization of the ecosystem described in Publication II.

The architecture is analyzed, and a discussion is given on how the goals of the system are achieved. The publication also includes a description of a client-side application that utilizes the underlying mashup ecosystem.

- **Publication IV.** Existing software environments and tools as well as numerous practical examples of mashups targeted at mobile devices are presented in Publication IV. These mashups are built with a special runtime environment, Lively for Qt, which allows to run the mashups even on devices with modest processing power. Publication presents the example mashups in detail and points out numerous practical issues in usability, connectivity, and performance of mashups, as well as general issues such as lack of well-defined interfaces and absence of a fine-grained security model.

- **Publication V.** Two mashup runtime environments for embedded systems following procedural and declarative development style are discussed in Publication V. This publication points out how using a specialized runtime benefits mashups especially on embedded devices where using a web browser might be an overkill for performance. The publication discusses how a mashup environment that accesses the user context and local resources, and combines them with web content can be implemented. The approach used for composing such mashups is similar to procedural development environment of Publication IV, but now a declarative Qt-based environment is used as well. Here, both environments are used to compose mashups that have ability to use and share local data and device peripherals. Furthermore, as a practical implementation, a simple application that accesses the device's GPS data and combines it with normal web data is presented. This approach can be generalized for different combinations of technologies and devices, provided that adequate means of communication are available.

- **Publication VI.** Thirteen guidelines for mashup developers are described in Publication VI. This set of guidelines is based on hand-on experiences in developing various client-side mashups, and it can help mashup developers to choose the right methods when building mashups. The guidelines are in the areas of mashup design and interfacing with web services. In addition, two guidelines that handle broader issues are presented. Furthermore, the publication describes a reference

architecture for client-side mashups and compares it with traditional web architectures. This publication includes real-life example mashups, targeted at desktop and mobile devices and constructed with traditional web development methods as well as Lively for Qt, the system used in Publication IV. The guidelines and the reference architecture are explained further with the examples.

- **Publication VII.** The Web is a reforming platform, and the way it is used evolves constantly. Publication VII describes how technologies listed under an umbrella term "Web 3.0" will benefit mashup development. Being a position paper, the publication argues how mashware, software created as mashup, will be an integral part of Web 3.0, and how it plays a significant role on how software is developed in the future. In addition, the publication lists different types of mashups that are already available, or will eventually become available.

- **Publication VIII.** Finally, the first realization of a mashware application is described in Publication VIII. The paper provides an overview for mashware computing, where downloadable components form applications in piecemeal fashion. In the publication, we describe in detail how mashware components can be derived and included dynamically into an application. Furthermore, we apply the previously presented mashup reference architecture (see Publication VI) to software created as mashup, and realize a proof-of-concept implementation of such application.

6.5 Author's Contributions in Publications

The author of this thesis made contributions to the included publications as follows.

- **Publication I.** Publication I is a joint effort with Tommi Mikkonen. In this publication, the author of this dissertation did the majority of writing out of the ideas, described the mashup ecosystem characteristics, and pointed out the practical issues that mashup ecosystems need to deal with. The ideas were hatched together with the co-author Tommi Mikkonen.

- **Publication II.** In Publication II the author of this dissertation did the writing work, conducted the literature review, and presented the fundamental challenges

of mashup ecosystems. Co-authors Jarno Kallio and Tommi Mikkonen gave suggestions and feedback about the challenges that mashup ecosystems encounter in practice.

- **Publication III.** Publication III is a joint effort with Mikko Hartikainen and Jarno Kallio. This publication extends and realizes the ideas presented in publication II. The author of this dissertation wrote out the mashup ecosystem architecture description and the background study. The proposed approach to mobile mashup ecosystem was evaluated together with Mikko Hartikainen who did the practical development work of the client-side implementation.

- **Publication IV.** In Publication IV the author developed the five example mashups. The author gleaned and wrote out the lessons learned about these experiments together with co-author Feetu Nyrhinen. Applicability of the Lively for Qt platform to mobile mashup development was analyzed together with Feetu Nyrhinen. The Lively for Qt platform was originally a dynamic programming environment developed by two other co-authors Tommi Mikkonen and Antero Taivalsaari.

- **Publication V.** Publication V is a joint effort with Tommi Mikkonen. The author of this dissertation developed the proof-of-concept implementation using both procedural and declarative approach and evaluated as well as compared the two approaches.

- **Publication VI.** Publication VI includes guidelines and reference architecture for mashups. The guidelines were generated together with co-author Feetu Nyrhinen. The reference architecture for mashups was created together with Tommi Mikkonen. The example mashups where the guidelines and the architecture are applied are implemented by the author of this dissertation.

- **Publication VII.** The author of this dissertation is the sole author in Publication VII. In this publication, Tommi Mikkonen had an advisory role and provided comments that lead to improvements.

- **Publication VIII.** Publication VIII is a joint effort with Tommi Mikkonen. The author derived the mashware components used in the proof-of-concept implementation, designed a system that could be used to deliver the components, created the example application, and evaluated the selected approach. Idea of the software as an on-demand service was further developed together with co-author Tommi Mikkonen, based on his previous research on the subject.

6.6 Future Work

Recent development in web browser capabilities, such as HTML5, WebGL, and other new interfaces, have made browser-based approach towards cross-platform application development more appealing. In addition, new fine grained proposals for browser security, such as CORS, are removing obstacles to utilizing different web services in a single application. Poor performance of web applications is going down in history as well, when JavaScript engines in browsers are gaining performance, and browser graphics are turning to be hardware accelerated on desktop and even on handheld devices. When these developments have been analyzed, it has been argued that this trend towards web-based software development will become even stronger in the future [12]. Therefore, also mashup development will focus on using existing browser-based solutions as runtime platform instead of native applications or other special purpose environments. However, as has been pointed out in [167], to foster this development, research efforts are necessary on education of dynamic programming languages, software deployment models, and web software testing. When the goal is to develop mashups and mashware, these research efforts can be summarized as follows.

- Mashups are programmed in evolutionary, exploratory programming style associated with dynamic languages [168], which is not familiar for developers that are used to create software with static programming languages. Therefore, research in software education is needed to be able to provide necessary skills for developers creating mashups and mashware.

- Software deployment practices for mashware are entirely different from conventional binary software. Executable software components in a mashware application can be deployed instantly all over the world, and this enables “nano releases”

of components – releases that can occur multiple times per day. One of the fundamental challenges of the deployment model of mashware is defining a model that supports software components nano release cycles and constant development, potentially without need to restart the mashware application when an update takes place.

- Mashups and mashware consists of pieces that are loaded dynamically without any static compilation, type checking, or linking. Because of this, and other issues related to such software, testing needs to be done in completely different manner than traditional software. Therefore, interesting research challenge is how to establish methodologies for testing mashups and mashware.

In addition to solving the issues listed above, dealing with conflicting terms of services in mashup ecosystems is a problem that needs to be addressed. Numerous web services provide an interface for their system, but some actions that are necessary when a consumer-oriented mashup is build can be forbidden in the terms of the service. For instance, caching the content on an application server may be prohibited. Method for evaluating possibility to combine two or more services in a mashup from perspective of legistlation is therefore interesting subject for further research.

Creating software as a mashup is very promising approach to web software development. If fits very well in the way the Web is used and – as pointed out in publication VII – this trend seems to gain strength as new web technologies emerge. Therefore, research about engineering practises in this evolving setting is in our future scope.

Glossary

API	*Application Programming Interface*. For complete description in the context of this thesis see "web interface".
Composite application	*Composite application* is built by combining multiple existing functions into a new application. Information used in composite applications is typically of business sources.
CSS	*Cascading Style Sheets* is a style sheet language used for describing the look and formatting of a document written in a markup language such as HTML.
DHTML	*Dynamic HTML* is an umbrella term for a collection of technologies (typically HTML, JavaScript, CSS, and DOM) used together to create interactive and animated web sites.
DOM	*Document Object Model* defines a standard way for accessing and manipulating HTML, XHTML, and XML documents.
HTML	*HyperText Markup Language* is the main markup language for displaying web pages and other information that can be displayed in a web browser.
Hyperlink	*Hyperlink* is a word, phrase, or image that you can click on to access to a new web document or a new section within the current document.
Hypertext	*Hypertext* is text with references (hyperlinks) to other documents that the reader can immediately access.

JavaScript	*JavaScript* is a scripting language commonly implemented as part of a web browser.
JSON	*JavaScript Object Notation* is a lightweight data-interchange format.
Mashup	*Mashup* is web application that combines resources over web interfaces into an integrated application that has increased value for the end-user.
Mashup ecosystem	*Mashup ecosystem* is a software ecosystem consisting of mashups and web services.
Mashware	*Mashware* application is a web application that is created from software components downloaded over the Web.
Portal	A web *portal* is a web site that brings information together from diverse sources in a uniform way.
Programmatic interface	*Programmatic interface* is a web interface that is used with a programming library.
REST	*Representational State Transfer* is a style of software architecture for distributed systems such as the Web.
RPC	*Remote Procedure Call* is an inter-process communication that allows a computer program to cause a subroutine or procedure to execute on another computer on a shared network without the programmer explicitly coding the details for this remote interaction.
Same origin policy	*Same origin policy* restricts how a document or script loaded from one origin can interact with a resource from another origin.
Screen scraping	See “web scraping”.
Service-level Agreement	*Service-level agreement* (SLA) is a part of a service contract where a service is formally defined.
Service-Oriented Architecture	*Service-oriented architecture* (SOA) is a set of principles and methodologies for designing and developing software in the form of interoperable services

Situational mashup	A mashup developed as situational application – an application that is created for a narrow group of users with unique needs.
Software ecosystem	*Software ecosystem* is a set of businesses functioning as a unit and interacting with a shared market for software and services, together with relationships among them.
Terms of Service	*Terms of Service* (TOS, Terms of Use, Terms and Conditions) are rules which one must agree to abide by in order to use a service.
Web	*The Web* (World Wide Web, WWW) is a system of interlinked hypertext documents accessed via the Internet.
Web feed	*Web feed*, such as RSS or Atom feed, is a data format used for providing users with frequently updated content.
Web interface	*Web interface* can be used by an application to access a web service.
Web scraping	*Web scraping* refers to parsing data from a representation intended to be read by a human, also referred to as "screen scraping".
Widget	*Widget* (control) is an element of a graphical user interface that displays an information arrangement changeable by the user, such as a window or a text box.
Wiki	*Wiki* is a website which allows its users to add, modify, or delete its content via a web browser usually using a simplified markup language or a rich-text editor.

References

[1] B. Taylor. **Mapping your way**. Google's Official Blog, February 2005. Available online: `http://googleblog.blogspot.fi/2005/02/mapping-your-way.html`.

[2] T. H. Nelson. **Complex information processing: A file structure for the complex, the changing and the indeterminate**. In *Proceedings of the 1965 20th national conference*, ACM '65, pages 84–100, New York, NY, USA, 1965. ACM.

[3] D. Raggett, A. Le Hors, and I. Jacobs. **HTML 4.01 specification**. W3C, December 1999. Available online: `http://www.w3.org/TR/html4`.

[4] D. Flanagan. *JavaScript: The Definitive Guide*. O'Reilly Media, Inc., Sebastopol, CA, USA, third edition, 1998.

[5] E.C.M.A. International. *ECMA-262: ECMAScript language specification*. European Association for Standardizing Information and Communication Systems, Geneva, Switzerland, third edition, December 1999.

[6] E. J. Etemad. **Cascading Style Sheets (CSS)**. W3C, May 2011. Available online: `http://www.w3.org/TR/CSS/`.

[7] W3C DOM Interest Group. **Document Object Model**. W3C, January 2005. Available online: `http://www.w3.org/DOM/`.

[8] Adobe Systems Inc. **Adobe Flash Player**. Available online: `http://www.adobe.com/software/flash/about/`.

[9] Oracle Corporation. **Rich internet applications deployment advice: Applet deployment**. Available online: `http://docs.oracle.com/javase/6/docs/technotes/guides/jweb/applet/applet_deployment.html`.

[10] J. J. Garrett. **Ajax: A new approach to web applications**, February 2005. Available online: `http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications`.

[11] R. Berjon, R. Leithead, E. D. Navara, E. O'Connor, and S. Pfeiffer. **HTML 5 specification**. W3C, December 2012. Available online: `http://www.w3.org/TR/2012/CR-html5-20121217/`.

[12] M. Anttonen, A. Salminen, T. Mikkonen, and A. Taivalsaari. **Transforming the Web into a real application platform: New technologies, emerging trends and missing pieces**. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 800–807, New York, NY, USA, 2011. ACM.

[13] A. Taivalsaari, T. Mikkonen, M. Anttonen, and A. Salminen. **The death of binary software: End-user software moves to the Web**. In *Proceedings of the 2011 Ninth International Conference on Creating, Connecting and Collaborating through Computing*, C5'11, pages 17–23, Washington, DC, USA, 2011. IEEE Computer Society.

[14] D. G. Messerschmitt and C. Szyperski. *Software Ecosystem: Understanding an Indispensable Technology and Industry*. MIT Press, Cambridge, MA, USA, 2003.

[15] J. Salo, T. Aaltonen, and T. Mikkonen. **MashReduce: Server-side mashups for mobile devices**. In *Proceedings of the 6th international conference on Advances in grid and pervasive computing*, GPC'11, pages 168–177, Berlin, Heidelberg, 2011. Springer-Verlag.

[16] A. Taivalsaari. **Mashware: The future of web applications**. Technical report, Sun Microsystems, Inc., Mountain View, CA, USA, 2009. Available online: `https://labs.oracle.com/techrep/2009/smli_tr-2009-181.pdf`.

[17] T. Mikkonen and A. Taivalsaari. **The mashware challenge: Bridging the gap between web development and software engineering**. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, FoSER'10, pages 245–250, New York, NY, USA, 2010. ACM.

[18] P. Järvinen. **Research questions guiding selection of an appropriate research method**. In *European Conference on Information Systems*, pages 124–131, Vienna University of Economics and Business Administration, July 2000. Available online: `http://www.cs.uta.fi/reports/dsarja/D-2004-5.pdf`.

[19] M. K. Sein, O. Henfridsson, S. Purao, M. Rossi, and R. Lindgren. **Action design research**. *MIS Quarterly*, **35**(2), June 2011.

[20] C. Bizer, T. Heath, and T. Berners-Lee. **Linked data – The story so far**. *International Journal on Semantic Web and Information Systems (IJSWIS)*, **5**(3):1–22, March 2009.

[21] J. Yu, B. Benatallah, F. Casati, and F. Daniel. **Understanding mashup development**. *IEEE Internet Computing*, **12**(5):44–52, 2008.

[22] J. Ruderman. **Same-origin policy for JavaScript**. Mozilla Developer Network. Available online: `https://developer.mozilla.org/en-US/docs/Same_origin_policy_for_JavaScript?redirect=no`.

[23] A. Barth. **The Web origin concept**. IETF, RFC 6454 (Proposed Standard), December 2011. Available online: `http://www.ietf.org/rfc/rfc6454.txt`.

[24] C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell. **Protecting browser state from web privacy attacks**. In *Proceedings of the 15th international conference on World Wide Web*, WWW '06, pages 737–744, New York, NY, USA, 2006. ACM.

[25] C. Karlof, U. Shankar, J. D. Tygar, and D. Wagner. **Dynamic pharming attacks and locked same-origin policies for web browsers**. In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 58–71, New York, NY, USA, 2007. ACM.

[26] T. Oda, G. Wurster, P. C. van Oorschot, and A. Somayaji. **SOMA: Mutual approval for included content in web pages**. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 89–98, New York, NY, USA, 2008. ACM.

[27] R. Thompson. **Web services for remote portlets specification v2.0**. OASIS Standard, June 2007. Available online: `http://docs.oasis-open.org/wsrp/v2/wsrp-2.0-spec.html`.

[28] A. Abdelnur, E. Chien, and S. Hepper. **JSR-168 portlet specification**. Java Community Process Program, October 2003. Available online: `http://jcp.org/en/jsr/detail?id=168`.

[29] S. Hepper. **JSR-286 portlet specification 2.0**. Java Community Process Program, June 2008. Available online: `http://www.jcp.org/en/jsr/detail?id=286`.

[30] J. Polgar. **Using WSRP 2.0 with JSR 168 and 286 Portlets**. *International Journal of Web Portals (IJWP)*, **2**(1):45–57, 2010.

[31] S. Peenikal. **Mashups and the enterprise**. MphasiS, White paper, September 2009. Available online: `http://www.mphasis.com/pdfs/Mashups_and_the_Enterprise.pdf`.

[32] C. Keyser. **Composite applications: The new paradigm**. Microsoft Developer Network. Available online: `http://msdn.microsoft.com/en-us/library/bb266335.aspx`.

[33] A. Gutmans. **PHP: Supporting the new paradigm of situational and composite web applications**. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 707–707, New York, NY, USA, 2006. ACM.

[34] R. Fernández, D. Lizcano, S. Ortega, and J. Soriano. **Towards a user-centered composition system for service-based composite applications**. In *Proceedings of the 11th International Conference on Information Integration and Web-based Applications & Services*, iiWAS '09, pages 321–330, New York, NY, USA, 2009. ACM.

[35] D. Lizcano, J. Soriano, M. Reyes, and J. J. Hierro. **EzWeb/FAST: Reporting on a successful mashup-based solution for developing and deploying composite applications in the upcoming web of services**. In *Proceedings of the 10th International Conference on Information Integration and Web-based Applications & Services*, iiWAS '08, pages 15–24, New York, NY, USA, 2008. ACM.

[36] A. H. H. Ngu, M. P. Carlson, Q. Z. Sheng, and Hye young P. **Semantic-based mashup of composite applications**. *IEEE Transactions on Services Computing*, **3**(1):2–15, January 2010.

[37] T. Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[38] M. P. Papazoglou and D. Georgakopoulos. **Introduction: Service-oriented computing**. *Communications of the ACM*, **46**(10):24–28, October 2003.

[39] R. Chinnici, J.-J. Moreau, A. Ryman, and S. Weerawarana. **Web Services Description Language (WSDL) Version 2.0 Part 1: Core language**. W3C, June 2007. Available online: `http://www.w3.org/TR/2007/REC-wsdl20-20070626`.

[40] L. Clement, A. Hately, C. von Riegen, and T. Rogers. **UDDI version 3.0.2**. OASIS Standard, October 2004. Available online: `http://uddi.org/pubs/uddi_v3.htm`.

[41] J. McGovern, S. Tyagi, M. Stevens, and S. Mathew. *Java Web Services Architecture*. Morgan Kaufmann Publishers, Burlington, MA, USA, 2003.

[42] T. Nestler. **Towards a mashup-driven end-user programming of SOA-based applications**. In *Proceedings of the 10th International Conference on Information Integration and Web-based Applications & Services*, iiWAS '08, pages 551–554, New York, NY, USA, 2008. ACM.

[43] J. McKendrick. **Mashup vs. SOA app: What's the difference?** ZDNet Article, June 2006. Available online: `http://www.zdnet.com/blog/service-oriented/mashup-vs-soa-app-whats-the-difference/647`.

[44] S. Watt. **Mashups – The evolution of the SOA, Part 1: Web 2.0 and foundational concepts**. IBM developerWorks, October 2007. Available online: `http://www.ibm.com/developerworks/webservices/library/ws-soa-mashups/`.

[45] A. Koschmider, V. Torres, and V. Pelechano. **Elucidating the mashup hype: Definition, challenges, methodical guide and tools for mashups**. In *Proceedings of 18th International World Wide Web Conference, 2nd Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web*, MEM '09, pages 1–9, New York, NY, USA, 2009. ACM.

[46] Nan Z. and M. B. Rosson. **What's in a mashup? And why? Studying the perceptions of web-active end-users**. In *Proceedings of the 2008 IEEE Symposium on Visual Languages and Human-Centric Computing*, VLHCC '08, pages 31–38, Washington, DC, USA, 2008. IEEE Computer Society.

[47] S. Bitzer, S. Ramroth, and M. Schumann. **Mashups as an architecture for knowledge management systems**. In *Proceedings of the 42nd Hawaii International Conference on System Sciences*, HICSS '09, pages 1–10. IEEE Computer Society, 2009.

[48] V. Hoyer and M. Fischer. **Market overview of enterprise mashup tools**. In *Proceedings of the 6th International Conference on Service-Oriented Computing*, ICSOC '08, pages 708–721, Berlin, Heidelberg, 2008. Springer-Verlag.

[49] V. Hoyer, K. Stanoesvka-Slabeva, T. Janner, and C. Schroth. **Enterprise mashups: Design principles towards the long tail of user needs**. In *Proceedings of IEEE International Conference on Services Computing*, SCC '08, pages 601–602, July 2008.

[50] A. Jhingran. **Enterprise information mashups: Integrating information, simply**. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06, pages 3–4. VLDB Endowment, 2006.

[51] E. Ort, S. Brydon, and M. Basler. **Mashups styles, part 1: Server-side mashups**. Technical report, Sun Microsystems, Inc., Mountain View, CA, USA, 2007.

[52] B. Biörnstad and C. Pautasso. **Let it flow: Building mashups with data processing pipelines**. In E. Nitto and M. Ripeanu, editors, *Service-Oriented Computing – ICSOC 2007 Workshops*, **4907** of *Lecture Notes in Computer Science*, pages 15–28. Springer-Verlag, Berlin, Heidelberg, 2009.

[53] E. Ort, S. Brydon, and M. Basler. **Mashups styles, part 2: Client-side mashups**. Technical report, Sun Microsystems, Inc., Mountain View, CA, USA, 2007.

[54] D. Crockford. **The `application/json` Media Type for JavaScript Object Notation (JSON)**. IETF, RFC 4627, July 2006.

[55] J. López, F. Bellas, A. Pan, and P. Montoto. **A component-based approach for engineering enterprise mashups**. In *Proceedings of the 9th International Conference on Web Engineering*, ICWE '9, pages 30–44, Berlin, Heidelberg, 2009. Springer-Verlag.

[56] Open Mashup Alliance. **Enterprise Mashup Markup Language (EMML)**. OMA EMML Documentation, September 2009. Available online: `http://www.openmashup.org/omadocs/v1.0/index.html`.

[57] J. Clark and S. DeRose. **XML path language (XPath) 1.0**. W3C, November 1999. Available online: `http://www.w3.org/TR/xpath/`.

[58] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernández, M. Kay, J. Robie, and J. Siméon. **XML path language (XPath) 2.0**. W3C, December 2010. Available online: `http://www.w3.org/TR/xpath20/`.

[59] C. Shirky. **Situated software**, 2004. First published March 30, 2004 on the "Networks, Economics, and Culture" mailing list. Available online: `http://www.shirky.com/writings/situated_software.html`.

[60] L. Richardson and S. Ruby. *Restful Web Services*. O'Reilly Media, Inc., Sebastopol, CA, USA, first edition, 2007.

[61] R. T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[62] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. **Hypertext Transfer Protocol – HTTP/1.1**. IETF, RFC 2616, June 1999. Available online: `http://www.ietf.org/rfc/rfc2616.txt`.

[63] N. Freed and N. Borenstein. **Multipurpose Internet Mail Extensions (MIME) Part two: Media types**. IETF, RFC 2046, November 1996. Updated by RFCs 2646, 3798, 5147. Available online: `http://www.ietf.org/rfc/rfc2046.txt`.

[64] J. E. White. **High-level framework for network-based resource sharing**. IETF, RFC 707, December 1975. Available online: `http://www.ietf.org/rfc/rfc707.txt`.

[65] S. Vinoski. **Convenience over correctness**. *IEEE Internet Computing*, **12**(4):89–92, July 2008.

[66] F. Daniel, M. Matera, and M. Weiss. **Next in mashup development: User-created apps on the Web**. *IT Professional*, **13**(5):22–29, September 2011.

[67] W. Al Sarraj and O. De Troyer. **Web mashup makers for casual users: A user experiment**. In *Proceedings of the 12th International Conference on Information Integration and Web-based Applications & Services*, iiWAS '10, pages 239–246, New York, NY, USA, 2010. ACM.

[68] C. Cappiello, M. Matera, M. Picozzi, G. Sprega, D. Barbagallo, and C. Francalanci. **DashMash: A mashup environment for end-user development**. In *Proceedings of the 11th international conference on Web engineering*, ICWE'11, pages 152–166, Berlin, Heidelberg, 2011. Springer-Verlag.

[69] J. Wong and J. I. Hong. **Making mashups with Marmite: Towards end-user programming for the Web**. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '07, pages 1435–1444, New York, NY, USA, 2007. ACM.

[70] J. Lin, J. Wong, J. Nichols, A. Cypher, and T. A. Lau. **End-user programming of mashups with Vegemite**. In *Proceedings of the 14th international conference on Intelligent user interfaces*, IUI '09, pages 97–106, New York, NY, USA, 2009. ACM.

[71] P. Baglietto, F. Cosso, M. Fornasa, S. Mangiante, M. Maresca, A. Parodi, and M. Stecca. **Always-on distributed spreadsheet mashups**. In *Proceedings of the 3rd and 4th International Workshop on Web APIs and Services Mashups*, Mashups '09/'10, pages 8:1–8:8, New York, NY, USA, 2010. ACM.

[72] W. Kongdenfha, B. Benatallah, J. Vayssière, R. Saint-Paul, and F. Casati. **Rapid development of spreadsheet-based web mashups**. In *Proceedings of the 18th international conference on World wide web*, WWW '09, pages 851–860, New York, NY, USA, 2009. ACM.

[73] R. Tuchinda, P. Szekely, and C. A. Knoblock. **Building mashups by example**. In *Proceedings of the 13th international conference on Intelligent user interfaces*, IUI '08, pages 139–148, New York, NY, USA, 2008. ACM.

[74] G. Wang, S. Yang, and Y. Han. **A spreadsheet-like construct for streamlining and reusing mashups**. In *Proceedings of The 9th International Conference for Young Computer Scientists*, ICYCS '08, pages 880–885, Washington, DC, USA, November 2008. IEEE Computer Society.

[75] H. Lin, G. Wang, P. Zhang, J. Wang, and Y. Han. **A two-level programming model based on spreadsheet and data flow chart**. In *Proceedings of the 7th Web Information Systems and Applications Conference*, WISA '10, pages 39–42, Washington, DC, USA, August 2010. IEEE Computer Society.

[76] Z. Pan, H. Tang, M. Ge, and C. Zhang. **A framewok of spreadsheet-based Web mashup**. In *Proceedings of International Conference on Computer Science and Service System*, CSSS '11, pages 970–973, Washington, DC, USA, June 2011. IEEE Computer Society.

[77] G. Wang, S. Yang, and Y. Han. **Mashroom: End-user mashup programming using nested tables**. In *Proceedings of the 18th international conference on World wide web*, WWW '09, pages 861–870, New York, NY, USA, 2009. ACM.

[78] J. Cao, K. Rector, T. H. Park, S. D. Fleming, M. Burnett, and S. Wiedenbeck. **A debugging perspective on end-user mashup programming**. In *Proceedings of the 2010 IEEE Symposium on Visual Languages and Human-Centric Computing*, VLHCC '10, pages 149–156, Washington, DC, USA, 2010. IEEE Computer Society.

[79] R. Ennals, E. Brewer, M. Garofalakis, M. Shadle, and P. Gandhi. **Intel Mash Maker: Join the Web**. *ACM SIGMOD Record*, **36**(4):27–33, December 2007.

[80] K. T. Stolee and S. Elbaum. **Refactoring pipe-like mashups for end-user programmers**. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 81–90, New York, NY, USA, 2011. ACM.

[81] G. Leshed, E. M. Haber, T. Matthews, and T. Lau. **CoScripter: Automating & sharing how-to knowledge in the enterprise**. In *Proceedings of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, CHI '08, pages 1719–1728, New York, NY, USA, 2008. ACM.

[82] R. Guo, B. B. Zhu, M. Feng, A. Pan, and B. Zhou. **Compoweb: A component-oriented web architecture**. In *Proceedings of the 17th international conference on World Wide Web*, WWW '08, pages 545–554, New York, NY, USA, 2008. ACM.

[83] D. Crockford. **The `<module>` tag**, October 2006. Available online: `http://json.org/module.html`.

[84] J. Howell, C. Jackson, H. J. Wang, and X. Fan. **MashupOS: Operating system abstractions for client mashups**. In *Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, pages 16:1–16:7, Berkeley, CA, USA, 2007. USENIX Association.

[85] S. Crites, F. Hsu, and H. Chen. **OMash: Enabling secure web mashups via object abstractions**. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 99–108, New York, NY, USA, 2008. ACM.

[86] C. Jackson and H. J. Wang. **Subspace: Secure cross-domain communication for web mashups**. In *Proceedings of the 16th international conference on World Wide Web*, WWW '07, pages 611–620, New York, NY, USA, 2007. ACM.

[87] F. De Keukelaere, S. Bhola, M. Steiner, S. Chari, and S. Yoshihama. **SMash: Secure component model for cross-domain mashups on unmodified browsers**. In *Proceedings of the 17th international conference on World Wide Web*, WWW '08, pages 535–544, New York, NY, USA, 2008. ACM.

[88] T. Berners-Lee, R. Fielding, and L. Masinter. **Uniform Resource Identifier (URI): Generic syntax**. IETF, RFC 3986, January 2005. Available online: `http://www.ietf.org/rfc/rfc3986.txt`.

[89] J. Magazinius, A. Askarov, and A. Sabelfeld. **A lattice-based approach to mashup security**. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS'10, pages 15–23, New York, NY, USA, 2010. ACM.

[90] S. Ikeda, T. Nagamine, and T. Kamada. **Application framework with demand-driven mashup for selective browsing**. In *Proceedings of the 10th International Conference on Information Integration and Web-based Applications & Services*, iiWAS'08, pages 33–40, New York, NY, USA, 2008. ACM.

[91] R. Hashimoto, N. Ueno, and M. Shimomura. **A design of usable and secure access-control APIs for mashup applications**. In *Proceedings of the 5th ACM workshop on Digital identity management*, DIM'09, pages 31–34, New York, NY, USA, 2009. ACM.

[92] M. Ogrinz. *Mashup Patterns: Designs and Examples for the Modern Enterprise*. Addison-Wesley Professional, first edition, 2009.

[93] T. Mikkonen, A. Taivalsaari, and M. Terho. **Lively for Qt: A platform for mobile web applications**. In *Proceedings of the 6th International Conference on Mobile Technology, Application & Systems*, Mobility '09, pages 24:1–24:8, New York, NY, USA, 2009. ACM.

[94] H. Saiedian and D. Broyle. **Security vulnerabilities in the same-origin policy: Implications and alternatives**. *IEEE Computer*, **44**(9):29–36, September 2011.

[95] A. van Kesteren. **Cross-Origin Resource Sharing**. W3C, April 2012. Available online: `http://www.w3.org/TR/cors/`.

[96] C. Marrin. **WebGL specification 1.0**. Khronos Group, February 2011. Available online: `https://www.khronos.org/registry/webgl/specs/1.0/`.

[97] S. Aghaee and C. Pautasso. **Mashup development with HTML5**. In *Proceedings of the 3rd and 4th International Workshop on Web APIs and Services Mashups*, Mashups '09/'10, pages 10:1–10:8, New York, NY, USA, 2010. ACM.

[98] J. Bosch. **From software product lines to software ecosystems**. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, pages 111–119, Pittsburgh, PA, USA, 2009. Carnegie Mellon University.

[99] S. Yu and C. J. Woodard. **Innovation in the programmable Web: Characterizing the mashup ecosystem**. In G. Feuerlicht and W. Lamersdorf, editors, *Workshop Proceedings of the 6th International Conference on Service Oriented Computing*, ICSOC '08, pages 136–147. Springer-Verlag, Berlin, Heidelberg, 2009.

[100] M. Weiss and S. Sari. **Evolution of the mashup ecosystem by copying**. In *Proceedings of the 3rd and 4th International Workshop on Web APIs and Services Mashups*, Mashups '09/'10, pages 11:1–11:7, New York, NY, USA, 2010. ACM.

[101] J. Cao, Y. Riche, S. Wiedenbeck, M. Burnett, and V. Grigoreanu. **End-user mashup programming: Through the design lens**. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 1009–1018, New York, NY, USA, 2010. ACM.

[102] M. Weiss and G. R. Gangadharan. **Modeling the mashup ecosystem: Structure and growth**. *R&D Management*, **40**(1):40–49, 2010.

[103] L. F. Cooper and S. Lee. **Mashups in the enterprise IT environment**. Technical report, BizTechReports.Com, Rockville, MD, USA, 2010. Available online: `http://www.jackbe.com/resources/whitepapers-ebooks`.

[104] E. Hammer-Lahav. **The OAuth 1.0 protocol**. IETF, RFC 5849 (Informational), April 2010. Available online: `http://www.ietf.org/rfc/rfc5849.txt`.

[105] I. Khan, M. Nauman, M. Alam, and F. Aziz. **SAuthMash: Mobile agent based self authorization in mashups**. In *Proceedings of the 7th International Conference on Frontiers of Information Technology*, FIT '09, pages 41:1–41:6, New York, NY, USA, 2009. ACM.

[106] S. Zarandioon, D. Yao, and V. Ganapathy. **Privacy-aware identity management for client-side mashup applications**. In *Proceedings of the 5th ACM workshop on Digital identity management*, DIM'09, pages 21–30, New York, NY, USA, 2009. ACM.

[107] M. Alam, X. Zhang, K. Khan, and G. Ali. **xDAuth: A scalable and lightweight framework for cross domain access control and delegation**. In *Proceedings of the 16th ACM symposium on Access control models and technologies*, SACMAT '11, pages 31–40, New York, NY, USA, 2011. ACM.

[108] S. Deering and R. Hinden. **Internet Protocol, version 6 (IPv6) specification**. IETF, RFC 2460 (Draft Standard), December 1998. Updated by RFCs 5095, 5722, 5871, 6437. Available online: `http://www.ietf.org/rfc/rfc2460.txt`.

[109] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. **HTTP authentication: Basic and digest access authentication**. IETF, RFC 2617 (Draft Standard), June 1999. Available online: `http://www.ietf.org/rfc/rfc2617.txt`.

[110] W. Diffie and M. E. Hellman. **New directions in cryptography**. *IEEE Transactions on Information Theory*, **22**(6):644–654, November 1976.

[111] C. Neuman, T. Yu, S. Hartman, and K. Raeburn. **The Kerberos network authentication service (V5)**. IETF, RFC 4120 (Proposed Standard), July 2005. Updated by RFCs 4537, 5021, 5896, 6111, 6112, 6113. Available online: `http://www.ietf.org/rfc/rfc4120.txt`.

[112] J. Klensin, R. Catoe, and P. Krumviede. **IMAP/POP AUTHorize extension for simple challenge/response**. IETF, RFC 2195 (Proposed Standard), September 1997. Available online: `http://www.ietf.org/rfc/rfc2195.txt`.

[113] B. Hartmann, S. Doorley, and S. R. Klemmer. **Hacking, mashing, gluing: Understanding opportunistic design**. *IEEE Pervasive Computing*, **7**(3):46–54, July 2008.

[114] S. Govardhan and G. Feuerlicht. **Itinerary planner: A mashup case study**. In E. Di Nitto and M. Ripeanu, editors, *Proceedings of International Conference on Service Oriented Computing Workshops*, **4907** of *Lecture Notes in Computer Science*, pages 3–14. Springer, 2007.

[115] Y. Liu, X. Liang, and L. Zhu. **A component-based approach to developing thematic mashups**. In *Proceedings of the 20th Australian Software Engineering Conference*, ASWEC '09, Los Alamitos, CA, USA, April 2009. IEEE Computer Society.

[116] A. Taivalsaari and T. Mikkonen. **Mashups and modularity: Towards secure and reusable web applications**. In *Proceedings of 23rd IEEE/ACM International Conference on Automated Software Engineering Workshops*, ASE '08, pages 25–33, Los Alamitos, CA, USA, September 2008. IEEE Computer Society.

[117] J. López, A. Pan, F. Bellas, and P. Montoto. **Towards a reference architecture for enterprise mashups**. In *Actas de los Talleres de las Jornadas de Ingeniería del Software y Bases de Datos*, **2** of *SISTEDES '08*, pages 67–76, 2008.

[118] G. Bader, A. Anjomshoaa, and A. M. Tjoa. **Privacy aspects of mashup architecture**. In *Proceedings of the 2010 IEEE Second International Conference on Social Computing*, SOCIALCOM '10, pages 1141–1146, Washington, DC, USA, 2010. IEEE Computer Society.

[119] ISO/IEC/(IEEE). **IEEE recommended practice for architectural description of software-intensive systems**. *IEEE Std 1471-2000*, pages 1–23, August 2002.

[120] G. Krasner and S. Pope. **A description of the Model-View-Controller user interface paradigm in the Smalltalk-80 system**. *Journal of Object Oriented Programming*, **1**(3):26–49, 1988.

[121] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, and J. E. Robbins. **A component- and message-based architectural style for GUI software**. In *Proceedings of the 17th international conference on Software engineering*, ICSE '95, pages 295–304, New York, NY, USA, 1995. ACM.

[122] W. W. Eckerson. **Three-tier client/server architecture: Achieving scalability, performance, and efficiency in client–server applications**. *Open Information Systems*, **10**(1), January 1995.

[123] T. Reenskaug. **Models-views-controllers**. *Technical note, Xerox PARC*, December 1979.

[124] T. Reenskaug. **THING-MODEL-VIEW-EDITOR: An example from a planning system**. *Technical note, Xerox PARC*, May 1979.

[125] D. Crockford. *JavaScript: The Good Parts*. O'Reilly Media, Inc., Sebastopol, CA, USA, 2008.

[126] S. Stefanov. *JavaScript Patterns – Build Better Applications with Coding and Design Patterns.* O'Reilly Media, Inc., Sebastopol, CA, USA, 2010.

[127] C. Cappiello, F. Daniel, and M. Matera. **A qality model for mashup components**. In *Proceedings of the 9th International Conference on Web Engineering*, ICWE '9, pages 236–250, Berlin, Heidelberg, 2009. Springer-Verlag.

[128] C. Cappiello, F. Daniel, M. Matera, and C. Pautasso. **Information quality in mashups**. *IEEE Internet Computing*, **14**(4):14–22, July 2010.

[129] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip. **A framework for automated testing of JavaScript web applications**. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 571–580, New York, NY, USA, 2011. ACM.

[130] V. Dallmeier, M. Burger, T. Orth, and A. Zeller. **WebMate: a tool for testing Web 2.0 applications**. In *Proceedings of the Workshop on JavaScript Tools*, JSTools '12, pages 11–15, New York, NY, USA, 2012. ACM.

[131] R. M. Lerner. **At the forge: Testing JavaScript**. *Linux Journal*, (191), March 2010.

[132] K. Väänänen-Vainio-Mattila and M. Wäljas. **Towards user-centered mashups: Exploring user needs for composite web services**. In *CHI '11 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '11, pages 1327–1332, New York, NY, USA, 2011. ACM.

[133] P. van Schaik and J. Ling. **The role of context in perceptions of the aesthetics of web pages over time**. *International Journal of Human-Computer Studies*, **67**(1):79–89, January 2009.

[134] G. Lindgaard, G. Fernandes, C. Dudek, and J. Brown. **Attention web designers: You have 50 milliseconds to make a good first impression!** *Behaviour & Information Technology*, **25**(2):115–126, April 2006.

[135] E. R. Tufte. *The visual display of quantitative information.* Graphics Press, Cheshire, CT, USA, 1986.

[136] H. J. Wang, X. Fan, J. Howell, and C. Jackson. **Protection and communication abstractions for web browsers in MashupOS**. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 1–16, New York, NY, USA, 2007. ACM.

[137] D. Liu, N. Li, C. Pedrinaci, J. Kopecký, M. Maleshkova, and J. Domingue. **An approach to construct dynamic service mashups using lightweight semantics**. In *Proceedings of the 11th international conference on Current Trends in Web Engineering*, ICWE'11, pages 13–24, Berlin, Heidelberg, 2012. Springer-Verlag.

[138] L. Dusseault and J. Shell. **PATCH method for HTTP**. IETF, RFC 5789, March 2010. Available online: `https://tools.ietf.org/html/rfc5789`.

[139] H. Pennington, A. Carlsson, A. Larsson, S. Herzberg, S. McVittie, and D. Zeuthen. **D-Bus Specification version 0.19**. freedesktop.org, February 2012. Available online: `http://dbus.freedesktop.org/doc/dbus-specification.html`.

[140] D. Crane and P. McCarthy. *Comet and reverse Ajax: The next-generation Ajax 2.0*. Apress, Berkely, CA, USA, 2008.

[141] A. Russell, G. Wilkins, D. Davis, and M. Nesbitt. **Bayeux protocol – Bayeux 1.0.0**. The Bayeux Specification, 2007. Available online: `http://svn.cometd.org/trunk/bayeux/bayeux.html`.

[142] I. Paterson, D. Smith, P. Saint-Andre, and J. Moffitt. **XEP-0124: Bidirectional-streams over synchronous HTTP (BOSH)**. XMPP Standards Foundation, July 2010. Available online: `http://xmpp.org/extensions/xep-0124.html`.

[143] E.M. Maximilien. **Mobile mashups: Thoughts, directions, and challenges**. In *Proceedings of the 2008 IEEE International Conference on Semantic Computing*, ICSC '08, pages 597–600, Washington, DC, USA, 2008. IEEE Computer Society.

[144] S. Block and A. Popescu. **DeviceOrientation event specification**. W3C, December 2011. Available online: `http://www.w3.org/TR/orientation-event/`.

[145] A. Popescu. **Geolocation API specification**. Last call WD, May 2012. Available online: `http://www.w3.org/TR/geolocation-API/`.

[146] J. Warner and S. A. Chun. **A citizen privacy protection model for e-government mashup services**. In *Proceedings of the 2008 international conference on Digital government research*, dg.o '08, pages 188–196. Digital Government Society of North America, 2008.

[147] S. Van Acker, P. De Ryck, L. Desmet, F. Piessens, and W. Joosen. **WebJail: Least-privilege integration of third-party components in web mashups**. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 307–316, New York, NY, USA, 2011. ACM.

[148] F. Batard, K. Boudaoud, and M. Riveill. **A middleware for securing mobile mashups**. In *Proceedings of the 20th international conference companion on World wide web*, WWW '11, pages 9–10, New York, NY, USA, 2011. ACM.

[149] A. Bohannon. **Building secure web mashups**, 2008. Available online: `http://www.cis.upenn.edu/~bohannon/mashups.pdf`.

[150] OpenAjax Alliance. **Ajax and mashup security**. White paper, 2010. Available online: `http://www.openajax.org/whitepapers/AjaxandMashupSecurity.php`.

[151] S. Yoshihama, F. D. Keukelaere, M. Steiner, and N. Uramoto. **Overcome security threats for Ajax applications**. IBM developerWorks, 2007. Available online: `http://www.ibm.com/developerworks/library/x-ajaxsecurity/index.html`.

[152] J. Shanmugam and M. Ponnavaikko. **Cross-site Scripting – Latest developments and solutions: A survey**. *International journal of Open Problems in Computer Science and Mathematics*, **1**(2):8–28, September 2008.

[153] S. Fogie, J. Grossman, R. Hansen, A. Rager, and P. D. Petkov. *XSS Attacks: Cross-Site Scripting Exploits and Defense*. Syngress Publishing, Waltham, MA, USA, 2007.

[154] A. Barth, C. Jackson, and J. C. Mitchell. **Robust defenses for cross-site request forgery**. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 75–88, New York, NY, USA, 2008. ACM.

[155] R. Pelizzi and R. Sekar. **A server- and browser-transparent CSRF defense for Web 2.0 applications**. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 257–266, New York, NY, USA, 2011. ACM.

[156] B. Chess, Y. T. O'Neil, and J. West. **JavaScript hijacking**. Technical report, FORTIFY Software, March 2007. Available online: `http://www.net-security.org/dl/articles/JavaScript_Hijacking.pdf`.

[157] W. Zeller and E. W. Felten. **Cross-Site Request Forgeries: Exploitation and prevention**, October 2008. Available online: `https://www.eecs.berkeley.edu/~daw/teaching/cs261-f11/reading/csrf.pdf`.

[158] C. Jackson, A. Barth, A. Bortz, W. Shao, and D. Boneh. **Protecting browsers from DNS rebinding attacks**. *ACM Transactions on the Web*, **3**(1):2:1–2:26, January 2009.

[159] I. Hickson. **HTML5 Web Messaging**. W3C, December 2012. Available online: `http://dev.w3.org/html5/postmsg/`.

[160] M. D. McIlroy. **Mass-produced software components**. In *Proceedings of NATO Conference on Software Engineering*, pages 88–98, October 1968.

[161] J. Hendler. **Web 3.0 emerging**. *IEEE Computer*, **42**(1):111–113, January 2009.

[162] J. M. Silva, A. S. Md. Mahfujur Rahman, and A. El Saddik. **Web 3.0: A vision for bridging the gap between real and virtual**. In *Proceedings of the 1st ACM international workshop on Communicability design and evaluation in cultural and ecological multimedia system*, CommunicabilityMS '08, pages 9–14, New York, NY, USA, 2008. ACM.

[163] T. Berners-Lee, J. Hendler, and O. Lassila. **The Semantic Web**. *Scientific American*, **284**(5):34–43, May 2001.

[164] N. Shadbolt, T. Berners-Lee, and W. Hall. **The semantic web revisited**. *IEEE Intelligent Systems*, **21**(3):96–101, May 2006.

[165] R. MacManus. **Eric Schmidt defines Web 3.0**. ReadWrite Blog, August 2007. Available online: `http://www.readwriteweb.com/archives/eric_schmidt_defines_web_30.php`.

[166] I. Hickson. **Web workers**. W3C, May 2012. Available online: `http://www.w3.org/TR/workers/`.

[167] T. Mikkonen and A. Taivalsaari. **Apps vs. open web: The battle of the decade**. In *Proceedings of the 2nd Workshop on Software Engineering for Mobile Application Development*, MSE'2011, pages 22–26, October 2011.

[168] L. D. Paulson. **Developers shift to dynamic programming languages**. *IEEE Computer*, **40**(2):12–15, February 2007.

Publication I

Mashups – Software ecosystems for the web era

A. Salminen and T. Mikkonen

Mashups – Software Ecosystems for the Web Era

Arto Salminen, Tommi Mikkonen

Department of Software Systems
Tampere University of Technology
P.O.Box 553, FI-33101 Tampere, Finland
arto.salminen@tut.fi, tommi.mikkonen@tut.fi

Abstract. Web-based software and services are available all over the world instantly after they are released online. They can be used and updated without need to install anything, and once in place, they can also be reused in other contexts. As the amount of web services and devices used to consume data has exploded, it is becoming difficult to handle and gain access to the relevant data. Mashups are a new breed of web applications that act as content aggregates that leverage the power of the Web to support instant, worldwide sharing of content. Another dimension of mashups is that since they build on services that are readily available, they are also implicitly creating software ecosystems between service providers and application developers. In this paper, we address the role of mashups in the creation of software ecosystems for the web era. In addition, we identify four levels of support that service providers can offer for mashups. Furthermore, we will also discuss the different flavors of mashups as well as implementation considerations that are relevant from the ecosystem perspective.

Key words: Mashups, web applications, software ecosystems

1 Introduction

The web-based software is available all over the world instantly after the online release. It can be used and updated without need to install anything. Applications can support user collaboration, i.e., allow users to interact and share the same applications over the Web. In addition, numerous web services allowing users to upload, download, store and modify private and public resources have emerged. These resources can include personal images, texts, videos, e-mails, etc. as well as public data such as stock quotes, weather data and news feeds.

As the amount of web services and devices used to consume data has exploded, it is difficult to handle and gain access to the relevant data. To be able to handle the situation, searching has become one of the most important service of the Web. However, searching can be used only for data accessing, not for analyzing or parsing it. Similarly to resources, communication has decentralized into different services such as e-mail, different social media services, instant messaging services, chats, blogs, etc. Therefore, new mechanisms are needed for resource handling and communication services of the Web.

An important realization is that applications built on top of the Web do not have to live by the same constraints that have characterized the evolution of conventional desktop software. The ability to dynamically combine content from numerous web sites and local resources, and the ability to instantly publish services worldwide has opened up entirely new possibilities for software development. In general, such systems are referred to as mashups, which are content aggregates that leverage the power of the Web to support instant, worldwide sharing of content.

In this paper, we address the role of mashups in the creation of implicit and explicit software ecosystems [1] for the web era. We perform this in the following fashion. Section 2 provides an overview to mashups and the potential associated with mashup development. Section 3 discusses mashup ecosystems from the viewpoint of already existing research as well as challenges we have encountered in practice. In addition, this section identifies the four levels of support that service providers can offer for mashups, and discusses the associated consequences of each approach. Section 4 discusses the different breeds and types of mashup ecosystems that have been introduced, and Section 5 discusses implementation considerations. Finally, Section 6 draws some final conclusions.

2 Mashups: An Overview

Mashups can be characterized as applications that combine resources – data, code and other content – from different services in the Web into an integrated experience. Mashups can combine the content in new, unforeseen ways, thus creating entirely new web services, or they can provide new visualizations for already existing service. For instance, a mashup can combine a map with images that can be attached to specific locations. Another type of mashup can visualize the images in novel fashion, for example on a timeline or as a collage.

Mashups have potential for great user experiences, as they include more functions than just composition. Mashups can be used to filter, combine and modify data retrieved from multiple sources over the Web. Combining web resources into mashups is an efficient way to create new services or extract relevant information from a complex mixture of source data. Even unexpected innovations are possible as mashups can combine resources in unforeseen fashion. Furthermore, mashups are even more usable when non-technical users can create them with special purpose tools and have their own views for data. This is very inspiring part of mashups as it allows creative users to design their own applications that are capable of doing unexpected things. Allowing "do-it-yourself" mashups serve the long tail of users having diverse needs that are not fulfilled by existing applications or services. On embedded devices, mobile devices being at the forefront, mashups can benefit from accessing the user's context to combine resources, potentially automatically.

Well-build mashups have functionality for filtering source data. By having adjustable filters a mashup can provide more relevant results. Filters can be based on much more relevant variables than manually entered limits such as the

highest and the lowest price of a product. Such filters can be time of the day, location of the user, past activity of the user, activity of other users (trends), profile setting of users mobile device, etc. Heavy processing, e.g. filtering images with face detection algorithm, can be executed on the server, using MashReduce programming model [2], for instance.

Different kinds of dependability mechanisms play an important role in a mashup. At least the mashup should be implemented so that it checks whether the input data is correct. More sophisticated mashups can have fall-back mechanisms that, instead just giving up on error, try to use next best strategy to ensure even partial functionality. Furthermore, mashups can have controlling mechanisms that supervise the functionality and replace failing parts with other ones. In addition, mashups can have capabilities to extract the result mashup to some external viewing device and change the user interface of the mashup accordingly. For instance, this enables the creation of a mashup in a mobile device whereas the resulting output can be shown on a bigger screen if one is available.

3 Mashup Ecosystems

Since mashups by definition combine data from multiple sources, the stakeholders that provide this data form an ecosystem, i.e. a set of entities that act as a single unit instead of each participating business acting separately. This ecosystem – formed by service providers, mashup authors, and users as visualized in Fig. 1 – need not be controlled by a central authority. In contrast, even though mashup authors and service providers may have an explicit service level agreements (SLA), it is common that mashups are developed without such contracts, and the ecosystem is formed implicitly. For instance, one can build a mashup on top of services freely available in the web with liberal enough licenses. In a broad sense, any web document author can be considered as a service provider, as it is common that content is gathered from web sites by technique called "screen scraping" or "web scraping", where source data is parsed from HTML pages aimed at human readers.

In the following we will first provide some background information regarding mashup ecosystems, and then advance to some challenges associated with the establishment of new mashup ecosystems.

3.1 Background

Yu and Woodard [3] have described mashup ecosystems by using the ProgrammableWeb mashup indexing service (`http://www.programmableweb.com/`) data as source. They investigate the structure and dynamics of the Web 2.0 ecosystems by analyzing the data available about mashups and APIs. The first finding was that at the time of the study APIs were organized into three tiers, which were 1) the most popular API (Google Maps), 2) popular APIs (many APIs used for social services and searching) and 3) less popular APIs (APIs often

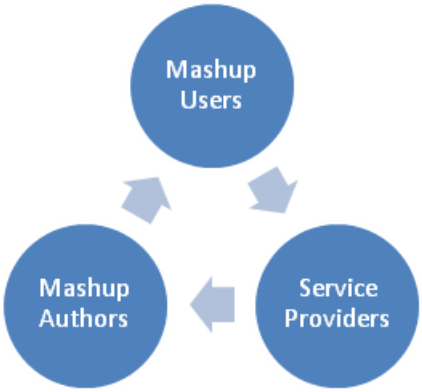


Fig. 1. Mashup Ecosystem

used for blogging, online retail, music, videos and feeds). The second finding was that mashups are often composed by combining APIs across tiers. This highlights the central role of the most popular APIs, but also reveals the importance of less popular APIs in dilution of the ecosystem. Many of the third tier APIs bring together novel combinations of functionality. Another interesting finding is that in contrast to what has been suggested [4], there is no long tail of services that would form a basis for a significant number of mashups. Instead, Yu and Woodart noticed that 95% of mashups are build on 20% of services, which is much more than in the famous Pareto Principle, or 80/20 rule as it is often called. Moreover, they noted that 51% of services were not used by mashups at all. However, one should bear in mind that Yu's and Woodard's data source, ProgrammableWeb, lists only those services and mashups that have been added to it by developers. Therefore there are services and mashups that are not included in the source data.

Bosch has reviewed mashup ecosystem from end-user programming point of view [5]. Bosch also pointed out two success factors as well as two challenges that this ecosystem has. The two success factors are, first, the value that end-users gain by designing their own applications, and second, sharing of applications among users. The two challenges are enabling the end-user programming for inexperienced developers and minimizing ecosystem maintenance efforts. Furthermore, Bosch identifies so called "undirected developers" that are able to use the platform in unforeseen ways and provide significant innovations for the overall ecosystem. Similarly to [5], our perception is that mashup ecosystems are very valuable for end-users and service providers. However, despite popularity that mashups have gained, polished end-user programming solutions for mashups have not been very successful. In contrast, some promising efforts by major players on the field, such as Google Mashup Editor and Microsoft Popfly, have been discontinued, and mashup composing still lacks tool support. On the other hand, mashup development has focused on building applications with traditional web development tools and architectures. Consequently, our previous

research has been focused on applying good software engineering practises on mashups, with the most practical tools at hand [6, 7].

Another interesting study is concerning the way a mashup ecosystem grows. For instance, hypothesis in [8] is that mashup developers create new mashups by copying existing ones. Simulations suggest that this would be true, as it is in line with the reports about mashup ecosystem growing [3]. However, the hypothesis of [8] has not been tested empirically.

3.2 Designing Services

Service providers are crucial stakeholders in mashup ecosystems, as they provide the necessary content that is reused in mashups. There are numerous motives to allow liberal access to the content of a service. One rationale is a desire for getting a wider audience for certain platform, product, or content accessed through the service. Moreover, opening a service can lead to numerous clients created by third party developers to emerge on different platforms and for different user requirements. Some services are designed so that spreading advertising messages along with the content is possible.

Service providers support mashup ecosystems in four identifiable levels, which are described in the following:

1. *No support for mashups.* Some web content authors do not support mashups at all and provide their content solely as regular web documents. This kind of content is still accessible with "screen scraping", but such accessing is typically error prone, and it often is illegitimate. Some services even have implemented technical measures to prevent scraping. Furthermore, even if reusing the content in mashups would be allowed, the web content author does not have control on what parts of the content is reused, and it is difficult to build a business model around such approach towards mashups. In addition, it is likely that accessing the content is very inefficient and cumbersome from mashup author's point of view. Furthermore, since even the smallest change in the web page can lead to a different interpretation of the content, mashups relying on such services are usually somewhat fragile.
2. *Access through a web feed.* It is common that regularly updated sites, such as blogs or news sites, provide their content through RSS, Atom, or other type of web feed. A web feed is easy to set up and maintain, particularly if some publishing system is used. The feed is intended mainly for users to subscribe with some feed reader application, but at the same time the data becomes accessible for mashups, too. While it is possible to establish some kind of licensing for reusing the content, the control over the content is still rather coarse. Use cases of web feeds are limited to accessing the content as a whole, as, for instance, querying certain content item is not possible. Utilizing web feeds in mashups is typically straightforward as helpful libraries and tools for such task are available on most platforms. Some dedicated mashup tools, Yahoo! Pipes (`http://pipes.yahoo.com/`) for instance, support only web

feeds if content from an arbitrary service is desired to be included into a mashup.

3. *Access through a web interface.* Providing a service with a web interface, typically following either REST or SOAP architecture style, enables using the service in mashups. Use cases of such interface allows not just data accessing but other types of services as well. For instance, a service can provide means for social communication, authentication, database accessing, or specialized functions such as reverse geocoding or music identifying. Setting up a web service with REST or SOAP interface requires careful planning and implementation, especially if sensitive information is handled. However, such system allows fine-grained control over the content as well as applications using the interface, and it enables different kinds of business models. Service load can be handled as well by limiting requests made in a time period, even individually for each application. Utilizing well-designed web interfaces in mashups is straightforward, and maintaining efforts that are needed when the service is updated are typically trivial. Conveniently, the content can be provided in different formats for the mashup developers to choose from, for instance both JSON and XML formats are often supported.
4. *Access through a programmatic interface.* Establishing a programmatic JavaScript API allows to integrate the sevice tightly with arbitrary web applications and mashup ecosystems. Such interface is used by including a JavaScript library into the application, which makes it possible to use the service with regular JavaScript function calls. Typically the JavaScript library is downloaded from the service provider's server instead of having a copy on the server hosting the mashup, which makes possible to always use the most recent version of the library. Setting up a programmatic JavaScript interface requires careful engineering, but it enables superior control over the content and applications. Diverse business models are possible, and the content can be provided with different terms and licenses for individual clients. Program code of the JavaScript library is often protected against misuse by code obfuscation or by other technical means. Considerable downside of the programmatic interfaces is that updating the interface affects directly on the mashup implementation. Therefore, programmatic interfaces are often provided in numerous versions, and a new version is introduced whenever features are added. Consequently, bug fixes need to be performed on all the versions, which makes maintaining the interface more laborious. Another downside is that if a programmatic interface is desired to be used on other runtime environments than a web browser, a parallel version needs to be provided. For instance, Google Maps API (`https://developers.google.com/maps/`) has separate native SDKs for Android and iOS mobile operating systems, and used to have another version for Adobe Flash Player (`http://www.adobe.com/products/flashplayer.html`). The Flash version was deprecated in September, 2011.

The proliferation of programmatic interfaces is a step towards software created from downloadable components, which is sometimes referred to as *mash-*

ware, web software development technique described in [9]. The most successful example of this kind of interface is Google Maps JavaScript API, which is also the most popular interface used in mashups [3]. It can be argued that one reason behind the success of this API has been the implementation style, which is particularly convenient for application developers, as it is similar to DOM (Document Object Model) and other interfaces that can be found from web browsers. However, Google Maps is not the only example of programmatic interface approach, as there are numerous other examples including user authentication, social networking, HTML5 music and video players, and data visualization, among others.

Until recently most of the services have been provided for free with the exception of some very specialized ones such as image content recognition services. However, in October 2011 Google announced that Google Maps API will be provided in two different versions: free and non-free, with the latter called Google Maps API for Business. The one with a prize tag provides more advantageous features such as higher request limitations and technical support. Even if this is the first remarkable example of this kind of development, it is an interesting change, particularly when bearing in mind that the Google Maps is the most popular service used in mashups, and it is widely utilized in other types of web applications, too. Therefore, this development may indicate a beginning of a new kind of emerging business model.

3.3 Designing Mashups

Typically, mashups are build with combination of server- and client-side parts. Functionality between these two parts is divided according to what is suitable for the current design. In the early days, dynamic web sites were created on server-side with combination of C programs, Perl, and shell scripts using Common Gateway Interface (CGI). Today, server-side web applications are often developed with Java, server-side JavaScript, Perl, PHP, Python or other suitable language. Applications of this kind work especially well if the client device has low processing resources as heavy processing takes place at the server-end and the client just shows the result. As client-end terminals have become more capable, it has become possible to compose mashups where the business logic resides completely on the client-end.

While mashups can be constructed in numerous different ways with a plethora of tools, there still are major practical problems related to mashup composing and security. For instance, the web browser security model is too restricting for mashups, tools introduced are lacking behind, and using dynamic languages for large applications is an unknown territory for many developers. The field of web programming is constantly changing as new interfaces, technologies and frameworks build upon novel technologies emerge constantly. The amount of different, constantly evolving APIs with different licenses is overwhelming. When developing large-scale mashups, situation may be even more problematic. Mashup authors build their applications on web services, and mashup users can add content to these services and consume it with mashups. Such ecosystem has commercial potential, which is, however, limited because of technical, legal and

other reasons. Some of the issues are general for both mobile and desktop environment, but naturally mobile mashups have their own specific things to handle as well.

While mobility restricts applications and application development, at the same time it is a great enabler from the mashup development point of view. The dynamic nature of mashups suits well for different ways mobile terminals can be used. Often, the information needed on the fly is related to user's context, which can be available for applications to access automatically [10]. This opens up opportunities to provide advantageous user experiences, as mashups can dynamically present eligible information, possibly even automatically without requiring specific user action. However, as mobile devices capabilities are limited, extending mashups to the mobile domain is not trivial, and special solutions are sometimes necessary. Mobile mashup ecosystem challenges, especially from utilizing multimedia in mashups point of view, have been described in detail in our previous paper [11].

There are situations when the composition of a mashup is not possible using only dynamic code. For example, applications that require a lot of computation power or access to interfaces that are not available for dynamic code, have to be constructed with both dynamic and native code. Therefore, offering an interface for mixing web technologies with the capabilities of native software components is sometimes necessary. On the other hand, utilizing hybrid technology allows one to combine the best of both worlds: performance and eye candy of traditional, installed binary applications and pervasiveness and seemingly infinite resources of the web.

3.4 Legal Considerations

In general, web interface legal terms and conditions are diverse. Commonly service providers set restrictions for those uploading content to the service, as well as those utilizing content of the sevice through an API, including mashup developers. In the following, some typical requirements and terms that affect mashup development are described.

- Service Level Agreements (SLA) are used to provide uptime guarantee or to state that the API has no liability for downtime or unexpected changes. Sometimes the latter is available for those who use the interface for free, and the former for paying customers.
- If the interface allows accessing user created content under different licenses, terms of service (TOS) require developers to strictly follow those licenses. If the application uses a cache, also the cache needs to reflect changes in content's licenses and availability. Sometimes service terms determine time limits for the cache reflecting these changes. Moreover, caching may be forbidden completely.
- If the interface enables accessing user's private data, TOS usually include restrictions about how this data can be used and stored. The service provider's logo or other branding needs to be explicitly available in the mashup. Other

services require adding acknowledgements to application source code. Detailed terms on how the branding is presented may be represented. For instance, when using Google Maps, the terms of the Google Maps API require that the Google logo is the largest logo in the final implementation (`https://developers.google.com/maps/terms`).
- Interface access rate can be limited to a certain amount of requests in a time period. For instance, Twitter limits unauthenticated calls to 150 requests per hour, whereas authorized calls are limited to 350 requests per hour (`https://dev.twitter.com/docs/rate-limiting`).
- Certain types of applications may be prohibited by the service provider. For instance, Flickr TOS deny using Flickr API for any application that replicates or attempts to replace the essential user experience of Flickr.com (`http://www.flickr.com/services/api/tos/`).
- Repeated violations of interface terms, for instance exceeding use rates or using the API in a forbidden type of an application, may make the service provider to terminate certain application from accessing the interface. Technically this can be achieved by restricting application IP addresses or application specific API key from accessing the service. In practice, TOS often contain a clause for such situation, although we have no data how commonly the clause is exercised.

The above issues are further complicated by the fact that in many cases, mashup developers have not signed a formal contract with service providers, but rely on licenses. Consequently, as copyright owners and service providers can change licenses more liberally than signed contracts, developers may end up accidentally violating license rights overnight when the original service provider updates license terms.

4 Sample Mashup Breeds and Ecosystems

Mashups can be classified based on numerous criteria, which in many ways affects the fashion the associated ecosystem can be established. One can classify mashups into breeds, such as server- and client-side mashups, and multiple and single API mashups. Moreover, mashup ecosystems can be classified into explicit and implicit ecosystems. The former includes commercial and enterprise mashups, and the latter includes situational mashups, as well as most mashups, that can be classified according to the most essential API used, establishing an ecosystem that is led by the provider of this API. In the following, mashup breeds and ecosystems are described in more detail and examples of different types of mashup ecosystems are presented.

4.1 Mashup Breeds

Server- and client-side mashups. One way to classify mashups is division between server-side and client-side mashups, based on where downloading, pro-

cessing and generating of the web content takes place. Server-side mashups application logic as well as accessing different web resources is implemented at the server-end. Client-side mashups are implemented completely on the client-end so that processing takes place at the user's web browser. Because of historical reasons, server-side approach has been more popular in the past, but as the processing power of web browser at the client-end has increased, client-side approach has become common as well. These two types of mashups have their advantages as well as disadvantages and suit for different situations, for instance a server-side mashup is not limited by browser's security model, the same origin policy, that isolates documents loaded from distinct origins from each other. Naturally hybrid approach combining server- and client-side mashup techniques is possible as well, and mashup developer can decide how to divide the functionality between the server and the client. If a mashup ecosystem consists of client-side mashups, it is necessary to pay more attention on how mashups can interact with services located at different origins. In addition, accessing specific mashup clients may be difficult because of addressing issues in IPv4-based networks.

Multiple and single API mashups. Instead of combining content from multiple APIs, which is usually the case, some mashups are using only one single API to create new visualization for existing web services. Often the user interface of this kind of mashups is simplified and added with attractive properties of some kind. Another kind of single API mashups provide more advanced ways for searching than the original service. For instance, there are numerous mashups that show images retrieved from the popular image service Flickr. Another example of a single API mashup is WikiMindMap (`http://www.wikimindmap.org/`), which generates a mindmap about a keyword based on Wikipedia articles. Mashup ecosystems that consist of numerous single API type of mashups are usually build around the few most popular services of the web. Such ecosystems have emerged, for instance, around Google Maps, Flickr, Wikipedia and Twitter.

4.2 Explicit Mashup Ecosystems

Commercial mashups. Commercial mashups are created to show a profit for the mashup publisher where as non-commercial mashups are provided non-profit. In commercial mashup ecosystems, mashup authors and service providers coordinate explicitly and use either specific contracts or common TOS agreements. Commercial mashup ecosystem is required to implement reliable and secure methods to in order to transfer sensitive data. In addition, availability of services in the ecosystem in a commercial setting is naturally vital. A typical example of a commercial mashup combines information about the product being sold with user reviews from multiple sources. Another type of commercial mashups is those including advertisements. Commercial mashups are targeted at consumers in contrast to enterprise mashups that are targeted at business users, even though both are often created by a company. It is common that a commercial mashup is provided for mobile device users as an alternative user interface for an electronic commerce. Further examples of commercial mashups

are price comparison and product search mashups. For instance, there are numerous mashups offering this kind of service based on Amazon's and EBay's price data. Another kinds of commercial mashups help to locate a certain dealer on a map. An example of a commercial mashup combining social network services is Scupal (`http://www.scupal.com/`), a social buying website launched in India. Scupal allows users to select a product they would be willing to purchase, and then gather other interested buyers of the same product within their social networking contacts. The more there are buyers the less is the price.

Enterprise mashups. Enterprise mashups are developed to solve some particular business-related problem. In contrast to consumer mashups, that utilize only open web services, they can use closed enterprise data sources and combine the information with data from the web. Forming more closed ecosystem than the commercial mashup ecosystem, enterprise ecosystem is controlled by organization's internal interface specifications and descriptions, in addition to usage of public web services available under common TOSs. Security features of an enterprise mashup ecosystem are crucial as sensitive data of an organization is often handled. For instance, storing the data should be done in controlled fashion within the organization's own storage facilities. Enterprise mashups can be created solely by the company's IT department or a sand-box environment may be provided for non-experts to create mashups. However, the more degree of freedom is allowed, the greater are the skills needed for mashup development. Typical to enterprise mashups is that they focus is on a single presentation and target at providing a tool to help collaboration with different people working with the same objective.

Reusing of existing mashup solutions is often in a key role in an enterprise mashup ecosystem. One activity that targets at such reuse is Enterprise Mashup Markup Language (EMML), which is a XML-based domain specific language for developing enterprise mashups developed by the Open Mashup Alliance (OMA). With EMML, OMA aims at introducing a standardized, consistent and interoperable way to develop enterprise mashups. In addition to defining the language, OMA provides a reference implementation of a runtime that processes mashup scripts written in EMML. EMML can be used to declaratively describe the data processing flow, i.e. data composing, of a mashup.

4.3 Implicit Mashup Ecosystems

Situational mashups. Term *situational application* is used about an application that is created for a narrow group of users with unique needs, and some mashups are developed as situational applications. In Clay Shirky's essay *Situated Software* [12] this type of applications are described to be "designed for use by a specific social group, rather than for a generic set of 'users' ", and therefore, ecosystems build around situational mashups are implicit. Typically situational applications have short life span and the quality of engineering may not be first class. In addition, scaling up is often difficult with situational applications. However, Shirky remarked that as the group of users is relatively small, it is often unnecessary to implement mechanisms for user supervision.

Furthermore, situational applications are typically more personalized, and they can contain pre-entered information that is relevant only for the small group of intended users. As simple mashups that utilize readily available interfaces can be composed together rather quickly, the cost of implementation is relatively low, and the ecosystem containing situational mashups may have rather lightweight security, moderation and authentication features. Therefore, mashups can be targeted at small, specific groups of users and be very personalized, as well. The architecture and other engineering aspects of this kind of mashups may not be the most polished, but with the specific target group and purpose, it does not have resonance. One should bear in mind, however, that when mashups are used to address non-trivial, more complicated issues, this approach should not be used as it quickly leads to difficulties. Situational mashup ecosystems can emerge swiftly, but typically lifespans of such ecosystems are shorter as well.

The most essential API. One way to do the classification is to use the type of most essential API to determine the mashup type. For instance, a mashup can be classified as social, news, map, image, video, audio or search mashup based on the main service utilized. In consequence the ecosystem is build around this central service, and it can contain both implicit and explicit interactions. Often these mashups are targeted at consumers and provided for free, and therefore the implicit model is more common. Mashup statistics divided into categories based on the essential API used in a mashup can be collected from ProgrammableWeb site. The site provides statistics about mashups as well as service interfaces used to create new mashups. Only those mashups that are submitted to the website are listed, but the site can be used as a source for suggestive information about consumer mashups. However, the site does not list enterprise mashups at all. As can be seen in Fig. 2, mapping mashups are the most popular type of mashups. Social, search, photo, shopping and video mashups are roughly equally popular. In addition, remarkable number of mashups have been discontinued (tagged "deadpool").

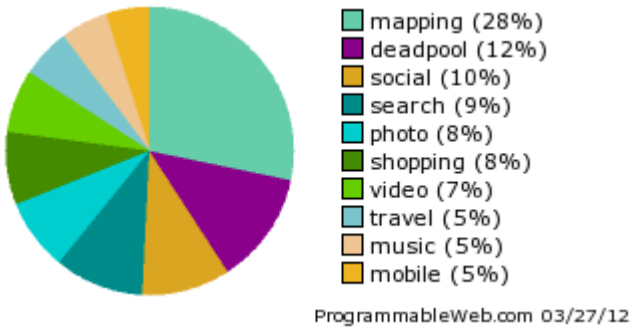


Fig. 2. Mashup types according to ProgrammableWeb (`http://www.programmableweb.com/mashups`).

5 Implementation Considerations

In our research (see e.g. [6, 7]), we have identified challenges that mashup ecosystem confronts in the areas of cloud infrastructure, web services, legal issues, and tool support. In the following, these will be briefly addressed.

Cloud Infrastructure. Cloud infrastructure related issues refer to addressability of mashup ecosystem endpoints as well as transparency and protocol support of network. Before IPv6 gains ground it might be necessary to use higher level methods to address network endpoints. Addressing mashup clients and services at too high level can derive scalability and performance issues. Another problem is caused by non-transparent network nodes that may cause some parts of the ecosystem become unattainable. Furthermore, lack of protocol support for other than HTTP can cause for instance video and audio streams fail to operate.

Web services. Web service reliability and complexity of integrating a high number of services are another type of challenges. Web service reliability can be addressed by adding fallback mechanisms, but this strategy will make the implementation more complex. While adding more services to the mashup can be attractive for users it makes the implementation more complex and increases vulnerability to service breakouts and incompatible version upgrades, which plague especially mashups that reuse services anonymously without explicit contracts. Furthermore, client-end device capabilities may be limited and operational expenses can be an issue, especially with mobile devices.

Legal issues. Moreover, legal issues related to mashup ecosystem are numerous. Service terms are often incompatible and hard to follow in complex mashups. The situation is even more complex when a mashup is hosted on third party platform, such as mashup tool providers servers. Mashups can be required to follow some content related limitations as well. For instance, some content can be freely available in U.S. but restricted from accessing in U.K. Some service providers may restrict their interfaces to be used only on desktop and prohibit using them on mobile devices. In addition, libraries and frameworks used in mashups may have conflicting licenses. Furthermore, protecting a client-side mashup from copying is often difficult as the executable code needs to be transferred to the client-end terminal.

Tools. Mashups can be developed with conventional web programming techniques using text editor and environment with debugging capabilities [13]. This requires considerable experience, as sometimes it is necessary to crawl the content from web pages. This is error-prone and can lead to hard-to-trace errors when subtle changes happen in the web sites from which content is downloaded. In contrast, dedicated mashup development tools can be helpful, especially when end-users are creating mashups. Typically, the target environment for dedicated tools is a web browser. Moreover, also the number of web sites from which content can be accessed is limited, and only few services are supported by the tools.

6 Conclusions

Web-based software and services have become commonplace. As virtually all imaginable content and services are becoming available online, there will be new, optimized ways to consume content and access services. In this paper, we have argued that mashups – special kinds of web applications that combine data and services from numerous sites – enables the development of new, improved applications that enrich the basic online facilities. A dimension that has been commonly overlooked with mashups is that they are not only about the technology, but their development and use is governed by other factors as well. Consequently the elements of opportunistic design – hacking, mashing and gluing, as pointed out in [14] – must be associated with the creation of sustainable software ecosystems of the web era.

Service interfaces are integral part of mashup ecosystems, and we indentified four levels of support that service providers can offer for mashups. We believe that the success of programmatic JavaScript interfaces is one indicator of the trend towards mashware ecosystems – software ecosystems that leverage source code and software components that are downloaded dynamically from all over the world. As pointed out in [9, 15], mashware ecosystems can dramatically improve productivity of web application development and allow global reuse of software components. However, research is needed in numerous areas including security, modularity and legal aspects, as well as software engineering methodologies to support the development of such ecosystems.

In the future, we expect that the multifaceted nature of mashups will lead to increasing interest also on the research side. So far, such applications, as well as associated ecosystems, have gained relatively little attention from researchers. Consequently, there are numerous directions for future work, where the main development principles of mashups in general as well as associated business impacts are analyzed in more detail.

References

1. Messerschmitt, D.G., Szyperski, C.: Software Ecosystem: Understanding an Indispensable Technology and Industry. MIT Press, Cambridge, MA, USA (2003)
2. Salo, J., Aaltonen, T., Mikkonen, T.: Mashreduce: Server-side mashups for mobile devices. In: Proceedings of the 6th international conference on Advances in grid and pervasive computing. GPC'11, Berlin, Heidelberg, Springer-Verlag (2011) 168–177
3. Yu, S., Woodard, C.J.: Service-oriented computing — icsoc 2008 workshops. Springer-Verlag, Berlin, Heidelberg (2009) 136–147
4. Hoyer, V., Stanoesvka-Slabeva, K., Janner, T., Schroth, C.: Enterprise mashups: Design principles towards the long tail of user needs. In: Services Computing, 2008. SCC '08. IEEE International Conference on. Volume 2. (july 2008) 601 –602
5. Bosch, J.: From software product lines to software ecosystems. In: Proceedings of the 13th International Software Product Line Conference. SPLC '09, Pittsburgh, PA, USA, Carnegie Mellon University (2009) 111–119

6. Mikkonen, T., Salminen, A.: Towards a reference architecture for mashups. In: Proceedings of the 2011th Confederated international conference on On the move to meaningful internet systems. OTM'11, Berlin, Heidelberg, Springer-Verlag (2011) 647–656
7. Salminen, A., Mikkonen, T., Nyrhinen, F., Taivalsaari, A.: Developing client-side mashups: experiences, guidelines and the road ahead. In: Proceedings of the 14th International Academic MindTrek Conference: Envisioning Future Media Environments. MindTrek '10, New York, NY, USA, ACM (2010) 161–168
8. Weiss, M., Sari, S.: Evolution of the mashup ecosystem by copying. In: Proceedings of the 3rd and 4th International Workshop on Web APIs and Services Mashups. Mashups '09/'10, New York, NY, USA, ACM (2010) 11:1–11:7
9. Taivalsaari, A.: Mashware: the future of web applications. Technical report, Mountain View, CA, USA (2009)
10. Mikkonen, T., Salminen, A.: Towards pervasive mashups in embedded devices. In: Proceedings of the 2010 IEEE 16th International Conference on Embedded and Real-Time Computing Systems and Applications. RTCSA '10, Washington, DC, USA, IEEE Computer Society (2010) 35–42
11. Salminen, A., Kallio, J., Mikkonen, T.: Towards Mobile Multimedia Mashup Ecosystem. In: IEEE International Conference on Communications Workshops, ICC Workshops. (2011)
12. Shirky, C.: Situated software. First published March 30, 2004 on the "Networks, Economics, and Culture" mailing list (2004)
13. Yu, J., Benatallah, B., Casati, F., Daniel, F.: Understanding mashup development. Internet Computing, IEEE **12**(5) (sept.-oct. 2008) 44 –52
14. Hartmann, B., Doorley, S., Klemmer, S.R.: Hacking, mashing, gluing: Understanding opportunistic design. IEEE Pervasive Computing **7**(3) (July 2008) 46–54
15. Mikkonen, T., Taivalsaari, A.: The mashware challenge: bridging the gap between web development and software engineering. In: Proceedings of the FSE/SDP workshop on Future of software engineering research. FoSER '10, New York, NY, USA, ACM (2010) 245–250

Publication II

Towards mobile multimedia mashup ecosystem

A. Salminen, J. Kallio and T. Mikkonen

Towards Mobile Multimedia Mashup Ecosystem

Arto Salminen
Department of Software Systems
Tampere University of Technology
P.O. Box 553, FIN-33101 Tampere, Finland
e-mail: arto.salminen@tut.fi

Jarno Kallio
PacketVideo Finland
Hallituskatu 8, FIN-33200 Tampere, Finland
e-mail: kallio@pv.com

Tommi Mikkonen
Department of Software Systems
Tampere University of Technology
P.O. Box 553, FIN-33101 Tampere, Finland
e-mail: tommi.mikkonen@tut.fi

Abstract — **Mashups that combine already existing data into an integrated experience are becoming increasingly popular. So far most mashups have been built around maps and images. However, as the web is becoming an increasingly ubiquitous media, also multimedia content – sound, video and even small programs – is emerging as a candidate for mashup creation, with potentially superior user experience. Currently available methods to implement mashups do not allow effortless access to personal data across domains or provide means to ensure that the user experience is coherent. This paper describes our mobile mashup ecosystem aimed to solve these problems and work as a ubiquitous platform for mobile multimedia mashups. Furthermore, we provide a brief literature review about existing mashup frameworks and end-user programming. In addition, to constitute a basis for future work, we discuss about challenges expected to emerge when the ecosystem is implemented.**

Mashup ecosystem, multimedia,mobile environment

I. INTRODUCTION

The Web has dramatically changed during the last 15 years. When in the old days we used the Web mainly for browsing static websites, sending e-mails and maybe for making occasionally VoIP calls, today it has become a deployment environment for personalized software systems such as word processors, spreadsheets, social network communication tools and even 3D applications. All this has had major effect on what we expect from our web experience. Mashup systems are one of the new breeds of applications that are used for personalized content and communication, made possible by the data that is readily available in the Web.

Mashup is a web application that combines code, data and other content from multiple sources creating some added value for the user. Traditional mashups have been executed inside a web browser but other systems can be used as runtime environment as well. In general, mashups are about simplicity, usability, and ease of access [1]. A mashup can combine the content in new, unforeseen way, thus creating entirely new web service, or a mashup can provide new visualization for existing service.

Currently there are numerous issues related to developing and using mobile mashups. Previous research [2, 3], based on our experiences about client-side mobile mashups build with Qt application framework (*http://qt.nokia.com*), described problems related to lack of well-defined, resilient interfaces. Many web services lack a well-defined API and even if the API exists, it has no public interface specification that would state which parts are meant to be used by third parties and which are only intended for internal use. In addition, our experience showed that mobile mashups also struggle with problems associated with usability, connectivity and performance.

While mobility sets restrictions regarding applications and application development, at the same time mobility provides a great opportunity from mashup development point of view. The dynamic nature of mashups suits well for different ways mobile terminals can be used. Often, the information needed on the fly is related to user's context, which can be available for applications to access automatically [4]. This opens up opportunities to provide advantageous user experiences, as mashups can dynamically present eligible information, possibly even automatically without requiring specific user action.

Since mashups combine data from numerous origins, the stakeholders that provide the information form an ecosystem – a set of entities that act as a single unit, instead of each participating business acting separately. Sometimes, explicit service level agreements (SLA) between businesses are made, but at times, an ecosystem can also be formed implicitly. For instance, one actor can build a service on top of services that are available from the web with liberal enough licenses that allow service composition.

In this paper, we discuss the creation of ecosystems for mobile multimedia mashup composing. The contribution of this paper is two-fold. First, we provide a literature review about currently available mashup ecosystems, mashup security and end-user programming solutions. Second, we describe our mashup ecosystem architecture requirements and we present number of fundamental challenges we expect to emerge, which constitutes a basis for future work that will be reported separately.

The paper is structured as follows. In Section 2, we provide a short literature review on already existing systems. In Section 3, we introduce a sample mashup ecosystem, based on which we reflect emerging challenges in Section 4. In Section 5, we draw some final conclusions.

II. BACKGROUND

Mashup ecosystems require technical enablers, such as open interfaces that provide support for combining content. There are numerous themes and already existing systems that can be associated with mashup composition in the context of

978-1-61284-955-3/11/$26.00 ©2011 IEEE

mobile multimedia. The themes that we address in the following include state-of-the-art mobile mashups, mashup frameworks, mashup security, and end-user programming.

A simple example of a communication mashup is Nokia N900 device's instant messaging application. It can be used to make connections to several communication services such as Facebook chat, SIP account, Microsoft Messenger, Google Talk or Skype, for instance. The application has support for add-ons so it can be easily expanded to handle new services as they emerge.

Google has created a mashup system called Google Maps for Mobile (*http://www.google.com/mobile/maps/*), whose main component is a map that shows the user's current GPS location. This mashup system can be used to track positions of friends and to display additional map related information, including for example traffic data, driving directions, interesting places or web camera images, on layers over the map.

Another interesting example of mobile map-based mashup is Telar Mashup for Nokia N810 tablet by Brodt et al. [5]. It uses modified mobile web browser to gain an access to GPS peripheral of the device and combines the location data with map retrieved from Google Maps.

De et al. have developed a mashup framework called Service Context Manager (SCM) framework [6]. SCM handles all the stages of context gathering, processing, inferring and reasoning to come up with useful recommendations. It consists of three parts: 1) device and service discovery function, 2) transformation framework and 3) reasoning module. The device and service discovery function searches the ambient environment for resources. Transformation framework aggregates the distributed context information and abstracts it to common, formal structure. The reasoning module is a hybrid inference and control system that asserts missing context information and provides application interface for reasoned and filtered context.

Ikeda et al. [7] have designed a framework for creating flexible mashups in which the user can selectively browse trough mashup items. The framework includes data management engine for on-demand data generation and GUI widgets that can be used to browse the data. These are both implemented on client-side as well as connections to different web services. On the server-side the framework provides only configuration files for widgets and data management.

Service access control API that aims to better mashup security has been studied by Hashimoto et al. [8]. The SAXAE API provides functions to the mashup to retrieve protected, non-public resources securely This allows the mashup to access user's private data, for example on some social service, in secure fashion.

Warner et al. have researched privacy protection model for government mashups [9]. This model allows users to specify their individual privacy policies that can be applied to the use of their data. The approach involves the protection of sensitive data based on not only who is requesting access but on the intended use too.

Another security related studies are lattice-based mashup security model by Magazinius et al. [10]. The security lattice is build from the origins of mashup components so that the each level of the lattice corresponds to a set of origins. To allow a controlled release of information between mashup components, a declassification mechanism is proposed. Declassification policy defines what pieces of information can be shared between components, so that sharing between components on the same level of the lattice is unrestricted, but limited between other levels. Sharing data from security level to another can only be done if it is allowed explicitly.

An identity management protocol for mashups has been studied by Zarandioon et al. [11]. Their proposal utilizes conventional public-key cryptography to eliminate need for a trusted identify provider. In addition, their research describes a framework that allows secure indirect communication between client-side mashup components. Both the identity management protocol and the framework are implemented as an in-browser library.

Tobias Nestler has done research [12] about Service Oriented Architectures from the Service-to-User point of view. He pointed out that existing mashup tools such as Yahoo Pipes (*http://pipes.yahoo.com/pipes/*), Microsoft Popfly (discontinued during autumn 2009) and IBM QedWiki (*http://www-01.ibm.com/software/info/mashup-center/*) are lacking support for input and output parameters other than simple data types. Another discovery was that UI components such as buttons, navigation elements or multiple screens, were not supported in existing tools.

Technical report by Antero Taivalsaari [13] lists a number of existing mashup development tools which are geared towards ordinary users instead of just professional programmers. Most of the tools described support programming via some kind of visual programming interface but allow source code editing as an advanced feature.

Research [14] by Wang et al. describes end-user mashup programming using nested tables and designing a developing environment that uses this approach. The programming model of the environment relies on spreadsheet-style view for the mashup data. The data inside worksheets can be modified with simple scripts that consist of very limited set of operations.

One interesting project by Resnick et al. [15] is Scratch, which is a programming language developed to allow children to make programs by using a simple graphical interface. Even though Scratch has been intended for children, people of all ages have used it. Over 540,000 registered members of Scratch online community (*http://scratch.mit.edu/*) have created over 1,100,000 projects (as of 13th June 2010).

Cao et al. studied end-user mashup designing [16] by conducting a think-aloud experiment with ten participants creating a web mashup using the Microsoft Popfly mashup tool already mentioned above. They discovered that participants designed their mashups by numerous iterations divided into framing, acting and reflecting phases. Another discovery was that presenting application logic alongside with the runtime output and providing visual hints about connection between those two would have made the debugging task a lot easier.

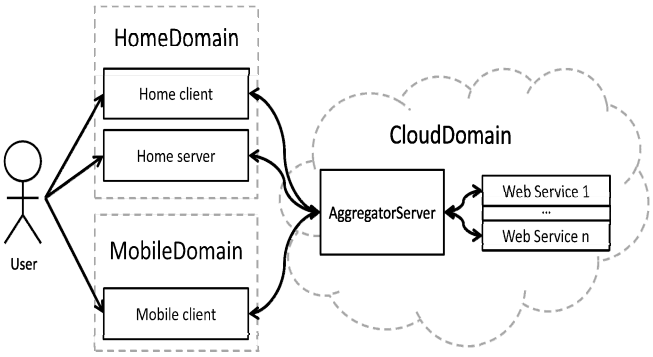


Figure 1. Sample mobile multimedia mashup ecosystem.

III. SAMPLE MASHUP ECOSYSTEM

In order to address challenges in a practical fashion, we next define a sample mashup ecosystem in terms of which we can discuss about general requirements regarding mashup development. The example ecosystem architecture, illustrated in Figure 1, consists of three domains: *HomeDomain*, *MobileDomain* and *CloudDomain*. HomeDomain can contain wide number of networked devices, which are usually personal or shared and have constant connection to the web. MobileDomain is typically the user's smart phone with high-performance internet connectivity. Finally, CloudDomain consists of AggregatorServer, which is connected to a number of arbitrary web services. AggregatorServer acts as a gateway that combines content from pool of web services and bridges all exemplary domains together enabling diverse mashup cases across all domains. The system does not pose restrictions on identity of the user, but in a typical scenario the user is the same person that can select the best possible combination of services, depending on the context.

Next, we discuss a number of relevant issues that form the important technical prerequisite for building such a system.

Specified interfaces. It should be possible to standardize the ecosystem to allow different vendors to implement compatible subsystems. The ecosystem should provide guidelines how to design interfaces so that the user experience would still remain coherent in different platforms. From user experience perspective having a minimum set of constraints for the non-functional properties of the ecosystem is important. The literature review shows that a number of frameworks for mashups have been proposed, and they can be evaluated further and applied to allow compatibility.

Portability. It should be possible to access the ecosystem with the most dominant mobile clients and still get roughly the same functionality. As an industrial example, PacketVideo provides mobile multimedia players for multiple platforms such as iPhone, Symbian, BREW, Linux and Windows Mobile (*http://www.pv.com/products/core/*).

Performance and Response Times. Performance and response times build should be comparable to multimedia applications using only local data. Research [17] done by Card et al. suggests the following guidelines as commonly applicable limits:

- 100 milliseconds is the limit for the user to have feeling that the interface is responding immediately,
- 1 second is the limit for the user to keep focus on the task at hand, without special feedback needed, and
- 10 seconds is the limit for keeping the user's attention focused at the program. For longer delays, some kind of process indicator is needed.

Security. Security and privacy issues should be considered carefully. Security model should be liberating enough to allow access to different services and user context data but eliminate possibility to misuse the system. The model should be easy to understand and implement without adding unnecessary complexity to mashup development. The literature review shows that work for mashup security has already been done, but applying it will still need careful evaluation, especially when targeting at mobile devices. In addition, we believe that extensive work performed earlier on other contexts – for example Mobile Java where a number of interfaces have been specified – can be applied.

IV. FUNDAMENTAL CHALLENGES

The main problem areas that constitute a basis for future work are listed in the following. We have divided the challenges under four categories: cloud infrastructure, web services, mobile devices and legal issues. Although often interrelated in real ecosystems, items in different categories are discussed separately.

A. Cloud infrastructure

Addressing. The universal lack of static IPv4 addresses forces the system to implement some of the low level connectivity functionality between mobile terminals and web servers with higher level protocols, such as HTTP and XML. This leads inevitably to performance issues, because network capacity is compromised when too high-level protocols must be used.

Routers. Protocol support for other than HTTP is often an issue. It is hard to predict what protocols are available in an arbitrary network, as there are numerous alternatives and resulting combinations. Problems may occur especially when applying streaming protocols in mobile context.

Firewalls. Accessing other parts of the mashup ecosystem might not be available because of unexpected protocol manipulation or communication blocks by firewalls, NATs or other non-transparent network nodes.

Bandwidth. Mobile broadband is inadequate for many kinds of multimedia services. Real time streaming of high-quality video may consume bandwidth to such a degree that simultaneous transfer of other multimedia content is not possible [18]. One option to address this problem is to deploy adaptive video transcoding [19, 20].

Servers. Scalability of the ecosystem for large amount of multimedia services and devices is a topic where further research is needed. There may be need to optimize servers for mobile connectivity and devices.

B. Web services

Aggregation of the Web. Implementing the Aggregator-Server efficiently will be a major challenge. In addition, because of web service APIs may change frequently, maintaining the AggregatorServer is most likely a laborious task, especially if a wide number of web services is supported.

Availability. Web service breakdowns and high service load should be taken into account. We need to build fallback mechanisms and alternative ways for providing the mashup experience.

Complexity. Because of integration complexity the ecosystem may become highly vulnerable for unforeseen problems. Therefore, the AggregatorServer interfaces should be carefully designed for simplicity. Applying research done on the field of REST interfaces may be helpful starting point for the design of the AggregatorServer [21].

C. Mobile Devices

Battery life. Rich mashups have negative effect for battery life. It is important to turn off unnecessary functionalities, such as animations, when they are not needed. One option to take this into account could be implementing power saving functionality based on the context of the user.

User operational expenses. Mobile networking costs for the user without data plan are relatively high. In addition, some major mobile operators have recently given up providing unlimited data plans. Mobile operators' business model does not necessarily suit for heavy use of mobile data.

Coherent user experience. Making mashups to work seamlessly together is difficult because of there is no standard way to implement web service interfaces. One important function of the AggregatorServer is to solve this problem.

D. Legal Issues

Licensing. Most mashups combine content that is created by different 3rd parties. Therefore, some legal issues may emerge. Different vendors may provide their content to be used freely, but others may strictly forbid using their data in other context. APIs may provide content marked with different licenses. For example, Flickr image hosting service API (*http://www.flickr.com/services/api/*) can be used to access both images marked with a Creative Commons license and with "all rights reserved" notices.

Terms of Service. Service provider's "Terms of Service" may set further regulations to the use of the content. For example, if a mashup application caches content in some way, it should take care of situations where the content author changes the content rights or visibility. Furthermore, it may be required that some specific text is displayed for the users of a mashup. For example, mashups that use content from Flickr need to prominently display a notice "This product uses the Flickr API but is not endorsed or certified by Flickr.". Furthermore, web applications using Flickr content are obligated to reflect removal of the content within 24 hours (*http://www.flickr.com/services/api/tos/*).

Commercial vs. other use. We are accustomed using most web sites for free. However, using their APIs for commercial purposes can lead to even stricter restrictions than described above. Often commercial use requires a separate agreement with the service provider. The service provider may charge for using the API although the same API is available for free for non-commercial use. For the creation of a fostering ecosystem, even if many systems are initially built on top of freely available facilities, this forms an essential item since building and hosting such a service obviously requires financial investments, at least in the long run.

E. Tool Support

End-user programming. Because by definition mashups add value from the user's perspective, it should be possible for *the user* to create mashups. System allowing this kind of actions requires dynamic rather than static composition. So far, tools have been restricted in features, or, as is the case with many fully programmatic approaches, too complex for a casual user. Furthermore, while numerous systems have been introduced, it is difficult to say which of the tools are ready for prime time, and which are simply temporary experiments that will not be supported in the long run.

Plug-and-play interfacing. To enable smooth development of complete mashup applications, a library containing software components with compatible interfaces is needed. In present tools either the set of available component subsystems is restricted to those where a ready-made interfacing system exists, or the developer is assumed to be educated enough to compose at least modest pieces of glue code for integrating the ready-made services.

V. CONCLUSIONS

Ecosystems have emerged as a commonly used notion in service economy. They commonly rely on a shared platform – such as the web – on top of which different parties contribute their own, company-specific innovations, and all the participants gain the benefits of joint investment in the platform. Furthermore, to leverage the full potential of the web, the systems built out of ecosystem services often result in mashups.

Existing mashup examples and frameworks reveal the great potential of combining mobile multimedia and other content from the web. Many components needed to implement elegant and attractive mobile multimedia ecosystem have already been build. However, besides some individual advanced applications, mashups available today in general do not provide good enough quality and consistency for end-users. Usability of proposed solution discussed in this report should be further researched from this point of view.

Our literature review shows that end-user mashup programming is a field of research where numbers of tools have been introduced and reviewed. Nevertheless, especially on mobile targets, implementing truly liberating mashup programming environment is a major challenge. However, allowing end users to design their own mashups with simple enough tools would enrich the user experience significantly. Furthermore, this would most likely also result in new and innovative systems.

We anticipate a number of technical challenges related to cloud infrastructure, web services, and mobile devices when the mashup ecosystem described in this paper is implemented. However, we believe that with careful design and further work these problems will be overcome. However, sorting out the technicalities is only a beginning of the work, and its role should not be overly emphasized.

Legal as well as economic issues are important topics where further work - performed by experts in respective fields - is needed. Most mashups developed by individuals for non-commercial purposes do not have same restrictions as ones developed or supported by a commercial vendor. Because different services may set almost arbitrary terms of use for those willing to access their data, combining them into a single integrated experience is difficult, in particular when considering the whole ecosystem needed in the composition of the application. Consequently this new frontier is an area where new business models as well as opportunities will emerge.

REFERENCES

[1] J. Yu, B. Benatallah, F. Casati, F. Daniel, "Understanding mashup development," *IEEE Internet Computing*, pp. 44-52, September/October, 2008.

[2] F. Nyrhinen, A. Salminen, T. Mikkonen, A. Taivalsaari, "Lively mashups for mobile devices," *in Proceedings of the First International Conference on Mobile Computing, Applications and Services*, San Diego, California, USA, October 26-29, 2009.

[3] A. Salminen, F. Nyrhinen, T. Mikkonen, and A. Taivalsaari, "Developing client-side mashups: experiences, guidelines and the road ahead," *to appear in Proceedings of the MindTrek'2010 Conference,* Tampere, Finland, October 6-8, 2010.

[4] A. Salminen and T. Mikkonen, "Towards pervasive mashups in embedded devices," *in Proceedings of the 16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. 35-42, IEEE Computer Society, 2010.

[5] A. Brodt, D. Nicklas, "The TELAR mobile mashup platform for Nokia internet tablets," *in Proceedings of 11th International Conference on Extending Database Technology*, Munich, Germany, 2008.

[6] S. De, K. Moessner, "A framework for mobile, context-aware applications," *in Proceedings of the 16th international Conference on Telecommunications*, Marrakech, Morocco, May 25 - 27, 2009.

[7] S. Ikeda, T. Nagamine, T. Kamada, "Application framework with demand-driven mashup for selective browsing," *in Proceedings of the 10th international Conference on information integration and Web-Based Applications & Services*, Linz, Austria, November 24 - 26, 2008.

[8] R. Hashimoto, N. Ueno, and M. Shimomura, "A design of usable and secure access-control APIs for mashup applications," in *Proceedings of the 5th ACM Workshop on Digital Identity Management*, Chicago, Illinois, USA, November 13 - 13, 2009.

[9] J. Warner, and S. A. Chun, "A citizen privacy protection model for e-government mashup services," *in Proceedings of the 2008 international Conference on Digital Government Research*, Montreal, Canada, May 18 - 21, 2008.

[10] J. Magazinius, A. Askarov, and A. Sabelfeld, "A lattice-based approach to mashup security," *in Proceedings of the 5th ACM Symposium on information, Computer and Communications Security*, Beijing, China, April 13 - 16, 2010.

[11] S. Zarandioon, D. Yao, and V. Ganapathy, "Privacy-aware identity management for client-side mashup applications," *in Proceedings of the 5th ACM Workshop on Digital Identity Management*, Chicago, Illinois, USA, November 13, 2009.

[12] T. Nestler, "Towards a mashup-driven end-user programming of SOA-based applications," *in Proceedings of the 10th international Conference on information integration and Web-Based Applications & Services,* Linz, Austria, November 24 - 26, 2008.

[13] A. Taivalsaari, "Mashware: The Future of Web Applications," Sun Microsystems Laboratories Technical Report TR-2009-181, February, 2009.

[14] G. Wang, S. Yang, Y. Han, "Mashroom: end-user mashup programming using nested tables," *in Proceedings of the 18th international Conference on World Wide Web,* Madrid, Spain, April 20 - 24, 2009.

[15] M. Resnick, J. Maloney, A. Monroy-Hermandez, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, Y. Kafai, "Scratch: programming for all," *Communications of the ACM,* November, 2009.

[16] J. Cao, Y. Riche, S. Wiedenbeck, M. Burnett, V. Grigoreanu, "End-user mashup programming: through the design lens," *in Proceedings of the 28th international Conference on Human Factors in Computing Systems*, Atlanta, Georgia, USA, April 10 - 15, 2010.

[17] S. K. Card, G. G. Robinson, J. D. Mackinlay, "The information visualize, an information workspace," *in Proceedings of the SIGCHI Conference on Human Factors in Computing Systems: Reaching Through Technology*, New Orleans, Louisiana, USA, April 27 - May 02, 1991.

[18] M. Li, M. Claypool, R. Kinicki, J. Nichols, "Characteristics of streaming media stored on the Web," *ACM Transactions on Internet Technology,* 5, 4 pp. 601-626, November, 2005.

[19] B. Shao, D. Renzi, P. Amon, G. Xilouris, N. Zotos, S. Battista, A. Kourtis, M. Mattavelli, "An adaptive system for real-time scalable video streaming with end-to-end QoS control," *in Proceedings of The 11th International Workshop on Image Analysis for Multimedia Interactive Services (WIAMIS)*, Desenzano del Garda, Italy, Apr 12 - 14, 2010.

[20] R. Kuschnig, I. Kofler, H. Hellwagner, "An evaluation of TCP-based rate-control algorithms for adaptive internet streaming of H.264/SVC," *in Proceedings of the First Annual ACM SIGMM Conference on Multimedia Systems*, Phoenix, Arizona, USA, February 22 - 23, 2010.

[21] R. T. Fielding, and R. N. Taylor, "Principled design of the modern Web architecture," *ACM Trans. Internet Technol.* 2, 2, May, 2002, 115-150.

Publication III

Towards mobile multimedia mashup architecture

M. Hartikainen, A. Salminen and J. Kallio

2012 38th Euromicro Conference on Software Engineering and Advanced Applications

Towards Mobile Multimedia Mashup Architecture

Mikko Hartikainen and Arto Salminen
Department of Software Systems
Tampere University of Technology
P.O. Box 553, FIN-33101 Tampere, Finland
Email: firstname.surname@tut.fi

Jarno Kallio
PacketVideo Finland
Hallituskatu 8, FIN-33200 Tampere Finland
Email: jarno.kallio@pv.com

***Abstract*—Accessing private and public multimedia content over the web with mobile terminals can be cumbersome. To approach this problem, we introduce an ecosystem where the content is accessed with clients that are constructed as mashups. Mashups are web applications that combine already existing data into an integrated experience. In addition to traditional mashup elements such as text and images, also other types of multimedia artifacts – sound, video and even small programs – can be used as content in mashups. However, currently available methods implementing mashups do not allow effortless access to personal data across domains or provide means to ensure that the user experience is coherent. This paper describes our mobile mashup architecture aimed to solve these problems and work as a ubiquitous platform for mobile multimedia mashups. The architecture enables a mashup ecosystem where different content providers and client developers can contribute. As a practical implementation, we present a video mashup client developed for Android. In addition, we discuss the lessons learned during the research.**

***Keywords*-mashup; architecture; multimedia; mobile device**

I. Introduction

Success of social media and other collaborative services clearly indicate that content sharing directly between users or via a publishing service is a necessity for great user experiences. Moreover, multimedia content – such as videos, music and images – is commonly consumed and shared with mobile terminals. This motivates the development of a multimedia mashup ecosystem architecture that enables compelling mashup clients build on top of it.

Mashup is a web application that integrates code, data, and other content artifacts from multiple sources over the Web creating some increased value for the user. Traditionally mashups are executed inside a web browser but other systems can be used as a runtime environment as well. In general, mashups are about simplicity, usability, and ease of access [1]. A mashup can combine the content in a new, unforeseen way, thus creating entirely new web service, or a mashup can provide new visualization for an existing service.

Having a mobile terminal as a target platform is a great opportunity for mashups and other applications that are used for personal entertainment. Taking user's context into account allows superior user experiences and more relevant advertising. Sharing the experience instantly with peers enables different scenarios in constantly changing environments.

This paper extends and realizes ideas introduced in earlier publications [2] and [3]. There we provided a wide background study about mashup ecosystem requirements and challenges. In this paper we present how these challenges and ecosystem requirements were faced and fulfilled in a specialized case of mobile multimedia.

The goal of the ecosystem is ambitious. We target at enabling liberal mashupping of all relevant web content available from the most popular Web 2.0 services as well as user's own local area network. Architecture is desired to hide interface complexity and to enable coherent user experience across services. It should be possible to access the mashup content with different mobile terminals, even those with limited capabilities. Different web-enabled platforms should be possible to use as an environment for the client-side mashup implementations. The result can be projected on numerous display devices, such as other mobile devices, televisions, video projectors, public displays, set top boxes, and home theater systems. The system should take user preferences, interpreted on the basis of past actions, into account, and automatically adapt to user needs. Consequently, this paper seeks answers to the following question:

- What kind of ecosystem architecture is needed to achieve the goals above?

The paper is structured as follows. First, in Section 2, we describe the related work. In Section 3, we introduce the mashup ecosystem architecture. Section 4 provides a brief introduction to software development on Android platform, and presents the sample application implementation. In Section 5, we discuss about lessons learned and future work, especially ideas how the mashup ecosystem could be expanded. Finally, in Section 6, we draw some conclusions.

II. Related Work

Different mashup architectures for server- and client-side have already been studied. A common denominator in the existing server-side architectures is some kind of aggregator that collects information from multiple sources and provides it for the mashup backend. Another important

978-0-7695-4790-9/12 $26.00 © 2012 IEEE
DOI 10.1109/SEAA.2012.28

issue, mashup security, has been subject to research from numerous perspectives. In the following we briefly present the work done on these two areas.

A. Mashup Architectures

Database-driven mashup architecture by Vancea et al. [4] relies on an aggregator server that collects the information from multiple sources. They determine APIs for the aggregator server, called "information provider", and a database server that acts as a backend for web browser clients.

In [5] Zhang et al. described a multimedia mashup architecture that utilized the CAM4Home framework (http://www.cam4home-itea.org/) for meta data. In addition, their architecture has support for social meta data that is created by the users. A core component of the architecture is Multimedia Mashup Engine (MME) that works as an aggregator between heterogeneous service repository and mashup clients. Quality of service is ensured with the MME and a Media Aware Network Element that works as a middlebox adapting the video stream according to network conditions and terminal capabilities.

Server-side enterprise mashup architecture has been studied by Lopez et al. [6]. Their architecture consists of four layers: *source access*, *data mashup*, *widgets*, and *widget assembly*. In addition, the architecture includes *common services*, which provides general functionalities and can be used from any layer. The result mashup developed using this architecture is somewhat similar to a web portal with the exception that the widgets are connected.

In a previous research [7] we introduced a general reference architecture for client-side mashups. There we identified five components of a client-side mashup: *content providers*, *data model*, *mashup creation*, *mashup manager*, and *renderer*. These functions can be discovered in this case in the client implementation. For instance, the renderer in the system is changeable and clearly a separate module.

B. Mashup Security

In [8] Bader et al. identify numerous security aspects of enterprise mashup architecture. They point out that mashups handling sensitive data of an organization should store data strictly on organization's own storage services, as using an external storage service in a cloud might cause unwanted information disclosure. One should bear in mind that even non-sensitive data can expose delicate information when aggregated. Moreover, mashup robustness and stability are especially important in an enterprise setting. Therefore, SOA-based (Service Oriented Architecture) approach to creating mashups is recommended. Downside of the SOA-based approach is complexity for unexperienced users that may not be able to build mashups on top of a complex SOA stack. Trustworthiness of mashups is significant as well, when they are used in a business setting. Unfortunately, when mashups utilize external web resources, confirming resource trustworthiness is typically difficult or even impossible. Therefore, authors of [8] suggest utilizing a central service governance, which would control services that are used through mashups. Furthermore, when mashups are used to access organization's private data, authentication of mashup users and restricting unauthorized users from accessing sensitive data is vital.

Service access control API that aims to better mashup security has been studied by Hashimoto et al. [9]. The SAXAE API provides functions to the mashup to retrieve protected, non-public resources securely. This allows the mashup to access user's private data, for example on some social service, in secure fashion.

III. Mashup Ecosystem Architecture

As pointed out in the background study, an aggregator-based architecture has been successful in the previous solutions. Similarly to these solutions, our ecosystem architecture (illustrated in Fig. 1) utilizes a server-side aggregator. However, significant part of the mashup composing logic resides on the client-side instead of the pure server-based model where the result mashup is only rendered on a web browser. This hybrid approach has numerous benefits in our setting, where the clients are executed in mobile terminals. First, it enables to run mashups even on devices that do not support all the web technologies. For instance, most of the video content in the Web is available in *Adobe Flash* (http://www.adobe.com/software/flash/about/) or HTML5 [10] video, but the support for these technologies was limited during the study. Second, clients that are able to take user context into account can be implemented using our approach without need for extensive data transferring with the server. Third, enabling the client to work without a connection to the Web can be achieved using our hybrid model, where the client-side application can function independently to access content in the local area network or the terminal's file system.

A central component of the mashup architecture is the Mashup Aggregator Server (MAS), which is implemented as a cloud service. It crawls trough different web services and creates a database with metadata and links to web content items. In addition, it connects to different domains and can publish public content from those too. Currently, MAS supports only video content.

The Mashup Client (MC), installed into user's mobile terminal, composes the mashup from content artifacts. It communicates with the MAS and gains access directly to the web content. In addition to aggregated content, it can browse content from servers in the local area network and use it as well. The third source of content are servers in other domains, provided that the MAS has necessary information about them. Naturally, if the MC has access to local filesystem, local files can be used.

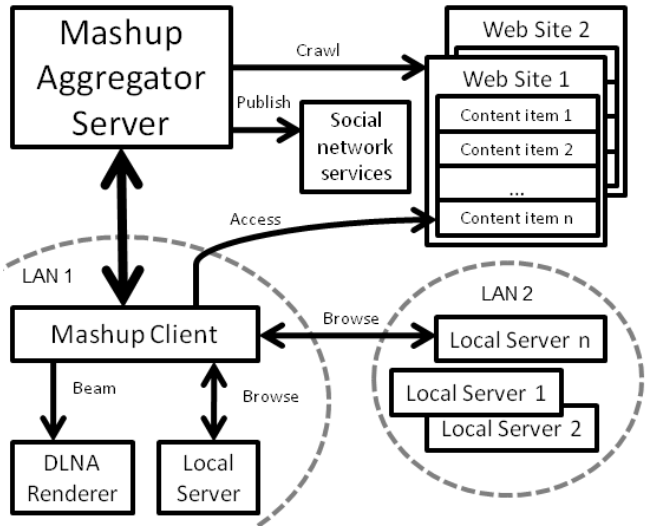


Figure 1. Mashup ecosystem architecture

Another feature that mashup clients can have is capability to utilize other displaying devices found on local network to render the result mashup. We call passing the result mashup to the rendering device "beaming". MC's beaming offers coherent way to show videos on arbitrary rendering devices with a modern graphical user interface. In the current implementation the MC can control any DLNA compatible renderer to play the mashupped videos.

Interfaces. The architecture interfaces rely loosely on REST principles [11]. Architecture can be categorized as a REST-RPC hybrid architecture [12]. Unlike pure REST architectures, where the communication between server and client is implemented with GET, POST, PUT, and DELETE methods, here only GET method is used. GET is intended to be used for safe operations, which do not change the data. Therefore, using GET for all communications has side effects, for instance how the server cache can be optimized [12]. However, it is common that Rich Internet Application (RIA) frameworks and web runtimes do not implement all HTTP methods. For instance some versions of *Adobe Flex* (http://www.adobe.com/products/flex/), *Java ME* (http://www.oracle.com/technetwork/java/javame/), and *Qt Quick* (http://qt.nokia.com/qtquick/) are lacking some of the methods. In this sense, using only GET method in the architecture interfaces is convenient even though it is not RESTful.

Mashup developer can choose between XML and JSON data formats to use them over HTTP for data exchange. This is convenient as XML is widely adopted and standardized by W3C, and parsers for XML are often readily available in different runtime environments. However, using data-oriented JSON has certain advantages in contrast to XML which is more document-oriented. One benefit that JSON has is the smaller data exchange overhead, which is important when transferring data over mobile networks where the network traffic can be expensive.

Portability. Portability has been a design goal for MAS. Because of the interfaces of MAS have been designed using standard web technologies such as JSON, XML, and HTTP, mashup clients can be implemented on any web-enabled platform with a capability to render HTML. Some MAS functionality requires showing rich media content and in this case HTML is used. However, if native technologies are used, a client needs to be reimplemented for mobile terminals with different operating systems, which can be considered as a downside. This can be avoided by utilizing some cross-platform technology for the client-side mashup.

MAS provides several URLs for content items (currently videos) to enable better support for different kinds of clients and rendering devices. Each video can be in various formats. Choosing the correct URL depends on many factors described as follows.

- *Resolution*. Tablets and external displays have much bigger resolution than low-end mobile devices. Choosing a video based on resolution will save processing power, battery life and consumes less bandwidth. Resolution can typically vary from 240x320 to 1920x1080 (27 times difference in the amount of pixels).
- *Bandwidth*. Slow mobile network connections are not always enough to stream high quality videos. Thus, lower quality video could be more suitable when the bandwidth is low.
- *Media Format*. Mobile terminal's native video player supports typically only limited amount of codecs. Support varies based on the runtime environment used. For instance, different browsers support HTML5 video in different codecs, and in Android devices the support varies between operating system versions and manufacturers.
- *Player*. The best video format can differ based on the player used. For example, videos in Adobe Flash format are suitable for desktop, but support for Flash in mobile devices is limited. Therefore, a client needs to use different URL for playing a video in mobile device and for sharing the video link to social media sites or beaming to a rendering device.
- *Location*. Service providers can restrict some videos to be shown in a limited amount of countries, and MAS as well as clients can be located in different countries. Therefore, MAS can crawl videos from service providers that are not available to all clients due to geolocation restrictions.
- *Streaming protocol*. Videos typically use either RTSP [13] or HTTP Progressive download protocol [14] for streaming. Some implementations of mobile device video players do not support RTSP URLs.

Performance. The architecture addresses the problem of varying client performance by providing multiple resource URLs for the client to choose from as we described above. At this point, videos are not transcoded and therefore there will be no delay caused by transcoding. Response time of the MAS is affected by the following variables:

- performance of network,
- performance of accessing MAS databases, and
- traffic caused by other users.

Security. MAS security is based on requiring authentication for clients. *HTTP digest access authentication* [15] is used for authenticating clients. This solution is relatively light-weight, but less secure when compared to strong authentication protocols such as *public-key* [16] or *Kerberos* [17] authentication. However, it is more secure than traditional digest authentication schemes such as *basic access authentication* [15] or *CRAM-MD5* [18]. Digest authentication encrypts user's password and leaves content of a message unencrypted, unlike stronger protocols, which encrypts also the content of the message. This is sufficient solution in basic functionality, which does not require information about the user (e.g. searches). However, when personal information is transferred between the client and servers, stronger authentication, such as public-key or Kerberos, is necessary to be implemented. Moreover, TLS [19], which is typically implemented on top of transport layer protocols, can be used in combination with HTTP digest and, therefore, added later when needed.

Authentication is invariably required when a request is made to MAS API. MAS utilizes user accounts to identify different mashup clients, and it supports user account creation directly from the client. The account creation is implemented with a simple HTML form. Social network services, such as Facebook (http://www.facebook.com) or Twitter (http://www.twitter.com), may require authentication as well when a content artifact is shared to these services. In this case, MAS forwards authentication page from the service to the client. After authentication, the client can communicate freely to the service without need for re-authentication.

Coherent user experience. Another design target of the mashup architecture was coherent user experience of client applications. To enable this, MAS provides a common API for numerous inconsistent service interfaces and social media services. As a result, clients can have fast and uniform access to all crawled metadata. However, MAS store only metadata of web content, but not actual web content items. Thus, quality of accessing the web content items always depends on the service providers. MAS can store user specific data as well. For instance, clients can use MAS to store user's bookmarked content and playlist data. This enables accessing the bookmarked content and playlists across different clients.

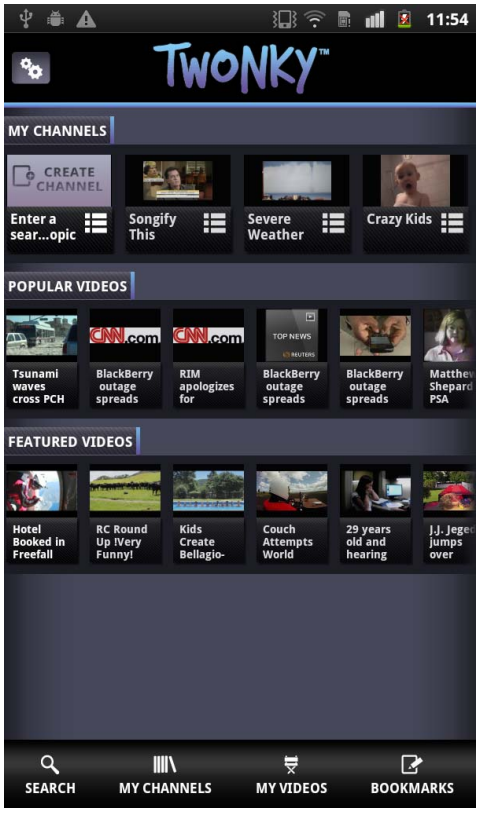


Figure 2. Video Mashup Client

IV. Video Mashup Client Implementation

Video Mashup Client (ViMC), shown in Fig. 2, is implemented as a native Android (http://www.android.com/) application. Together ViMC and MAS create a video mashup and offer fast access to video content in the Web, local network, and user's device. Naturally, ViMC can play video content in the mobile terminal itself, but it is also capable of rendering the video into any discovered DLNA compatible remote device. ViMC uses MAS for searching and browsing videos, manipulating live playlists (referred to as *channels*), storing bookmarks, and sharing content to social network services. In the following, we present the mashup client features in more detail.

Playing. By default ViMC uses Android's default video player implementation to play videos, but the client architecture allows using local 3rd party video players and remote players as well. If the default player is used, other videos related to the currently played one are displayed on the screen when the video is paused. DLNA (http://www.dlna.org/) compatible remote players (e.g. televisions, set top boxes, or public displays) can be used to show videos, and the client is able to search them from the local network.

Searching. ViMC utilizes MAS's Search API for searching for videos with keywords. The Search API offers flexible

way to aggregate arbitrary web video services under one common API, and it provides fast indexing of crawled videos. Search results can be ordered based on video meta data, and by default video view count is used as order basis. Furthermore, search operations can be done for web service content through the MAS. MAS can contain an arbitrary amount of different source services, and each of them typically has different way to access the content. Consequently, client gets several benefits from using MAS. First, less code is needed because there is no need for access more than one service. Second, user interface responsiveness improves because one call to the backend server can return (filtered and/or ordered) results from multiple services in one response.

Browsing. ViMC enables user to browse aggregated web videos through video searches such as "popular videos", "featured videos", and "more like this". The user can browse more videos related to current video. This search gives a list containing videos with matching keywords to currently played one. In addition, user can browse digital media servers (DMS) found from the local area network. For instance, all content in DLNA compatible servers can be browsed and searched for with the client.

Creating Channels. User can create a channel based on video keywords or search terms. Channels are saved into the MAS. They work like live playlists, which update items in the playlist according to the changes in the MAS. Currently, there is no support for reporting of a new content in the channel and there is no indication for user when the content is changed. Moreover, channels can contain only web content.

Bookmarking. MAS has a bookmarking mechanism that can be used to create a collection of favourite videos. Bookmarks can be fetched from MAS in alphabetical order or by date. After authentication, the bookmark collection can be accessed with ViMC. New bookmarks can be created in ViMC's playing screen. In contrast to channels that are dynamic lists of multiple videos, bookmarks are static and refer to a single video.

Sharing. MAS offers a Sharing API for sharing aggregated videos to social network services such as Twitter and Facebook. ViMC utilizes this API to allow user to share videos with other users directly in the client application. In addition to social network services, videos can be shared via e-mail and SMS by using Android's core libraries. Currently sharing means sending a link to a video to another user or publishing the link as a message in Facebook or Twitter. Benefit of using the aggregator server for accessing social network services over accessing them directly is possibility to access them through a single API. When a new social network service is introduced, there is no need to make changes to client side, as the MAS handles the communication with the service and provides the necessary interface for clients.

V. Discussion

In this section we briefly discuss about the lessons learned during the development of the mashup architecture and ViMC. In addition we briefly present some ideas for future improvements.

A. Lessons Learned

Important design targets of the system were portability and coherent user experience. The MAS provides multiple URLs through an open API for better portability, but the ViMC implementation is very platform specific. However, platform specific techniques used in the client application made the result mashup more responsive and the user experience more polished. Furthermore, the implementation revealed the need for locating more suitable URLs, which is discussed about in the following.

Locating a suitable URL. As the MAS sometimes fails providing a suitable URL for ViMC, there may be need for a method for solving a better URL for a content artifact, a video, for instance. Some service providers offer an API to locate several URLs, from which a developer can choose the URL that fits best for the current need, based on the URL provided by MAS. There is a possibility to use these APIs to located a URL, which ViMC is capable of playing.

Fig. 3, explained in the following, shows an example how the method locating more suitable URL could work.

1) First, ViMC requests a video from MAS.
2) ViMC does not support the URL for Flash video provided by MAS and begins to locate more suitable URL.
3) ViMC gets more information from service provider about the URL and tries to get a URL that best fits to ViMC needs.

However, there is a risk that when the service provider's API changes, the method stops to work, and all the videos, which depend on this service, will become unavailable.

Portability of the ViMC Even though Android applications are written in Java, hardly any code in ViMC can be used outside of Android due to two reasons. First, ViMC's UI strongly relies on Android framework. Second, ViMC does not contain business logic, which is executed on server-side. Android does support Java Native Interface, which allows Android applications to run components written in C or C++ languages. This kind of components would have been more portable to other mobile devices, which typically use C compilers.

B. Future work

Expanding the mashup architecture into other domains than video is included in our future plans. Providing links to relevant web content could provide a great user experience for mashup users. For instance, if a music video was played, a link to music store providing the song to be downloaded and articles discussing the artist could be shown.

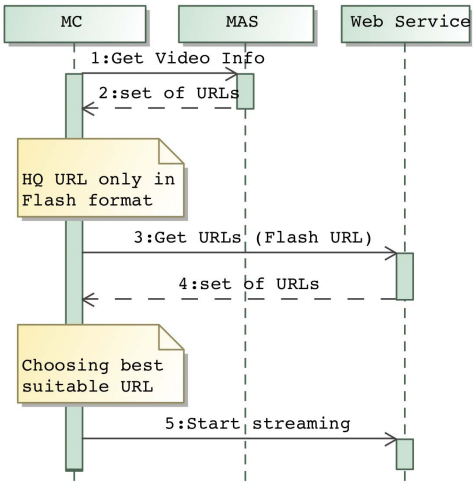


Figure 3. Method for locating a more suitable URL

Many of the challenges discussed earlier [2] could be tackled by using video transcoding. Transcoding is a process where one multimedia object is converted into another. In this case it would mean re-encoding a video into more suitable format. According to [20], three transcoding categories can be identified based on the location of the operation: 1) server transcoding (i.e. external web service), 2) intermediate proxy transcoding (i.e. MAS) or 3) client transcoding (i.e. MC). In our architecture, we have no desire to use an external web service, and transcoding needs to be located in MAS or MC. Due to capabilities and low network bandwidth of mobile networks, transcoding is not typically done in the mobile device [21]. Intermediate proxy transcoding could be suitable in our case and can be seen as subject of further study.

Unlike ViMC, MAS has been implemented with cross-platform compatibility in mind. It utilizes open standards such as HTTP, REST, JSON, and XML, which makes possible to create client applications on practically any web-enabled platform available. Interesting possibility – that we plan to explore further – is creating the mashup client with regular web application development technologies including HTML, JavaScript, and CSS. This would allow reusing the same code base extensively in different clients. Furthermore, the upcoming HTML5 standard [10] includes an element for presenting video content, and we expect that utilizing this technology in client implementations could reduce the need to support multiple video formats as well as need to perform video transcoding.

VI. Conclusion

In this paper, we described our mobile multimedia mashup ecosystem architecture and implementation of a video mashup client for Android platform. In addition, we briefly discussed about the lessons learned during the development of the mashup client.

Our target was set to form an ecosystem that enables composing mashups of all relevant web content available. Thanks to the aggregator-based hybrid architecture, the ecosystem can be diversified with new web services and mashup clients. The architecture enables mashup clients to work without constant network connection as well as take user's context into account without communicating it to the server-side. Coherent user experience was pursued with MAS providing a set of URLs for each web content artifact, which helped to tackle with varying user operating situations and device configurations.

The smart mashup client is capable of accessing videos provided by numerous web services. In addition, videos from the local area network or terminal's file system can be accessed as well. The mashup adaptively suggests new videos for the user based on his/hers earlier actions, and provides means for bookmarking videos and creating playlists out of them. The result mashup can be beamed into any renderer device with DLNA compatibility.

Our project has been successful in the domain of web videos. Expanding the ecosystem to include other forms of multimedia is in the scope of our future work. Another area where further work is needed is video transcoding. Furthermore, developing more cross-platform solution for the client-side with upcoming HTML5 technologies is an interesting possibility that requires additional research.

References

[1] J. Yu, B. Benatallah, F. Casati, F. Daniel, "Understanding mashup development," IEEE Internet Computing, pp. 44-52, Sept./Oct. 2008.

[2] A. Salminen, J. Kallio and T. Mikkonen, "Towards mobile multimedia mashup ecosystem," In Proceedings of IEEE ICC 2011 Workshop on Advances in Mobile Networking - "Towards a Next Generation Mobile Core Network", pp. 1–5, June 2011.

[3] A. Salminen and T. Mikkonen, "Mashups – Software ecosystems for the web era," In Proceedings of the 4th Workshop on Software Ecosystems (IWSECO 2012). Springer, June 2012.

[4] A. Vancea, M. Grossniklaus, and M. C. Norrie, "Database-driven web mashups," In Proceedings of the Eighth International Conference on Web Engineering (ICWE '08). IEEE Computer Society, pp. 162–174, 2008.

[5] H. Zhang, Z. Zhao, S. Sivasothy, C. Huang, and N. Crespi, "Quality-assured and sociality-enriched multimedia mobile mashup," EURASIP Journal on Wireless Communications and Networking, vol. 2010, Article ID 721312, 11 pages, 2010.

[6] J. López, A. Pan, F. Bellas and P. Montoto, "Towards a reference architecture for enterprise mashups". Architecture, 2(2), pp. 67–76, 2008.

[7] T. Mikkonen and A. Salminen, “Towards a reference architecture for mashups,” In Proceedings of the 10th Confederated international conference on On the move to meaningful internet systems, Second International Workshop on Variability, Adaptation and Dynamism in software systEms and seRvices (VADER 2011), R. Meersman, T. Dillon, and P. Herrero (Eds.). Springer-Verlag, Berlin, Heidelberg, pp. 647–656, 2011.

[8] G. Bader, A. Anjomshoaa, and A. M. Tjoa, “Privacy aspects of mashup architecture,” in Proceedings of Social Computing / IEEE International Conference on Privacy, Security, Risk and Trust, pp. 1141–1146, 2010.

[9] R. Hashimoto, N. Ueno, and M. Shimomura, “A design of usable and secure access-control APIs for mashup applications,” in Proceedings of the 5th ACM Workshop on Digital Identity Management, Nov. 2009.

[10] I. Hickson (Ed.), “HTML5 – A vocabulary and associated APIs for HTML and XHTML,” W3C Working Draft (http://www.w3.org/TR/2012/WD-html5-20120329/), March 2012.

[11] R.T. Fielding and R.N. Taylor, “Principled design of the modern web architecture,” ACM Trans. Internet Technol. (TOIT) 2 (2), pp. 115–150, 2002.

[12] L. Richardson and S. Ruby. Restful web services (First ed.). O’Reilly, 2007.

[13] H. Schulzrinne, A. Rao and R. Lanphier, “RFC 2326: Real Time Streaming Protocol (RTSP),” April 1998.

[14] D. Kaspar, K. Evensen, P. Engelstad, A.F. Hansen, P. Halvorsen, and C. Griwodz, “Enhancing video-on-demand playout over multiple heterogeneous access networks,” Consumer Communications and Networking Conference (CCNC), pp. 1–5, Jan. 2010.

[15] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart, “RFC 2617: HTTP authentication: basic and digest access authentication,” June 1999.

[16] W. Diffie and M. Hellman, “New directions in cryptography,” IEEE Transactions on Information Theory, Vol. IT-22, No. 6., pp. 644-654, 1976.

[17] C. Neuman, T. Yu, S. Hartman, and K. Raeburn, “RFC 4120: The Kerberos network authentication service (V5),” July 2005.

[18] J. Klensin, R. Catoe, and P. Krumviede, “RFC 2195: IMAP/POP AUTHorize extension for simple challenge/response,” Sept. 1997.

[19] Dierks, T, and E Rescorla. “RFC 5246 - The Transport Layer Security (TLS) Protocol Version 1.2.” IETF 2008.

[20] V. Cardellini, P.-S. Yu, and Y.-W. Huang, “Collaborative proxy system for distributed web content transcoding,” in ACM CIKM, Mar. 2000.

[21] H. Bharadvaj, A. Joshi, and S. Auephanwiriyakul, “An active transcoding proxy to support mobile web access,” in Proceedings of 17th IEEE Symp. Reliable Distributed Systems, 1998.

Publication IV

Lively mashups for mobile devices

F. Nyrhinen, A. Salminen,
T. Mikkonen and A. Taivalsaari

Lively Mashups for Mobile Devices

Feetu Nyrhinen[1], Arto Salminen[1], Tommi Mikkonen[1], and Antero Taivalsaari[2]

[1] Tampere University of Technology, Korkeakoulunkatu 1, FI-33720 Tampere, Finland
{feetu.nyrhinen,arto.salminen,tommi.mikkonen}@tut.fi
[2] Sun Microsystems Laboratories, P.O. Box 553 (TUT), FI-33101 Tampere, Finland
antero.taivalsaari@sun.com

Abstract. The software industry is currently experiencing a paradigm shift towards web-based software and web-enabled mobile devices. With the Web as the ultimate information distribution platform, mashups that combine data, code and other content from numerous web sites are becoming popular. Unfortunately, there are various limitations when building mashups that run in a web browser. The problems are even more challenging when using those mashups on mobile devices. In this paper, we present our experiences in building mashups using *Qt*, a Nokia-owned cross-platform application framework that provides built-in support for web browsing and scripting. These experiences are part of a larger activity called *Lively for Qt*, an effort that has created a highly interactive, mobile web application and mashup development environment on top of the Qt framework.

Keywords: mobile web applications, mashup development, Qt, Lively for Qt.

1 Introduction

In the past few years, the Web has become a popular deployment environment for new software systems and applications such as word processors, spreadsheets, calendars and games. In the new era of web-based software, applications live on the Web as services. They consist of data, code and other resources that can be located anywhere in the world. Furthermore, they require no installation or manual upgrades. Ideally, applications should also support user collaboration, i.e. allow multiple users to interact and share the same applications and data over the Internet.

An important realization about web applications is that they do not have to live by the same constraints that characterized the evolution of conventional desktop software. The ability to instantly publish software worldwide, and the ability to dynamically combine data, code and other content from numerous web sites all over the world will open up entirely new possibilities for software development.

In web terminology, a web site that combines ("mashes up") content from more than one source is commonly referred to as a *mashup*. Mashups are content aggregates that leverage the power of the Web to support instant, worldwide sharing of content. Typical examples of mashups are web sites that combine photographs or maps taken from one site with other data (e.g., news, blog entries, weather or traffic information, or price comparison data) that are overlaid on top of the map or photo.

S. Ystad et al. (Eds.): MobiCASE 2009, LNICST 35, pp. 123–141, 2010.

Mashups usually run inside a web browser. However, because the web browser was originally designed to be a document viewing tool – not an environment for highly interactive applications – there are challenges when running web applications and mashups that behave in a highly interactive fashion. Support for user interface widgets can also be limited. Furthermore, poor performance of the web browser can be a major issue especially when running mashups in mobile devices. On mobile devices, usability issues cannot be ignored either [9].

In this paper, we present our experiences in developing practical, compelling web mashups, with a special emphasis on making those mashups work well on mobile devices. The work reported here is part of a larger activity called *Lively for Qt* (http://lively.cs.tut.fi/qt) – a project that has created a highly interactive, mobile web application and mashup development environment for the Qt cross-platform application framework (http://www.qtsoftware.com/). The Qt framework was recently acquired by Nokia, and versions of the framework have already been announced for Nokia's device platforms.

The rest of the paper is structured as follows. In Section 2 we summarize the existing environments and tools available for mashup development. In Section 3 we provide an overview of the Qt platform from the viewpoint of web application and mashup development. Section 4 contains a description of the most interesting mashups and other applications that we have written, including some source code of one of the applications. In Section 5, we discuss our experiences and lessons learned during the development of those mashups. Finally, Section 6 concludes the paper and outlines some future directions.

2 Existing Mashup Development Environments and Tools

The landscape of mashup development technologies is still rather diverse, reflecting the rapidly evolving state of the art in web development. Since mashups are usually built on top of existing content available on the Web, mashups can be composed manually using the classic DHTML technologies available in every commercial web browser: HTML, Cascading Style Sheets (CSS), JavaScript and the Document Object Model (DOM) [6]. However, since the actual representation of data, behavior and content can vary dramatically between different web sites, manual mashup construction can be extremely tedious, fragile and error-prone. For instance, since web sites do not generally present any well-defined interfaces that would clearly separate the public parts of the sites from their implementation details, there are usually few guarantees that the behavior and the data representations used by those web sites would remain the same over time.

To facilitate mashup development, a number of tools are available. In principle, mashups can be developed using general-purpose web application development platforms such as Adobe AIR [15], Google Web Toolkit [8], Microsoft Silverlight [12] and Sun Microsystems' JavaFX [1]. However, in practice the capabilities of these general-purpose web programming environments are still somewhat limited when it comes to the flexible extraction and combination of data from different web sites. The same comment applies also to general-purpose web content development tools including for example Adobe Creative Suite (http://www.adobe.com/products/creativesuite/) and Microsoft Expression (http://www.microsoft.com/expression/).

There are a number of existing tools that have been designed specifically for mashup development. Such tools include (in alphabetical order):

- Google Mashup Editor (http://code.google.com/gme/),
- IBM Mashup Center (http://www.ibm.com/software/info/mashup-center/),
- Intel Mash Maker (http://mashmaker.intel.com/),
- Microsoft Popfly (http://www.popfly.com/),
- Open Mashups Studio (http://www.open-mashups.org/),
- Yahoo Pipes (http://pipes.yahoo.com/).

We have reported our experiences in using these systems in an earlier paper [13]. In analyzing the systems, some common themes and trends have started to emerge. Such trends include:

- *Using the web browser not only to run applications/mashups but also to develop them*. For instance, Google Mashup Editor, Microsoft Popfly and Yahoo Pipes use the web browser to host the development environment and to provide seamless transition between the development and use of the mashups.
- *Using visual programming techniques to facilitate end-user development*. Visual "tile scripting" and "program by wire" environments are provided, e.g., by Microsoft Popfly and Yahoo Pipes.
- *Using the web server to host and share the created mashups*. Most of the mashup development tools mentioned above store the created mashups and applications on a web server that is hosted by the service provider.
- *Direct hook-ups to various existing web services*. Since the Web itself does not provide enough semantic information or well-defined interfaces to access information in web sites in a generalized fashion, most of the mashup development tools include custom-built hook-ups to existing web services such as Digg, Facebook, Flickr, Google Maps, Picasa, Twitter, Yahoo Traffic and various RSS newsfeeds.

So far, very little attention has been put on optimizing mashup development for mobile devices. It should also be mentioned that most of the above listed mashup development tools are still under development, e.g., in beta or some other pre-release stage, reflecting the rapidly evolving state of the art in mashup development. Nevertheless, many of the systems are already quite advanced and capable, and – perhaps most importantly – a lot of fun even for children to use.

3 Qt as a Mashup Platform

As part of the broader *Lively for Qt* (http://lively.cs.tut.fi/qt) activity mentioned earlier, we have created a dynamic, cross-platform mashup environment based on the Qt application framework. The broader *Lively for Qt* activity is reported in a separate paper [10]. In this section we provide an introduction to Qt, with a particular emphasis on its suitability for mashup development.

3.1 Introduction to Qt

Qt (http://www.qtsoftware.com/) is a mature, well-documented cross-platform application framework that has been under development since the early 1990s. Qt supports a rich

set of APIs, widgets and tools that run on most commercial software platforms, including Mac OS X, Linux and Windows. In addition, Qt is available for mobile devices based on Nokia's Maemo Linux platform (http://maemo.org/) and Series 60 Symbian platform (http://www.s60.com/). Qt has been used in various commercial applications before. Examples of desktop applications built with Qt include Adobe Photoshop Elements, Google Earth, Skype, and the KDE desktop environment for the Linux operating system. In addition, Qt has been used in various embedded devices and applications, including mobile phones, PDAs, GPS receivers and handheld media players.

Trolltech, the company developing Qt, was acquired by Nokia in 2008. Nokia is currently in the process of making Qt libraries available on their phone platforms. Nokia's market share will make Qt an extremely interesting target platform for mobile applications as well.

From the technical viewpoint, Qt is primarily a GUI framework that includes a rich set of widgets, graphics rendering APIs, layout and stylesheet mechanisms and associated tools that can be used for creating compelling user interfaces that run in a wide array of target platforms. Qt widgets range from simple objects such as push buttons and labels to advanced widgets such as full-fledged text editors, calendars, and objects that host a complete web browser. Dozens and dozens of widget types are supported.

The GUI features of Qt adapt to the native look-and-feel of the target platform. For instance, on Mac OS X, all the widgets look like native Macintosh widgets, while on Windows applications utilizing the same widgets will look like native Windows applications. An essential part in enabling cross-platform GUI behavior is flexible support for widget positioning using *layouts*. Qt's layout components can adapt to different sizes, styles and fonts used by the host operating system. In general, automated layouts give significant advantage when a program is translated to other platforms and languages. The program adapts automatically to changed text sizes and resizes widgets in an aesthetically pleasant way. Additionally, since Qt supports full *internationalization*, all the locale-specific components (such as a calendar widget) automatically adapt to the current regional settings of the target platform.

In addition to its GUI capabilities, Qt has classes for networking, file access, database access, text processing, XML parsing and many other useful tasks. A multimedia framework called *Phonon* is included to support audio and video playback. Qt networking libraries provide support for *asynchronous HTTP communication* familiar from Ajax [2]. Asynchronous networking support is critical in building web applications that do not block their user interface while networking requests are in progress.

3.2 Qt and Web Development

What makes Qt relevant from the viewpoint of web development is that Qt libraries include a complete web browser based on the *WebKit* (http://webkit.org/) browser engine. The necessary DOM and XML APIs are also included to parse, manipulate and generate new web content easily. In addition, Qt includes a fully functional ECMAScript [4] (JavaScript) engine called *QtScript*. The presence of a JavaScript engine is important, since JavaScript – along with XML – is the *lingua franca* of the Web that is used by popular web service APIs such as the Google Maps API [5].

The web browser integration in Qt works in a number of different ways. For instance, it is possible to instantiate any number of web browsers inside a Qt application

using the *QWebView* API. The *QWebView* class provides a widget that can be used to view and edit web documents inside applications. The data in web documents can be manipulated using the built-in DOM and XML APIs.

To support Qt applications in any web browser, a plugin called *QtBrowserPlugin* exists for embedding the Qt environment into any commercial web browser such as Mozilla Firefox or Apple Safari. The plugin makes it possible to run Qt applications inside a web browser, either as standalone Rich Internet Applications or alongside (or embedded in) conventional DHTML and Ajax web content.

JavaScript support in Qt is available both inside and outside the web browser. By default, the QtScript engine can only access those APIs that are part of the ECMAScript Specification [4]. However, by using a tool called *QtScriptGenerator* bindings to all the Qt APIs can be made visible to the JavaScript engine. This makes it possible to create JavaScript applications that combine classic DHTML behavior with widgets and other APIs offered by Qt.

3.3 Using Qt for Mashup Development

With Qt and its built-in web browser and JavaScript support, we have created a dynamic mashup environment that makes it possible to create mashups that can run inside the web browser as well as native desktop or mobile "phonetop" applications. The mashups are written in JavaScript, and they communicate with existing web services using asynchronous networking.

The mashups can leverage the rich Qt APIs for information visualization and processing. This is important since mashup development commonly relies on a plethora of data formats used by different web sites and services. In addition to binary image and video formats such as GIF, JPEG, PNG and MPEG-4, textual representations such as XML, CSV (Comma-Separated Value format), JSON (JavaScript Object Notation) and plain JavaScript source code play a central role in enabling the reuse of web content and scripts in new contexts. Qt provides excellent capabilities for processing such information, especially when combined with a dynamic language such as JavaScript that allows new object types to be constructed on the fly to accommodate the different data formats.

4 Sample Mashups

In this section we summarize the mashups that we have developed for our Lively for Qt system. First, we provide an example that includes source code as well. Then, we present two mashups that have been built on top of the Google Maps API [5]. Finally, we introduce some other types of mashups. All our mashups run on desktop computers (inside and outside the web browser), as well as in the Nokia N810 mobile device – a handheld WiFi webpad built around Nokia's Maemo Linux platform.

In the application descriptions below, some of the screen snapshots have been taken on the Nokia N810 device. For improved viewability, some of the snapshots have been taken on a PC. Further information on these applications is available on our website (http://lively.cs.tut.fi/qt).

4.1 QtFlickr: Animated Flickr Photo Viewer

In order to demonstrate mashup development with Qt, this section provides a simple example that includes source code. The application used here is called *QtFlickr* – a photo viewer application that fetches images from *Flickr* (http://www.flickr.com/) photo service based on keywords (photo tags) that are obtained automatically from the *Twitter* (http://www.twitter.com/) microblogging service, based on current Twitter trends (http://twitter.com/trends). Images are displayed using timer-based animation (rotation).

The general idea of this application is to automatically display images that reflect the most actively microblogged topics in the world. For instance, when the screenshot of the application shown in Figure 1 was taken, the most actively discussed topic in the world was the swine flu (H1N1).

Fig. 1. *QtFlickr* application running on a PC

When the *QtFlickr* application is started, it first obtains a list of the current microblogging trends from Twitter. Then, the application fetches images from Flickr using the trend names as photo tags. The actual loading of trends and images is performed asynchronously using *QNetworkRequest* and *QNetworkAccessManager* classes so that the user does not have to wait while data is being loaded. The image feed from Flickr is parsed using the *QXmlStreamReader* class. The image URLs contained within the feed are stored and the images to be shown are chosen randomly. Initially, each image is scaled according to the size of the application window. To simplify the implementation and to shorten the source code, only one image is displayed at a time.

Source code. The source code of the application's main class definition is shown in Listing 1. Note that this source code is ECMAScript (ECMA standard 262 [4]) code without any additional syntactic sugar. The *Lively for Qt* includes the option to also use the more class-oriented syntax defined by the *Prototype* JavaScript library (http://www.prototypejs.org/).

The main function of the application, *FlickrWidget*, defines the photo viewer class and its constructor. The class is defined as a subclass of Qt's class *QWidget*, allowing

the application to flexibly behave both as a standalone main application (main window) as well as a widget that can be embedded in other Qt components.

The *FlickrWidget* constructor sets up the UI components and connects the components to the required actions. Two layout components are created to arrange widgets within the application window. A *QHBoxLayout* instance is used for horizontally lining up the *QLabel* widgets shown at the top of the application window. A *QVBoxLayout* object then vertically arranges the *QHBoxLayout* object and the *QLabel* object holding the image (*QPixmap*) to be displayed. Two separate *QPixmap* objects are used for images: the first one holds the current image and the second one the image to be displayed next.

```
function FlickrWidget(parent) {

    // FlickrWidget is a subclass of QWidget
    QWidget.call(this, parent);

    // The image references
    this.flickrUrl = 'http://api.flickr.com/'
    +'services/feeds/'
      +'photos_public.gne?format=rss2';
    this.imageUrls = new Array();

    // The visible UI components
    this.currentTagLabel = new QLabel("", this);
    this.imageLabel = new QLabel(this);
    this.imageLabel.setSizePolicy(
    QSizePolicy.Ignored, QSizePolicy.Ignored);
    this.imageLabel.alignment = Qt.AlignCenter;

    this.imagePixmap = new QPixmap();
    this.nextPixmap = new QPixmap();

    // The timers for downloading and rotation
    this.changeTagsTimer = new QTimer(this);
    this.changeTagsTimer["timeout"].connect(this,
    this.changeTagsTimerTimeout);
    this.changeTagsTimer.start(30000); // 30 seconds

    this.fetchImageTimer = new QTimer(this);
    this.fetchImageTimer["timeout"].connect(this,
    this.fetchImageTimerTimeout);

    this.rotTimer = new QTimer(this);
    this.rotTimer["timeout"].connect(this,
    this.rotTimerTimeout);

    this.angle = 90;

    // The layout components
    var hBoxLayout = new QHBoxLayout();
    hBoxLayout.addWidget(new QLabel("Tags:"),0,0);
    hBoxLayout.addWidget(this.currentTagLabel,1,0);
    this.layout = new QVBoxLayout();
    this.layout.addLayout(hBoxLayout);
    this.layout.addWidget(this.imageLabel,1,0);

    this.resize(300,300);
    this.getTwitterTrends();
}
```

Listing 1. The main function (JavaScript class) *FlickrWidget*

Three *QTimer* timer objects are utilized to execute functions in regular intervals. The first timer called *changeTagsTimer* handles the downloading of image tags from Twitter. The second *QTimer* called *fetchImageTimer* is used for downloading the next image from Flickr on the background after the current image has been displayed for five seconds. The third timer called *rotTimer* is used for rotating the current image. Qt's *connect* function is used for creating the connections between the timers and the callback functions that are invoked when the timers are triggered.

When the timer named *changeTagsTimer* timeouts, the function *getTwitterTrends*, shown in Listing 2, is called to downloads the current Twitter trends. At first a URL pointing to trend file (a JSON file available from Twitter's web site) is defined. The actual asynchronous HTTP GET request is sent using the class *QNetworkAccessManager*.

```
FlickrWidget.prototype.getTwitterTrends =
function() {
  var url = 'http://search.twitter.com/'
    + 'trends.json';
  var accessMgr = new QNetworkAccessManager(this);
  accessMgr["finished(QNetworkReply*)"].connect(
    this, twitterReplyFinished);
  accessMgr.get(new QNetworkRequest(
    new QUrl(url)));
}
```

Listing 2. Function *getTwitterTrends*

Parameter *twitterReplyFinished* defines the callback function that will be called when the asynchronous network request has been completed. The function *twitterReplyFinished*, shown in Listing 3, processes the JSON file that contains a list of Twitter trends. The JSON string is parsed and the tags in it are stored in an array. When the tags have been obtained, the function *loadFeed* is invoked to load images from Flickr.

```
twitterReplyFinished = function(reply) {
  var trendJSONString =
    reply.readAll().toString();
  var trendJSONObject =
    eval('(' + trendJSONString + ')');
  var tags = new Array();
  for(i=0;i<trendJSONObject.trends.length;i=i+1) {
    tags.push(trendJSONObject.trends[i].name);
    }
  this.loadFeed(tags);
}
```

Listing 3. Function *twitterReplyFinished*

The function *loadFeed*, presented in Listing 4, handles the loading of the Flickr XML feed containing image URLs. At first a URL for the HTTP GET request is constructed by adding the user's search terms (image tags) to it. This is the URL that is sent to the Flickr web service to obtain images. The network request and the callback functionality are created and handled in a manner that is analogous to the functions that were used for downloading Twitter data.

The function *flickrReplyFinished*, presented in Listing 5, reads the contents of the HTTP reply utilizing the *QXmlStreamReader* class. The image URLs found in the HTTP reply are parsed and stored in the *imageUrls* array. The actual images are then downloaded using a function called *showRandomImage*. Its behavior is analogous to the *loadFeed* function, so the code is not presented here.

```
FlickrWidget.prototype.loadFeed = function(tags) {
    this.imageUrls = [];
    var currentTag = tags[Math.floor(
      Math.random()*tags.length)];
    var url = this.flickrUrl +"&tags=" +currentTag;
    this.currentTagLabel.text = currentTag;
    var accessMgr = new
    QNetworkAccessManager(this);

    accessMgr["finished(QNetworkReply*)"].connect(
      this, flickrReplyFinished);
      accessMgr.get(
        new QNetworkRequest(new QUrl(url)));
  }
```

Listing 4. Function *loadFeed*

To support image animation (rotation), the *QTimer* object stored in the *rotTimer* variable invokes a function called *rotTimerTimeout* (shown in Listing 6) every 50 milliseconds. The function utilizes a *QTransform* object to rotate the currently displayed image. Qt's fast transformation mode is used instead of smooth transformation to improve animation performance on mobile devices at the cost of the quality of the displayed images. If the angle of rotation is 90 or 270 degrees, the image is projected sideways and is invisible to the user. At that point the image can be switched to the next one.

```
flickrReplyFinished = function(reply) {
  var xml = new QXmlStreamReader();
  xml.addData(reply.readAll());

  while (!xml.atEnd()) {
    xml.readNext();
    if (xml.isStartElement()) {
      if (xml.name() == "enclosure") {
      this.imageUrls.push(
        xml.attributes().value("url").toString());
      }
    }
  }
  /* fetch next image after 3 seconds */
  this.fetchImageTimer.start(3000);
}
```

Listing 5. Function *flickrReplyFinished*

Since image rotation is rather computation-intensive, it is not well suited to low-end mobile devices. We have used it in this application, because it gives a rather

realistic view of the limited processing power and the graphics capabilities of the mobile device and its software stack.

```
FlickrWidget.prototype.rotTimerTimeout=function(){

  // When current image is drawn sideways,
  // switch to the next image
  if (this.angle % 90  == 0 ||
      this.angle % 270 == 0) {

    // Switch to the next image
    this.imagePixmap =
      new QPixmap(this.nextPixmap);
  }

  // Perform image rotation
  var trans = new QTransform();
  trans.rotate(this.angle, Qt.YAxis);
  trans.rotate(this.angle, Qt.ZAxis);

  // Display the image
  this.imageLabel.setPixmap(
    this.imagePixmap.transformed(
      trans, Qt.FastTransformation));
}
```

Listing 6. Function *rotTimerTimeout*

4.2 QtWeatherCameras: Live Road Weather

QtWeatherCameras is a mashup that utilizes the Google Maps JavaScript API and the road weather camera information available from Finnish Road Administration (http://www.tiehallinto.fi) – the government branch in Finland that is responsible for the highway network and road maintenance. The application utilizes the Google Maps API to calculate an optimal route between two chosen points on the map of Finland. The application then obtains information about the nearest road weather cameras along the route, and displays those cameras as markers on the map (see Figure 2). When the user clicks on any of the markers on the map, a live image and current weather conditions from the selected camera are fetched.

The displayed weather conditions include air temperature, road surface temperature and rain measurements. The weather camera image and weather conditions are displayed using a collapsible, semi-transparent widget placed over the map. The map underneath can be panned and zoomed freely.

At the implementation level, the *QtWeatherCameras* mashup uses Qt's *QWebView* web browser widget that has been placed in a *QVBoxLayout* layout component to allow smooth resizing of the application. The *QWebView* widget displays the map images from the Google Maps API.

After the user has selected a weather camera from the map, the application opens a semi-transparent widget called *ImageViewer* on top of the main widget. The widget consist of two *QLabel* components, a *QTextBrowser* and a *QPushButton* widget. The first *QLabel* is used for displaying the name of the weather camera. The image of the

camera is loaded into the second *QLabel* widget. The *QTextBrowser* widget is used for displaying weather conditions from the nearest weather station. Although it looks like a simple text box, the widget is a full-fledged rich text browser that accepts any HTML-formatted string as a parameter, and also supports hypertext navigation. The *QPushButton* widget is used for minimizing and expanding the *ImageViewer* widget. In addition, the mashup utilizes a widget *styleSheet* property that defines the customizations to the widgets' style, including their transparency.

Fig. 2. *QtWeatherCameras* application running on a PC

Images and weather information are downloaded from the web server of Finnish Road Administration. Because the Finnish Road Administration does not provide any well-defined API for accessing the weather camera information, rather heavy parsing is needed in the *QtWeatherCameras* application to digest the information. Locations of road weather cameras are loaded into an array upon application startup. When a new route is created by the user, the application performs the selection of the nearest weather cameras and weather stations locally using the preloaded information. When the user clicks a marker representing a camera, web page containing camera data is loaded and parsed. Weather camera image will be passed on to the *ImageViewer* widget. The data from the nearest weather station is also parsed and passed to the *ImageViewer*.

4.3 QtMapNews: Geotagged RSS Feed Viewer

QtMapNews is a mashup that displays geotagged news items and other geotagged information utilizing the Google Maps API (see Figure 3). The application includes a *QTreeWidget* (tree view) component that lists a selection of predefined geotagged RSS feeds:

- *Earthquakes*: All the Magnitude 5 or greater earthquakes in the world in the past seven days.
- *Emergencies*: The last one hundred incidents/emergencies in Finland based on information available from *Finnish Rescue Service* (http://www.pelastustoimi.fi).
- *News*: Geotagged news from CNN, Yahoo and Yle (Finnish Broadcasting Service).

Fig. 3. *QtMapNews* application running on Nokia N810

The user can add more RSS feeds by pressing a QPushButton labeled "Add...", which will open a simple dialog to enter a new RSS feed. If the new feed is not a geo-coded GeoRSS feed, the QtMapNews application uses a publicly available RSS to GeoRSS converter service (http://www.geonames.org/rss-to-georss-converter.html) to geocode news items contained within the RSS feed. After the geocoding process, the items in the feed are displayed on the map as markers. When the user clicks on a marker, an overview of the news item is displayed on the map.

An interesting additional feature of the *QtMapNews* application is that it includes an embedded web browser to display more detailed information. Whenever the user clicks on a map item that contains an URL, a web browser view is opened inside the *QtMapNews* application (on top of the map) to display the contents of that web page. The web browser is implemented using a *QWebView* widget that is displayed on top of the map view when necessary.

4.4 QtScrapBook: Web Camera Scrapbook

QtScrapBook mashup (Figure 4) is a visual scrapbook that can be used for collecting and displaying static or dynamic images from the Web. The application is intended primarily for keeping track of the user's favorite web cameras. The application uses tabs (a *QTabWidget* object) to display multiple images from the web or local file system. To conserve screen space and to allow the application to be used on a mobile device, only a single image (a single tab) is displayed at a time. Images are updated on regular intervals based on a user-defined interval value that can be adjusted using a *QSlider* widget. A *QTimer* object is used for keeping track of time between updates. Images are fetched asynchronously utilizing the *QNetworkRequest* and *QNetworkAccessManager* classes.

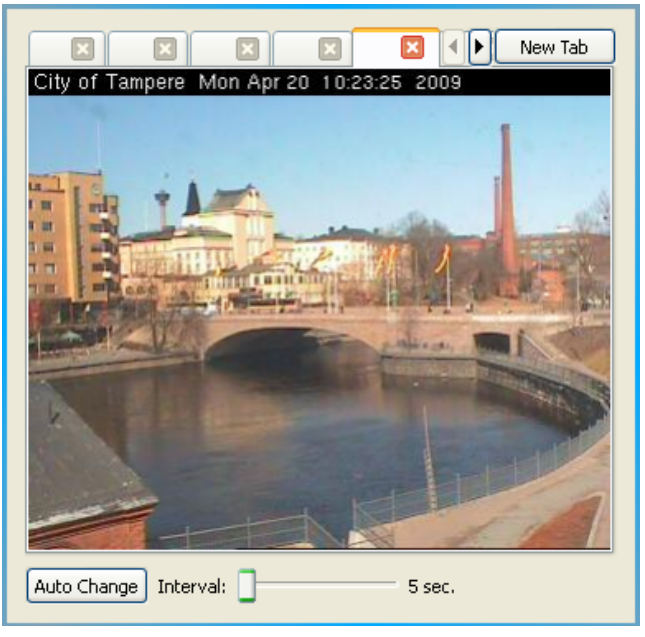


Fig. 4. *QtScrapBook* application running on a PC

The user can add new images and webcams to the application by dragging them from a web browser or from the file explorer of the host operating system. At the implementation level, this is accomplished using Qt's built-in drag-and-drop mechanism that enables the sending of drop events to the application. Every drop event holds MIME data that can be used for determining if the application should handle the event. In this case, the application accepts only those drop events that contain a web address (URL) or a path to a local file.

Image rendering within the *QtScrapBook* application is performed using the *QPainter* class and its *drawImage* method. This makes it possible to resize and scale images flexibly. When the user changes the size of the application, a paint event is sent to the application implicitly. The *drawImage* method is then invoked to (re)render the current image. Rendering is performed in the *Qt.KeepAspectRatio* mode so that the aspect ratio of the original image is always preserved. Furthermore, we utilize bilinear filtering (*Qt.SmoothTransformation* mode) to ensure smooth resizing of graphics.

4.5 QtComics: Comic Strip Viewer

QtComics application collects and displays comic strips from all over the world based on RSS feeds published on the Web. When the application is started, the user can select from a set of feeds that contain multiple comic strips. The selection is performed using a popup list (a *QComboBox* object) that lists the available feeds. To conserve screen space, the application displays only one strip at a time, as shown in Figure 5. The comic shown in this figure is from XKCD (http://xkcd.com/); reprinted with permission.

Fig. 5. *QtComics* application running on Nokia N810

At the implementation level, the *QtComics* application uses the *QNetworkAccessManager* and *QtNetworkRequest* classes to download the RSS feed asynchronously. The feed is parsed with a *QXmlStreamReader* object. A typical comic strip RSS feed item contains an HTML formatted string. The HTML code found inside the RSS feed item elements is stored into an array. The first array element is then shown inside *QWebView* web browser component. The *QWebView* object downloads content defined in the HTML code asynchronously and displays it on the screen. Thanks to the maturity of the Qt APIs used for accomplishing all this, the source code of the *QtComics* application is very short, only about 180 lines of JavaScript code.

5 Experiences and Discussion

In addition to the mashups described in the previous section we have developed a number of other mashups and web applications. These applications range from various map-based mashups to sports news tracking, weather forecast applications, media players and games. For instance, one of the applications is a mobile audio player that automatically collects artist information and other related information from different web sites. In the Web era, most of such information is available on the Web, albeit not necessarily in an easily digestible form.

All the applications have been written in JavaScript, utilizing the web browser, the JavaScript engine and the rich APIs provided by the Qt platform. While writing those applications, we have gained a lot of experience that is summarized in this section. We start from general observations related to mashup development, and proceed to comments that are specific to mashups on mobile devices. Finally, we summarize our experiences in developing mashups programmatically using Qt.

5.1 General Experiences and Comments

As we have already discussed earlier [13, 14], the majority of problems in web application and mashup development can be traced back to the fact that the Web was not originally designed to be a platform for active content and applications. The transition

from static web pages towards web applications is something that has occurred relatively recently, and the Web has not yet adapted fully to this transition. The problems in areas such as usability, compatibility and security are apparent when attempting to build web applications that run in a standard web browser.

From the viewpoint of mashup development, the two main problem areas are the lack of well-defined interfaces and insufficient security mechanisms. These two areas are discussed below.

Lack of well-defined interfaces. A key problem in mashup development today is the lack of well-defined interfaces that would describe the available web services in a standardized fashion. Although a number of web interface description languages exist, such as the Web Services Description Language (WSDL) [16] or the Web Application Description Language (WADL) [7], these languages are not yet in widespread use.

In general, only a fraction of the data, code and other content on the Web is available in a form that would make the content safely reusable in other contexts. Most web sites do not offer any public interface specification that would clearly state which parts of the site and its services are intended to be used externally by third parties, and which parts are implementation-specific and subject to change. In the absence of a clean separation between the specification and implementation of web sites, there are few guarantees that the reused services would remain consistent or even available in the future. This makes mashup development error-prone and the resulting mashups very brittle.

During the development of the mashups described in this paper, we found that only a small number of services, such as Google Maps and Flickr, offer a well-defined API through which these services can be used programmatically. In many cases, we had to parse HTML pages manually to scoop up the desired data from the web page. If there are subsequent changes in the format of the HTML page, the mashup that parses the page may suddenly stop working properly. This happened to us a few times, e.g., when developing the *QtComics* application.

Security-related issues. Another important problem in the creation of mashware is the absence of a fine-grained security model. The security model of the web browser is based on the *Same Origin Policy* introduced by Netscape back in 1996. The philosophy behind the same origin policy is simple: it is not safe to trust content loaded from arbitrary web sites. When a document containing a script is downloaded from a certain web site, the script is allowed to access resources only from the same web site ("origin") but not from other sites.

The same origin policy makes it difficult to build and deploy mashups or other web applications that combine content from multiple web sites. Since the web browser (the client) cannot easily access data from multiple origins, the mashing up of content must generally be performed on the server. Special proxy arrangements are usually needed on the server side to allow networking requests to be passed on to external sites.

The security problems of the Web present themselves in many other ways. Since there is no namespace isolation in the JavaScript engine, code and content downloaded from different web sites can interfere with each other. For instance, overlapping variable or function names in code downloaded from different sites will

almost surely result in errors that are very difficult to detect. Vulnerabilities based on this characteristics – collectively known as *cross-site scripting* (XSS) issues – have been exploited to craft phishing attacks and other browser security exploits. The possibility of such vulnerabilities is the reason why the same origin policy restrictions were originally introduced.

In the mashup development work described in this paper, we managed to bypass the limitations of the same origin policy by using Qt's networking primitives which do not adhere to the same origin policy. However, the namespace problems could not be avoided, and in a few situations overlapping variable declarations causes us considerable debugging headache, in spite of the relatively advanced debugging capabilities offered by Qt.

The key observation arising from all these problems is that there is a need for a *more fine-grained security model* for web applications. Until a more fine-grained security model and proper namespace isolation are available, mashup development is unnecessarily tedious and unsafe.

5.2 Comments Related to Mobile Mashups

Mashup development for mobile devices is still a new area. In principle, there should be little difference between mashups developed for mobile devices and the general Web. Ideally, as described in the *Mobile Web Best Practices* document of the World Wide Web Consortium [11], there should be just "One Web", meaning that the same information and services should be available to users irrespective of the device they are using.

In practice, One Web is still a dream, although we believe that over time most of the issues will be resolved [9]. In this subsection we discuss the main issues today, focusing on usability, connectivity and performance issues.

Usability issues. The mashups that we developed were not written only for mobile use. Rather, we intended them to be practical on desktop computers as well. However, since our target mobile device (Nokia N810) is stylus-operated and has a significantly smaller, 800x480 pixel screen than a typical desktop computer, usability problems could not be avoided. For instance, many of the Qt widgets used in our mashups are so large that they used excessive amounts of precious screen space. Initially, some widgets ended up being outside the viewable area. Font size differences gave us some problems, too. Fonts that look nice on desktop computers are not necessarily readable on the small screens of mobile devices.

Since our target device had a stylus, applications that require the precise use of a pointing device (e.g., the *QtWeatherCameras* application in which the user has to choose precise points on a map) are quite easy to use even on a small screen. However, we suspect that on other types of mobile devices, such as on conventional “candybar” mobile phones with only a numeric keypad, the use of such applications could be challenging.

Connectivity issues. The availability of a reliable Internet connection is vital for mashups. In mobile devices the network connection can often be slow, unreliable or unavailable altogether. The application developer should take this into account in the design of the applications, and provide feedback to the user when problems do occur.

In our mashup development work with Qt, sporadic connection blackouts did not usually pose major problems. Since our mashups run mainly on the client, the applications remain active if the network connection goes down. When using the Qt networking classes, the network requests will remain active until they are successfully completed, or they will timeout eventually if something goes wrong.

Performance issues. One of the main factors separating mobile devices from desktop computers is performance. Not only are mobile devices considerably slower than their desktop counterparts, but they usually have significantly less memory and storage capacity as well. Although processor and memory limitations will decrease over time, performance issues still cannot be ignored in developing mobile applications today. In the development of the mashups described in this paper, performance differences played a significant factor. Mashups such as *QtFlickr* and *QtWeatherCameras* require a lot of computation, and they are very slow on our target device. The performance problems are caused partially by the slow JavaScript engine used by Qt. In the last year or so, several high-performance JavaScript engines such as Apple's *SquirrelFish Extreme* (http://webkit.org/blog/214/) and Google's *V8* (http://code.google.com/p/v8/) were released, with performance improvements of more than an order of magnitude over conventional JavaScript interpreters. Once such engines become widely available, application response and load times should improve considerably.

5.3 Comments Related to Qt

One of the characteristic features of our mashups and the Lively for Qt system is that the majority of software development is performed programmatically, using *imperative* development style familiar from desktop software development. This is in contrast with traditional web technologies, which rely heavily on *declarative* languages such as HTML and CSS. In this respect, our applications bear close resemblance to applications developed with Rich Internet Application (RIA) platforms such as Adobe AIR [15] or Microsoft Silverlight [12].

Given that Qt APIs have been in development and use since the early 1990s, the Qt APIs are on par with the best RIA systems today. For instance, the expressive power of Qt APIs such as the *QXmlStreamReader* class saved us a lot of work when parsing complex XML data. Furthermore, API documentation and the available development and debugging tools for Qt are in good shape. In general, it was very easy to get started with Qt.

More generally, the combination of an existing, mature application framework with a built-in web browser and popular, fully dynamic programming language (JavaScript) turned out to be a powerful combination. With a JavaScript engine and only a few thousand lines of JavaScript code, it is possible to turn an existing, mostly static, binary, desktop-era application framework into a highly interactive, dynamic web development environment supporting mobile mashup development. Applications require no compilation, binaries or explicit installation, and yet they can utilize the full power of the existing, mature application framework.

On the negative side, although Qt is intended to guarantee platform-independence, we did experience some portability issues. For instance, some widgets or fonts refused to render themselves correctly on some target platforms. Apart from rendering

errors, event-handling differences and some performance-related issues on mobile devices, no other major issues were encountered, though.

The use of the JavaScript language for developing real applications is still a relatively new topic. When JavaScript is used as a programming language for developing full-fledged applications – as opposed to the conventional use of JavaScript as a *scripting* language – one has to be aware of its caveats and peculiarities. These topics have been summarized well by Crockford [3].

6 Conclusions and Future Work

In this paper we have presented an overview of the mobile web mashups that we have implemented in JavaScript on top of Qt – a cross-platform application framework recently acquired by Nokia. This work is part of a larger activity called *Lively for Qt*, an effort that has created a highly interactive, mobile web application and mashup development environment on top of the Qt framework. Here, we summarized our experiences in developing these applications, and included some source code to illustrate the development style.

Plenty of interesting avenues remain for future work. We are especially excited about the possibility of building *location-aware mashups* that have been customized to take into account the user's current location, utilizing the GPS satellite position information available in modern mobile devices. With increasingly reliable and inexpensive network connections, *collaborative mashups* that allow real-time collaboration between multiple mobile users are also becoming a reality. In general, the use of mashups and web applications in mobile devices offers entirely new possibilities that are beyond the reach of web services on desktop computers. Although various obstacles still remain, we are inspired by these possibilities and hope that this paper, for its part, encourages people to continue the work in this exciting new area.

Acknowledgments

This research has been supported by the Academy of Finland (grant 115485).

References

1. Clarke, J., Connors, J., Bruno, E.: JavaFX: Developing Rich Internet Applications. Java Series. Prentice Hall, Englewood Cliffs (2009)
2. Crane, D., Pascarello, E., James, D.: Ajax in Action. Manning Publications (2005)
3. Crockford, D.: JavaScript: The Good Parts. O'Reilly Media, Sebastopol (2008)
4. ECMA Standard 262: ECMAScript Language Specification, 3rd edn. (December 1999), `http://www.ecma-international.org/publications/standards/Ecma-262.htm`
5. Gibson, R., Erle, S.: Google Maps Hacks. O'Reilly Media, Sebastopol (2006)
6. Goodman, D.: Dynamic HTML: The Definitive Reference. O'Reilly Media, Sebastopol (2006)

7. Hadley, M.: Web Application Description Language Specification (November 9, 2006), https://wadl.dev.java.net/
8. Hanson, R., Tacy, A.: GWT in Action: Easy Ajax with Google Web Toolkit. Manning Publications (2007)
9. Mikkonen, T., Taivalsaari, A.: Creating a Mobile Web Application Platform: The Lively Kernel Experiences. In: Proceedings of the 24th ACM Symposium on Applied Computing, SAC 2009, Honolulu, Hawaii, March 8-12, pp. 177–184 (2009)
10. Mikkonen, T., Taivalsaari, A., Terho, M.: Lively for Qt: A Platform for Mobile Web Applications. In: The Proceedings of the Sixth ACM Mobility Conference, Mobility 2009, Nice, France, September 2-4 (2009) (to appear)
11. Mobile Web Best Practices 1.0. World Wide Web Consortium Recommendation Document (July 29, 2008), http://www.w3.org/TR/mobile-bp/
12. Moroney, L.: Introducing Microsoft Silverlight 2.0, 2nd edn. Microsoft Press (2008)
13. Taivalsaari, A.: Mashware: The Future of Web Applications. Sun Labs Technical Report TR-2009-181 (February 2009)
14. Taivalsaari, A., Mikkonen, T.: Mashups and Modularity: Towards Secure and Reusable Web Applications. In: Proceedings of First Workshop on Social Software Engineering and Applications, SoSEA 2008, L'Aquila, Italy, September 16 (2008)
15. Tucker, D., Casario, M., De Weggheleire, K., Tretola, K.: Adobe AIR 1.5 Cookbook. O'Reilly Media, Sebastopol (2008)
16. Web Services Description Language. World Wide Web Consortium (W3C) Specification (March 15, 2001), http://www.w3.org/TR/wsdl

Publication V

Towards pervasive mashups in embedded devices: Comparing procedural and declarative approach

A. Salminen and T. Mikkonen

Towards pervasive mashups in embedded devices: comparing procedural and declarative approach

Arto Salminen* and Tommi Mikkonen

Department of Software Systems,
Tampere University of Technology,
P.O. Box 553, FIN-33101 Tampere, Finland
Fax: +358 (0) 3-3115-2913
E-mail: arto.salminen@tut.fi
E-mail: tommi.mikkonen@tut.fi
*Corresponding author

Abstract: The web has become pervasive. This has led to a paradigm shift, where applications live on the web as services, where they can be accessed with different types of terminals. The ability to dynamically combine content from numerous sources, and the ability to instantly publish services worldwide has opened up entirely new possibilities for software development. Such applications that aggregate content from the web are commonly referred to as mashups. Unfortunately, for various reasons, the browser is inadequate for hosting complex mashups, in particular when considering embedded devices and subsystems that are not readily available in the web. In this paper, we introduce two environments, intended for hosting context-aware mashups on embedded devices. These environments have different approaches as one can be used to compose mashups in procedural and another in declarative fashion. As an example, we describe a location-aware mashup composed for both environments.

Keywords: mobile runtime; mashup; runtime environment; embedded devices; pervasive web; procedural approach; declarative approach; web applications; context-aware mashups; device peripherals; location-aware.

Reference to this paper should be made as follows: Salminen, A. and Mikkonen, T. (xxxx) 'Towards pervasive mashups in embedded devices: comparing procedural and declarative approach', *Int. J. Communication Networks and Distributed Systems*, Vol. X, No. Y, pp.000–000.

Biographical notes: Arto Salminen is a PhD student at Tampere University of Technology. Currently, he holds a position as a Researcher at Tampere University of Technology as part of a research group studying web applications. His research interests include web applications, especially mashups – applications that combine content from multiple sources over the web. His most recent efforts include studying dynamic 3D web applications and research about composing dynamic user interfaces for binary software.

Tommi Mikkonen is a Professor of Computer Science at Tampere University of Technology, Finland, pioneered mobile software development related education and research in Finland. Over the years, he has arranged numerous courses on software engineering, mobile computing, embedded systems, and operating systems. His present research interests include software engineering, cloud computing, web programming, embedded systems, and mashup development.

1 Introduction

The web has become pervasive. This has led to a paradigm shift, where applications live on the web as services. Moreover, the devices that can be used for accessing services can be many, including – in addition to regular computers – many kinds of embedded devices, such as mobile phones, game consoles, and so forth. We believe that this is only the beginning of a new era, where the web is a ubiquitous distribution channel for data, code and other content.

An important realisation is that applications built on top of the web do not have to live by the same constraints that have characterised the evolution of conventional desktop software. The ability to dynamically combine content from numerous websites and local resources, and the ability to instantly publish services worldwide has opened up entirely new possibilities for software development. In general, such systems are referred to as mashups, which are content aggregates that leverage the power of the web to support instant, worldwide sharing of content.

In this paper, which is a revised and extended version of article by Mikkonen and Salminen (2010), we introduce two ways to compose mashups that can be specific to the used embedded device and application. Moreover, we show that by combining data that is available locally with information available in the web, increasingly complex applications can be built with relative ease. The first approach used for composing such mashups is similar to that of Nyrhinen et al. (2009), but now we use the Qt to enable composition of mashups with the ability to use (and share) local data and device peripherals as well. The second approach is based on novel technology called Qt Quick, which allows declarative mashup composition on top of Qt APIs and enables combination of native data processing and dynamic user interface.

As a practical implementation, we use a simple system where GPS data available in mobile devices is combined with normal web data. The approach is not however restricted to this particular case, but can be generalised for different combinations of technologies and devices, provided that adequate communication means are available. The basis of our implementation is Lively for Qt (http://lively.cs.tut.fi/qt), a JavaScript-based system we have used for experimenting the use of web applications in mobile devices.

The rest of this paper is structured as follows. In Section 2, we provide motivation for our work in the realm of embedded systems, and introduce a number of other systems where composing context-aware mashups is possible. Then, in Section 3 we discuss how we have built our runtime environment. In Section 4, we introduce two sample applications that mash up content from the web and from the physical environment of a mobile device using its GPS sensors. Here, we give an insight to the implementation of the applications at the level of programme code. In addition, a discussion is given how the system could be generalised to arbitrary embedded devices or even distributed systems, such as sensor networks. In Section 5, we provide a discussion on lessons learned and some future possibilities, and in Section 6, we draw some final conclusions.

2 Background

In this section, we provide a discussion on our motivation, and quickly review some pieces of existing research in the field of composing mashups with data in the internet and local resources.

2.1 Motivation

An increasing number of embedded devices – mobile phones being the forerunners in this field for obvious reasons – provide wireless connectivity. Indeed, it is not uncommon for televisions, game consoles, smart home devices, and other systems to offer internet access or accessibility from the internet.

For the most part, connectivity in the present embedded devices and systems is commonly used for delivering upgrades to the device when, e.g., new software version has been introduced, and for playing collaborative applications, such as multi-player games. Moreover, one can use connectivity in a fashion where the device offers either a web browser or a web server. The former enables one to use the device like a regular computer, whereas the latter can be used for accessing the embedded device and its internal data.

These two different forms of usage – binary software on the one hand, and web-oriented, somewhat restricted infrastructure on the other hand – are by no means the only options. Instead, offering an interface for mixing web technologies with the capabilities of embedded devices can be used as a platform for composing complex applications that combine the best of both worlds: performance and eye candy of traditional, installed binary applications and pervasiveness and seemingly infinite resources of the web.

Although mashups are usually run inside a web browser, this is not an essential requirement. In fact, when considering embedded devices and resources in external systems, browser's security mechanisms become a burden, since they do not allow accessing arbitrary addresses from programme code. Furthermore, when running mashups in embedded devices the browser may be overkill for performance. Finally, when considering the situation in terms of technology, mashups are a fragmented mixture of different kinds of technologies like AJAX, HTML, CSS, DOM and JSP (Paulson, 2005), with some JavaScript (Flanagan, 2006) code to enable client-side executions. This complex combination of technologies then makes it more difficult to create maintainable applications (Mikkonen and Taivalsaari, 2008), which in turn is a necessity for introducing them in embedded devices.

Most of the above issues can be tackled using a special-purpose runtime environment, as already shown by Mikkonen et al. (2009). However, in the field of combining data not downloaded directly from the web information systems but generated by different devices, sensors and actuators, numerous challenges remain. The scripts that on the one hand communicate with the web should on the other hand be able to access scarce resources in an embedded device.

From the implementation perspective, such an approach requires a liberal web runtime environment that enables accessing the web with JavaScript, the lingua franca of the web, and provides an access to the facilities of the device in an unrestricted fashion. Provided with such a runtime environment, the development of mashup applications that

combine content from the web with context-aware data available from the device can be considerably simplified.

Finally, while in general only the device manufacturer has been providing applications for embedded devices, at times it could be beneficial to allow the execution of 3rd party applications in embedded devices. Provided with scripting facilities and reasonable access to devices' capabilities, new innovative application development ecosystem may emerge. As an example, the development of games using mobile Java has grown into a considerable global business in a relatively short amount of time.

2.2 Related work

Developing mobile, context-aware mashups has gained more attention as more advanced tools and platforms have been provided. In this section, we briefly present some of the research and solutions related to our work.

As stated by Maximilien (2008), the ability to access device data, especially accurate location, is important for numerous use cases of mobile mashups. Information related to user context, community and authentication can be collected and provided to web services by a middleware component, as described by Koskela et al. (2007) and López-de-Ipiña et al. (2007). Another approach is to use a modified web browser. For example, TELAR Mashup platform utilises Mozilla browser with two extensions – the GPS access module and the delivery context client interface module – to act as the interface for providing location data to web pages (Brodt et al., 2008).

Mobile device manufacturers, prime examples being Nokia and Apple, have their own platform specific solutions for writing web applications that have access to device peripherals. Nokia's web runtime is an extension for the Symbian web browser to enable widgets, which in essence are small web applications that can use JavaScript APIs for access to user and device information, for example, GPS data. Apple's iPhone SDK can be used to create web applications written in Objective-C. It includes UIWebView class for implementing web browser functionality inside native applications. Furthermore, the mobile version of the Safari browser supports W3C Geolocation API that enables client side JavaScript applications to gain location information. While the proposed HTML5 standard includes Geolocation API – that can be used to obtain WLAN or GPS-based location information – there still is no way to use it to access other device peripherals such as light sensor or accelerometer. Described solutions can only be used with proprietary devices while method to develop mashups for an arbitrary embedded device is desired.

Number of frameworks such as Nitobi's PhoneGap, Appcelerator's Titanium and RhoMobile's Rhodes can be used to create cross-platform web applications for major mobile operating systems. PhoneGap and Titanium are used with combination of HTML, CSS and JavaScript and Rhodes is programmed in Ruby. These frameworks enable access to the mobile device peripherals as well as to web resources but cannot be used outside the mobile operating systems in arbitrary devices. All these solutions have other restrictions too. User interface in PhoneGap and Rhodes is created as a web application which is why they rely heavily on the web browser component of the underlying deployment environment. As we discussed earlier, this approach can be overkill for performance and lead to hard to maintain applications. Titanium utilises native user interface controls, but has some platform specific APIs that reduce the cross-platform

capability of the result application. Furthermore, extending Titanium or Rhodes application with native code is cumbersome.

3 Building a runtime environment

In combination with the Lively for Qt, we have introduced a view to composing compelling mashups for mobile devices (Mikkonen et al., 2009). A part of the solution is to introduce a runtime system that is capable of accessing native routines, which leads to improved performance compared to the web browser. In addition, the system must be flexible enough for liberating applications from restrictions of the browser. In the following sections, we introduce the rationale we have had in our mind when developing a runtime that satisfies the above requirements, and how it can be realised.

3.1 Towards a liberal web runtime

Originating from mobility domain, the steps needed for implementing our original vision of the mobile web applications were originally listed in Mikkonen et al. (2009). They are also listed in the following:

1 Take a rich graphics framework that supports a rich suite of widgets and provides programmatic support for direct manipulation and flexible graphical transformations.

2 Take JavaScript, the world's most widely used dynamic language, and make the graphics APIs available to it.

3 Make it easy to access the web from the platform using asynchronous HTTP networking and other commonly used networking standards; provide JavaScript APIs for processing XML, JavaScript object notation (JSON), DOM and other frequently used formats and structures.

4 Add support for device-specific JavaScript APIs for areas such as wireless messaging, location, Bluetooth and camera support.

5 Make the platform available both inside the web browser and natively. All the applications should be able to run both inside the web browser and as 'phonetop' applications that behave like native applications.

6 Add a fine-grained security model so that network-downloaded applications can be executed safely; provide minimal access to APIs for those applications that are downloaded from untrusted sources, and more extensive access to APIs for applications from trusted sources.

Mikkonen et al. (2009) introduced an implementation that satisfies a number of the above steps. The system was built on top of the Qt platform and its auxiliary facilities, but our actual applications are composed solely with scripts.

Qt (http://qt.nokia.com) is a mature, well-documented cross-platform application framework that has been under development since the early 1990s. Qt has been used in various embedded devices and applications, including mobile phones, PDAs, GPS receivers, handheld media players and automotive user interfaces.

The Qt platform offers an extensive set of APIs for various purposes. To begin with, facilities for developing a rich graphical user interface are included in the platform. Both low-level graphics and high-level widgets are included. In addition, Qt offers APIs for networking, including in particular Ajax-style asynchronous network accesses (Crane et al., 2005), file access, D-Bus communication and many other useful tasks. Finally, Qt libraries include a complete web browser based on the WebKit browser engine. The necessary DOM and XML APIs are also included to parse, manipulate and generate new web content.

While the vision originated from the realm of mobile devices, it can be easily extended, since the Qt platform is commonly used in other types of embedded systems, not just mobile devices. Consequently, potential range of gadgets that can benefit from the implementation of the vision increases considerably. In the following, we give an overview to our approach, followed by a discussion on how to add device specific interfacing capabilities needed in various embedded systems.

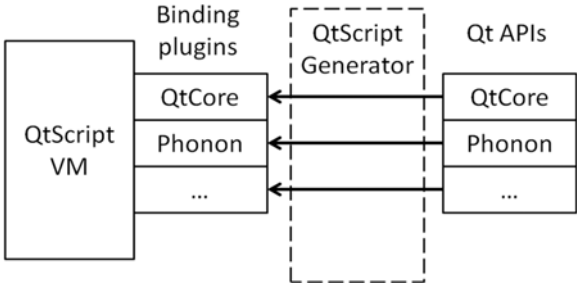
3.2 Mashup runtimes based on Qt

There are two approaches to create dynamic applications on top of Qt C++ APIs. The first method is creating bindings for JavaScript and constructing the application entirely in procedural fashion. The second method is having a declarative approach to define the application user interface and using only small snippets of JavaScript.

3.2.1 QtScript

The first approach is based on QtScript scripting engine – essentially a JavaScript virtual machine – that has been part of the Qt framework since version 4.3. Qt APIs, which have been originally written in C++, have been made visible to the JavaScript environment using an additional tool called QtScriptGenerator. The tool generates a plugin component for every Qt API, which must be loaded by the QtScript virtual machine before using the interface. An overview of this system is given in Figure 1.

Figure 1 QtScript runtime environment



So far, we have shown that this environment is well-suited for implementing compelling mashups that can be run in laptops and mobile devices (Nyrhinen et al., 2009). However, the implementation still requires further work. In the context of mashup development, mobility interfaces are far from complete, and additional interfaces to other devices are missing. In the following, we demonstrate how such interfaces can be added. An additional problem, which we will overlook in this paper, is the fact that the platform is

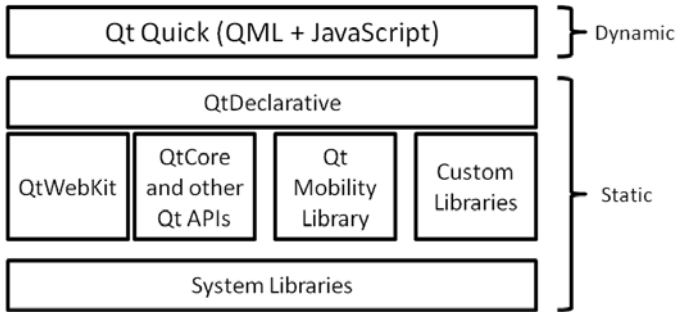
composed by simply generating bindings that give all JavaScript applications an arbitrary access to all Qt interfaces.

3.2.2 Qt Quick

The second approach is based on Qt Quick (see Figure 2), which is a collection of technologies that are designed for creating user interfaces on mobile phones, media players, set-top boxes and other portable devices. It introduces a set of user interface elements, declarative language (called QML) for describing the interface and a language runtime. QML markup resembles combination of HTML, CSS and JSON. The user interface is specified as a tree of QML elements acting as graphical and behavioural building blocks that can be combined together. These elements include basic shapes and text as well as more complex items such as a web browser component, element views and models. Transitions between different element combinations and states can be animated. Furthermore, QML is network transparent which allows loading a complete user interface over the web.

The user interface can consist of multiple declarative QML modules, each defining one Qt Quick element. These modules can be used similarly to widgets, and libraries providing user interface components, for instance, Colibri (Qt Quick COmponent LIBRary) and Qt Quick components, have emerged.

Figure 2 Qt Quick architecture



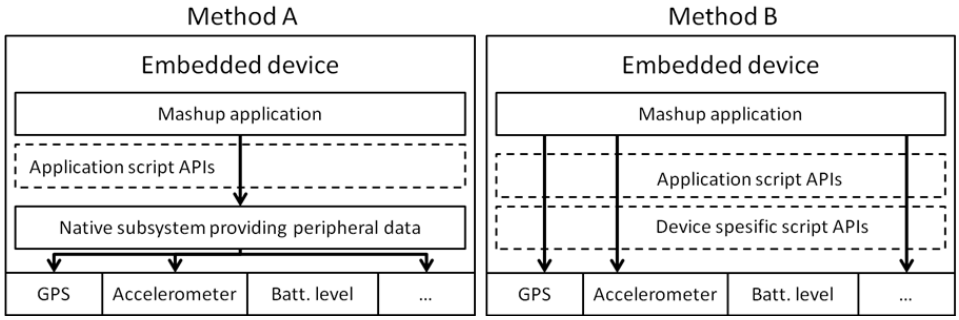
The application logic or libraries can be defined both in separate JavaScript files and native binaries that have an interface for QML. This possibility utilise both JavaScript and native code for application logic enables the programmer to flexibly use Qt C++ for high performance and security critical parts and JavaScript for those parts where dynamic properties are needed. Classes included in the QtDeclarative module are used to bind the dynamic Qt Quick to static Qt APIs and QtWebKit. Similar to QtScript, mobility interfaces are not available out of the box for the Qt Quick. However, as the Qt Quick can be expanded with native custom components, the access to device peripherals can be added. In this case, it is possible to restrict dynamic code accessing the APIs.

3.3 Adding device specific interfaces

In order to enrich the facilities offered for mashup development, our system can be enhanced with additional interfaces. There are two possible ways (A and B) to accomplish this as shown in Figure 3. The first way (A) is that an additional interface is

introduced using C++, and bindings to our runtime system are either generated automatically similarly to any other interface, or implemented manually. This however assumes that a programme-level interface is made available, which may be an overly optimistic assumption for numerous cases. The other way (B) is to use facilities that are already included in the platform, and introduce additional interfaces using JavaScript. This approach can be used if the other subsystem can be accessed with standard interfacing mechanisms, such as sockets for instance.

Figure 3 Alternative ways for a mashup to access embedded device peripherals



To support mashup development, an embedded device manufacturer could provide the necessary JavaScript interfaces thus making the former approach more compelling. This way the manufacturer could also define parts of the device that are safe for being accessed using JavaScript.

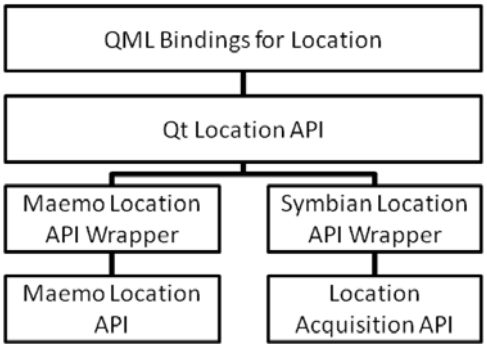
In our implementation, where we access the GPS subsystem of a mobile device, we have followed the former approach (A) in our Qt Quick-based application and the latter approach (B) in QtScript-based application. Since the socket interface is commonly used in various systems, it is possible to generalise our approach with QtScript for numerous other devices as well. The same applies for Qt Quick as it uses the native interfaces for accessing the device APIs. For obvious reasons, application-specific parts require further attention whenever a new interface is introduced.

- *Accessing the GPS peripheral data with QtScript*. The QtScript-based system was originally designed for devices that include a GPS module, but it was later improved to support other mechanisms for accessing GPS data as well, assuming that an IP address to the GPS module can be provided to the application. Now we have used the latter approach (B). At the level of implementation, we use sockets for the communication between the JavaScript application and the GPS module. Since sockets can either be local to the device or access an external device, which simplifies our implementation. Inside the application, we still must parse the information provided by the GPS module, which typically is made available in the NMEA data format, into a format that is usable for the application.
- *Accessing the GPS peripheral data with Qt Quick*. Qt module called Qt Mobility (http://doc.qt.nokia.com/qtmobility/) can be used to generate both C++ and Qt Quick bindings to device's peripherals. Qt Mobility includes APIs for working with mobile devices, for example, bearer management, contacts, location, messaging, multimedia, service frameworks and system information. Some of the Qt Mobility features, such as video and audio support in the multimedia API, can be used on desktop as well. In

fact, portability has been an important design target for Qt Mobility, and it already supports Maemo, Meego and Symbian targets as well as Linux, Mac OS X and Windows.

The Qt Quick-based system is designed for touch screen Maemo, Meego and Symbian devices where the latest Qt with support for Qt Quick and Qt Mobility is available. In this case, we have followed the former approach, as the QML bindings for positioning are available thanks to Qt Mobility. The communication with the GPS hardware is based on device specific positioning APIs. The API stack used to access the GPS peripheral on Maemo and Symbian is shown in Figure 4.

Figure 4 Qt mobility location API stacks for Maemo and Symbian platforms



4 Sample applications

As an example, we introduce next multi-user location mashup, which is a classic example of a collaborative application that utilises location database, into which data is collected from GPS devices, and map component that is used as the background on top of which GPS devices are shown. The mashup can be used to present user locations and their status messages over the map. Nicknames are used to identify different users.

The database used by the mashup is Persevere (http://www.persvr.org/), an open source JSON database and JavaScript application server. Persevere is very convenient when using JavaScript, as the data is always accessed through JSON HTTP/ReST calls. In our implementation, each client updates its location in the database. Moreover, in order to get data regarding other users, each client will also download locations of other clients.

4.1 Qt Script implementation

The QtScript-based mashup was developed as a Qt widget, similarly to the widgets described in Mikkonen et al. (2009) and Nyrhinen et al. (2009). The user interface of the system running in Nokia N810 Maemo environment is illustrated in Figure 5.

Since some of the mobile devices that run the application may not include a GPS receiver, a simple fallback mechanism was introduced to ensure user positioning. At first, the application tries to make connection to GPS hardware interface. The GPS position is the most accurate way for this application to define the location. If the GPS interface is

not available, a dialog is opened for the user to decide either to try GPS positioning again or to attempt positioning based on IP address location. If the location service is down, or the IP address is otherwise out of reach, the user is asked to provide his or hers location by hand.

Figure 5 Multi user location mashup executed in Nokia N810 device (see online version for colours)

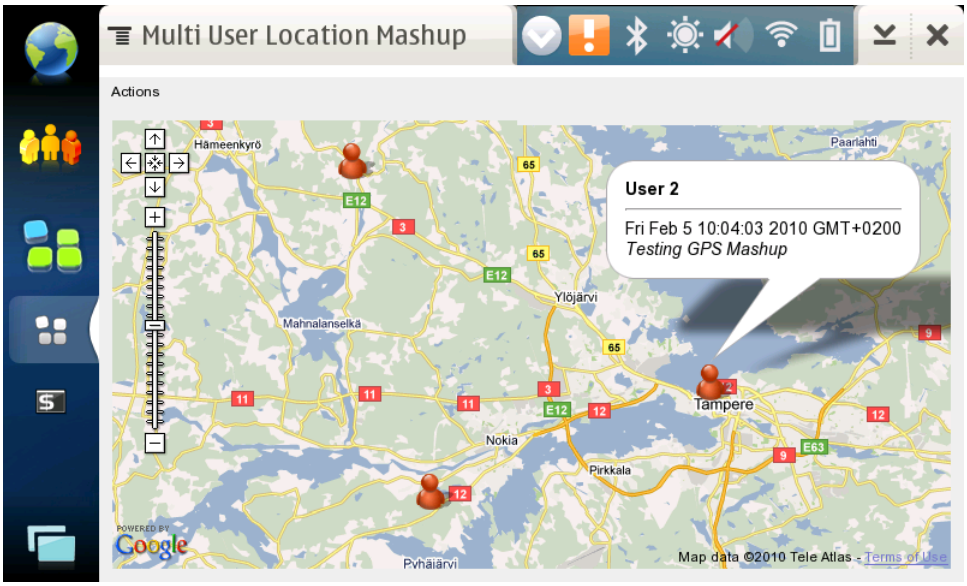
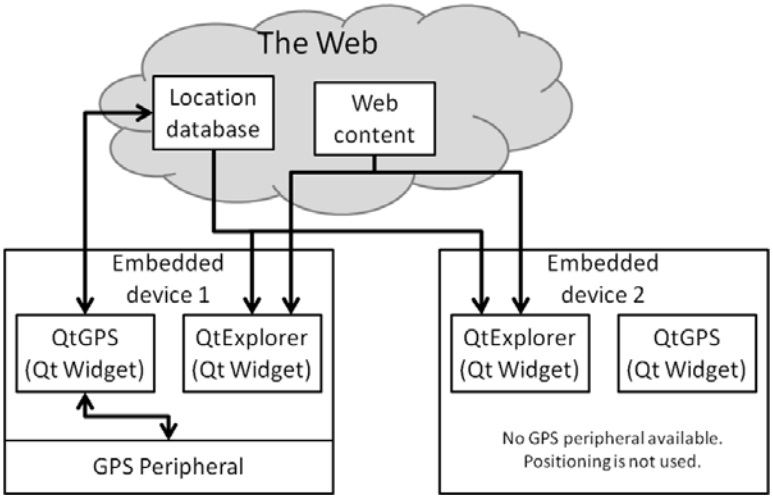


Figure 6 Sample application implementation



The implementation (see Figure 6) we have composed is two-fold. On the one hand, we have implemented GPS peripheral access routines using JavaScript. On the other hand, we have composed a user interface that enables displaying the information on top of a

map downloaded from the web, which in turn provides context for the location. These parts are implemented as JavaScript classes; the latter called `QtGPS` and the former `QtExplorer`. In the following, these two subsystems are discussed.

4.1.1 Accessing GPS data from JavaScript

Socket connection implementation is very straightforward in our system. The Qt framework readily includes a class called `QTcpSocket` for TCP socket connections, which communicates with the GPS daemon running in the background. The state of the mashup is maintained according to signals that class `QTcpSocket` emits using Qt's signals and slots mechanism used for managing callbacks.

Listing 1 is a code fragment where a new `QTcpSocket` is created and its connections are defined and initialised.

Listing 1 Constructing a new QTcpSocket object and defining its connections

```
this.socket = new QTcpSocket(this);
this.socket.readyRead.connect(
  this,
  this.readSocket
);

this.socket.stateChanged.connect(
  this,
  this.stateChanged
);

this.socket.error.connect(
  this,
  this.socketError
);
```

When a user puts positioning into operation, the function shown in Listing 2 is executed. At first, Qt socket is utilised to connect GPS socket. In the case of connection timeout, fallback mechanism is initiated by a separate function `askAboutLocationSimulation`. The user can then select to try connecting the GPS daemon again, use another strategy to determine the position or decide not to use positioning at all.

It is necessary for the application to have up to date information about GPS socket connection state. For example, if GPS daemon crashes, accurate location information is not available anymore and user actions may be required to start the daemon again or to make decision about fallback mechanism.

Listing 3 shows functions that handle signals that the GPS socket connection class may emit. Effectively, both functions only inform other parts of the programme about problem in GPS positioning.

Listing 2 Connecting to the GPS socket

```
/* Connect to the GPS Socket */
QtGPS.prototype.connect = function() {
  /* If a nickname is not set, ask for it */
  if(this.name == undefined) {
    this.name = this.getNickname();

    var host = new QHostAddress(this.GPS_HOST);

    this.socket.connectToHost(host, this.GPS_PORT);

    if(this.socket.waitForConnected() == false){
      this.isConnected = false;
      this.posTimer.stop();
      /* Initiate fallback mechanism */
      this.askAboutLocationSimulation();
    }
    else {
      /* Successful connection to GPS Socket */
      this.isConnected = true;
    }
    /* Start timer for user positioning */
    this.posTimer.start(this.POS_TIME);
  }
};
```

Listing 3 GPS socket state handling functions

```
/* Handle GPS Socket state changes */
QtGPS.prototype.stateChanged = function(state) {
  if(this.socket.state() ===
       QAbstractSocket.UnconnectedState) {
    this.isConnected = false;
    this.hasFix = false;
  }
};

/* Handle GPS Socket errors */
QtGPS.prototype.socketError = function(error)
{
  this.error = true;
};
```

GPS daemon does not write new NMEA data to the socket if it is not requested to do so. Querying of GPS information and reading it from the socket at the level of code is presented in Listing 4. The first function called `updatePosition` writes a request to the socket if connection is up. When new NMEA data is available another function, `readSocket` is executed to read and parse the data available. Finally, updated user position information is written to Persevere database.

Listing 4 Accessing GPS information

```
/* Request NMEA data from the GPS Socket */
QtGPS.prototype.updatePosition = function(state)
{
   if(this.socket.state() ===
        QAbstractSocket.ConnectedState) {
     this.writeSocket(
       new QByteArray(this.NMEA_REQUEST)
     );
   }
};

/* Write data to the GPS Socket */
QtGPS.prototype.writeSocket = function(data) {
   if(data && this.socket.state() ===
        QAbstractSocket.ConnectedState) {
     this.socket.write(data);
   }
};

/* Read data from GPS Socket, process it
   and upload it to the database */
QtGPS.prototype.readSocket = function() {
   var str = this.socket.readAll().toString();
   this.NMEAParser(str);
   this.updatePersevere();
};
```

4.1.2 Mashing up GPS data and other content

To combine user location data with other content we have implemented a class called `QtExplorer`. This part of the application defines the user interface, accesses the database and presents the most recent data on Google Maps component.

User interface of the application consists of a map component and a drop down menu as seen in Figure 5. The map is used to visualise database contents, as it presents different

users on their locations. The drop down menu, which opens when the application header is clicked, holds items for setting user status, nickname and visibility of user location. User can also toggle automatic position updates from the database and fit the map viewport to show all markers.

In the constructor of the `QtGPSExplorer` class a new instance of `QtGPS` object is created to produce position information to the database. Furthermore, a `QTimer` is utilised to request data from the database in regular intervals. When new data is obtained, it is parsed and presented over the map component. Listing 5 presents functions that make asynchronous request to user status database and handle the response. The former utilises Qt class `QNetworkAccessManager` to make a new network request and the latter reads the response and parses it to JavaScript object that holds information about user locations and statuses. Separate function `updateMap` is used to present fresh data over the map component.

Saving user status and location to the database is done with asynchronous network request similarly to what is shown in Listing 5. First the data to be saved is formatted into JSON. Second the data is send to the database with a POST request. Finally, the response from the database needs to be handled to make sure that the operation was successful.

Only data more recent than two days is displayed. A marker is placed over the map to illustrate a user on his location. When the marker is clicked, user information shows up in a balloon created with Google Maps API as can be seen in Figure 5.

Listing 5 Requesting new data from database

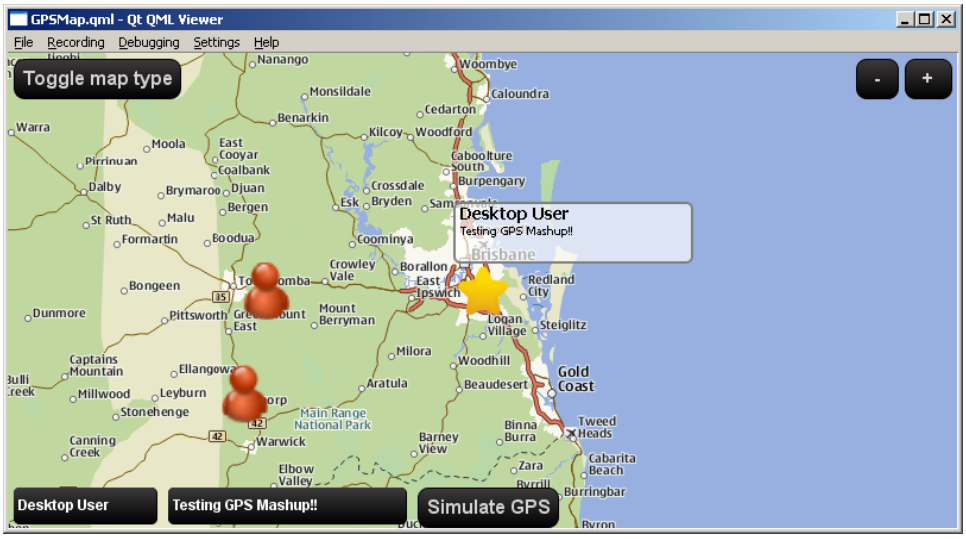
```
/* Request data from database */
QtExplorer.prototype.requestData = function(){
   var am = new QNetworkAccessManager(this);
   am.finished.connect(this, this.handleReply);
   am.get(
     new QNetworkRequest(new QUrl(this.URL))
   );
};

/* Handle database response */
QtExplorer.prototype.handleReply = function(reply) {
   var JSONStr = reply.readAll().toString();
   var JSONObj = parse(JSONStr);

   /* Update the Google Maps component */
   this.updateMap(JSONObj);
};
```

As the application is divided logically in to two different parts – one that produces information to the database and another that consumes it – it is easy to build up more complex mashups by re-implementing the `QtGPSExplorer` class.

4.2 Qt Quick implementation

The Qt Quick implementation is build with Qt Mobility QML bindings. Besides Qt Mobility, other native libraries have not been used. We have utilised JavaScript libraries for JSON parsing and accessing the database. User interface elements are implemented with Colibri widget library written in QML. The application executed on a desktop environment can be seen on Figure 7.

Figure 7 Qt Quick-based GPS mashup executed on desktop environment (see online version for colours)



The Qt Mobility 1.1 location module is included into the main QML file with import `QtMobility.location 1.1` statement. This module introduces new elements, such as `Map` – a component that creates a world map that can be panned and zoomed. Listing 6 shows definition of the Map element used in our mashup. The map size is defined with QML anchor layout to be full screen. For graphics, the map uses a separate `Plugin` element, in this case the graphics are provided by Nokia. The map centre is defined with a `Coordinate` element with default latitude and longitude. Finally, we attach a function to centre and zoom level change events to make the elements representing users be repositioned accordingly.

It was necessary to create a separate custom QML element called `User` to represent different mashup users on their locations. Listing 7 shows how the `User` element that marks the position of the current mashup user himself is defined. This element is not visible until the system obtains valid position information.

We have utilised Colibri QML component library for user interface widgets such as buttons and line edits. For example, Listing 8 shows how the button intended for simulating the GPS data is created. The button position related to other elements is defined with anchor based layout. Appearance of the button can be changed by editing different parameters, such as `color`, `gradientDefaultOn` and `textColor`. We have attached an inline function to handle button clicks. The function sets the

`PositionSource` element called `myPositionSource` to obtain data from an external file and activates it.

Listing 6 Definition of the Map element

```
Map {
   id: map;
   size.width: parent.width;
   size.height: parent.height;
   anchors.fill: parent;
   plugin : Plugin {
          name : "nokia";
         }
   zoomLevel: 7;
   mapType: Map.SatelliteMapDay;
   center: Coordinate {
          latitude: defaultLatitude;
          longitude: defaultLongitude;
         }
   onCenterChanged: {
     setUsersOnMap();
   }
   onZoomLevelChanged: {
     setUsersOnMap();
   }
}
```

Listing 7 Definition of the user element that represents the mashup user

```
User{
   id: myself;
   icon: "./qml/star.png";
   status: myStatus;
   name: myName;
   lat: myLat;
   lng: myLng;
   visible: false;
}
```

Listing 8 Button created with QML element library called Colibri

```
CLButton {
   id: simulateButton;
   anchors.left: statusLineEdit.right;
```

```
    anchors.leftMargin: 10;
    anchors.top: statusLineEdit.top;
    color: "darkgrey";
    gradientDefaultOn: true;
    textColor: "lightgrey";
    text: "Simulate GPS";
    onClicked: {
      myPositionSource.nmeaSource = "qml/nmealog.txt";
      myPositionSource.start();
    }
}
```

4.2.1 Accessing GPS data from QML

Qt Mobility's location module includes elements such as `PositionSource`, `Coordinate` and `Place` that make composition of a location aware mashup effortless. The GPS positioning is enabled with `PositionSource` element that gains the position information from device peripheral. This element allows using a separate file containing raw NMEA data for testing if real GPS data is not available. Listing 9 shows definition of the `PositionSource` element used in the main QML file. The element is activated when the application starts and it attempts to obtain the position information in every 10,000 milliseconds. We have attached an inline JavaScript function to the position change event of the `PositionSource`. In this function, we check if the position information is valid, and the user of the application can be shown on the map. Whenever a valid position data is acquired, the data is saved to the database with `savePosition` function.

Listing 9 Definition of the PositionSource element

```
PositionSource {
   id: myPositionSource;
   active: true;
   updateInterval: 10000;
   onPositionChanged: {
     if(
       !myPositionSource.position.longitudeValid ||
       !myPositionSource.position.latitudeValid
       ) {
       myself.visible = false;
       tracking = false;
       return;
     }
     tracking = true;
     map.center = myPositionSource.position.coordinate;
```

```
      myLat = myPositionSource.position.coordinate.latitude;
      myLng = myPositionSource.position.coordinate.longitude;

      var point = map.toScreenPosition(
                myPositionSource.position.coordinate
              );
      myself.visible = true;
      savePosition();
   }
}
```

4.2.2 Mashing up GPS data and other content

Accessing resources over the internet is straightforward with Qt Quick. The network transparency enables to use URLs to locate resource files – images or fonts, for instance. The Qt Quick supports the XMLHttpRequest object, which can be used to asynchronously obtain data over the network. The XMLHttpRequest follows the same W3C standard as many popular web browsers with few exceptions. It does not enforce the same origin policy nor support synchronous requests.

Listing 10 Obtaining data from Persevere database with XMLHttpRequest

```
function getPersevereData(url, callback) {
   var doc = new XMLHttpRequest();
   doc.onreadystatechange = function() {
     if (doc.readyState == XMLHttpRequest.DONE) {
       if(doc.status == 404) {
         console.log("Sorry, error 404!")
       }
       else if(doc.status >= 200 && doc.status < 300) {
         callback(doc);
       }
     }
   }
   doc.open("GET", url);
   doc.send();
}
```

In order to make accessing our Persevere database effortless, we have created a small library in a separate JavaScript file, which is included in the QML project with `import "./qml/persevere.js" as DB` statement. This library consists of two functions – one for saving data and another for requesting it. Listing 10 shows the latter, `getPersevereData` function, which can be used to obtain data from the database. We use the XMLHttpRequest object to send a GET request to the server, and then attach an inline function to handle the state changes of the request. When the request state is

DONE, the callback function provided is executed, if the request was successful. The Listing 11 demonstrates how the library is used from the main QML file. QML `Timer` element is used to call the function `updateLocalUserData` in every 20,000 milliseconds to keep the mashup data up to date.

Listing 11 Using JavaScript library for database access in regular intervals

```
Timer {
   running: true;
   interval: 20000;
   repeat: true;
   onTriggered: {
     updateLocalUserData();
   }
}
```

5 Results and discussion

In this paper, we introduced two mashup runtime environments intended for embedded devices. These environments have different perspective – one with procedural and another with declarative approach for mashup composing. Both environments can be used to access arbitrary device peripherals as well as resources over the internet with a liberal security model.

Compared with existing solutions described in Section 2.2, environments presented in this paper can be used to access arbitrary device peripherals in different types of embedded devices. The existing solutions are restricted to proprietary devices. Furthermore, while a modified browser as a mashup platform might be overkill for performance, environments described here consume fewer resources. In addition, both Qt Script and Qt Quick-based environments are based on Qt application framework that is available for numerous existing operating systems on desktop and on embedded devices. Therefore, the mashup already programmed and tested for one platform can be used on another relatively easily.

When the two mashup runtime environments, Qt Quick and QtScript, are compared, it is clear that with the former a mashup composer can work on much higher level than with the latter. If necessary native libraries and interfaces for accessing device peripherals are readily provided, the programmer can focus on combining the data and polishing the visual appearance of the mashup. However, if the target device manufacturer, or some third party, does not provide the necessary interfaces, much more skill is needed when the peripherals are accessed with native code and bindings for Qt Quick are created. Especially data exchange between the dynamic Qt Quick and static Qt C++ can be difficult to implement. Therefore, using the QtScript-based approach might be more convenient for some programmers, as the whole mashup can be build dynamically, including the peripheral accessing.

While in principle JavaScript could be replaced with other scripting languages in our implementation, its use has special advantages. In particular, a number of web services offer a JavaScript API that can be accessed in a natural fashion. Luckily, despite its

reputation as a language that is not well-suited for serious applications, JavaScript is actually a nice language when overlooking parts that are directly associated with the browser, such as the I/O model which can agreeably be regarded as degenerated (Crockford, 2008; Mikkonen and Taivalsaari, 2007).

Although the application example provided in this paper is simple, it generalises to more complex cases as well. However, depending on the implementation environment, different facilities could be used. For example, in certain embedded systems, the D-Bus system (http://dbus.freedesktop.org/) would most likely be a better option than a socket, since more advanced services could be offered in a simpler fashion. This however would require generating additional plugin interfaces for the underlying JavaScript environment either manually or automatically. Based on some practical experiments, neither of the options seems to be overly difficult to implement in practice.

Different ways to deliver data from one device to another provides further research issues. While it is in principle possible to include a web server in numerous devices, the smallest sensor-based systems will most likely require a lower-level interface. Moreover, issues associated with connectivity must also be taken into account, since some configurations can be based on near-field communication. Sharing information originating from different sources for collaborative applications in turn requires collaborative means in the form of a web server or peer-to-peer architecture established between some of the devices. In such a network, bearers other than WLAN, such as Bluetooth, could be used as well.

Finally, while the use of a separate runtime environment has provided us additional flexibility for accessing external resources, the approach raises obvious concerns regarding security. In our case, we believe that extensive work performed earlier in other contexts – such as Mobile Java, where numerous interfaces have been specified (Riggs et al., 2003) – can be applied. However, since we are aiming at developing real applications, the usual single-origin security model is not adequate unless individual devices provide their own code as well.

6 Conclusions

In this paper, we have described a fashion to liberally and flexibly combine content that is downloaded from the web and data available in embedded devices. As the method of implementation, we used JavaScript, the lingua franca of the web, for which an interpreter is available in all web browsers as well as in numerous other systems, such as the Qt framework that we used as the vehicle of implementation. While only a simple implementation was presented, we believe that with adequate connectivity the system can be generalised to a number of different configurations, where devices, sensors and arbitrary actuators provide application-specific data.

Our approach is based on using a separate runtime. We do not consider this as an essential restriction, since if needed the same (or similar) runtime could be included inside a browser as a plugin, thus enabling data access from within the browser. Furthermore, the underlying application framework, Qt, is already in use in numerous embedded devices. Therefore, by enabling scripting access to local resources, the embedded devices would be in reach for truly pervasive mashups.

References

Brodt, A., Daniela, N., Sailesh, S. and Bernhard, M. (2008) 'Context-aware mashups for mobile devices', *Proc. 9th Int. Conf. on Web Information Systems Engineering*, Auckland, New Zealand, pp.280–291.

Crane, D., Pascarello, E. and James, D. (2005) *Ajax in Action*, Manning Publications Co., Greenwich, CT, USA.

Crockford, D. (2008) *JavaScript: The Good Parts*, O'Reilly Media, CA, USA, ISBN: 978-0-596-51774-8, ISBN 10: 0-596-51774-2.

Flanagan, D. (2006) *JavaScript: The Definitive Guide*, 5th ed., O'Reilly Media, CA, USA, ISBN 978-0-596-10199-2, ISBN 10: 0-596-10199-6.

Koskela, T., Kostamo, N., Kassinen, O., Ohtonen, J. and Ylianttila, M. (2007) 'Towards context-aware mobile Web 2.0 service architecture', *Proc. Int. Conf. on Mobile Ubiquitous Computing, Systems, Services and Technologies*, IEEE Computer Society, pp.41–48.

López-de-Ipiña, D., Vazquez, J.I. and Abaitua, J. (2007) 'A Web 2.0 platform to enable context-aware mobile mash-ups', *Proc. European Conf. on Ambient Intelligence*, Darmstadt, Germany, 7–10 November, pp.266–286.

Maximilien, E.M. (2008) 'Mobile mashups: thoughts, directions, and challenges', *IEEE Int. Conf. on Semantic Computing*, pp.597–600.

Mikkonen, T. and Salminen, A. (2010) 'Towards pervasive mashups in embedded devices', *Proc. of the 16th IEEE Int. Conf. on Embedded and Real-Time Computing Systems and Applications*, IEEE Computer Society, pp.35–42.

Mikkonen, T. and Taivalsaari, A. (2007) 'Using JavaScript as a real programming language', Tech. Rep. TR-2007-168, Sun Microsystems Laboratories.

Mikkonen, T. and Taivalsaari, A. (2008) 'Web applications – spaghetti code for the 21st century', *Proc. of the 6th ACIS Int. Conf. on Software Engineering Research, Management and Applications*, IEEE Computer Society, pp.319–328.

Mikkonen, T., Taivalsaari, A. and Terho, M. (2009) 'Lively for Qt: a platform for mobile web applications', *Proc. Sixth ACM Mobility Conf.*, Nice, France, 2–4 September, pp.1–8.

Nyrhinen, F., Salminen, A., Mikkonen, T. and Taivalsaari, A. (2009) 'Lively mashups for mobile devices', *Proc. of the First Int. Conf. on Mobile Computing, Applications and Services*, San Diego, CA, 26–29 October.

Paulson, L.D. (2005) 'Building rich web applications with Ajax', *Computer*, Vol. 38, No. 10, pp.14–17.

Riggs, R., Taivalsaari, A., Van Peursem, J., Huopaniemi, J., Patel, M. and Uotila, A. (2003) 'Programming wireless devices with the Java™ 2 platform, Micro Edition', 2nd ed., Addison-Wesley Java Series, Pearson Education, New Jersey, USA, ISBN-10: 0201746271, ISBN-13: 978-0201746273.

Publication VI

Developing client-side mashups: Experiences, guidelines and reference architecture

A. Salminen, F. Nyrhinen,
T. Mikkonen and A. Taivalsaari

Editors: Artur Lugmayr, Olli Sotamaa, Heljä Franssila, and Hannu Kärkkäinen

Developing Client-Side Mashups: Experiences, Guidelines and Reference Architecture

Arto Salminen
Tampere University of Technology

Tommi Mikkonen
Tampere University of Technology

Feetu Nyrhinen
Institut für angewandte Systemtechnik Bremen GmbH

Antero Taivalsaari
Nokia Research Center

Abstract

Software mashups that combine content from multiple web sites to an integrated experience are a popular trend. However, methods, tools and architectures for creating mashups are still rather undeveloped, and there is little engineering support behind them. In this paper we present guidelines that can serve as a helpful starting point for the design of new mashups. Guidelines focus mainly on mashup creation methods. Furthermore, we describe a reference architecture for client-side mashup development. In addition, we provide insight into mashup development based on our practical experiences in implementing various sample client-side mashup applications and tools for creating them. The long term goal of our work is to facilitate the development of compelling, robust and maintainable mashup applications, and more generally ease the transition towards web-based software development.

Keywords: Web applications, mashups, mashup development, web-based software development, web engineering.

1. Introduction

Software mashups that combine content from multiple web sites are a hot trend. It is becoming increasingly common to find compelling web applications that aggregate and deliver images, text, and other data from numerous web sites in an innovative and often entirely unforeseen fashion. The ability to combine code and content from multiple sources from anywhere in the world has opened up entirely new possibilities for software development. The

trend towards software mashups has given rise to a number of environments and tools that have the specific objective to make mashup development easier. However, due to the relatively *ad hoc* nature of mashups, it has been – and still is – difficult to provide developers with general purpose tools for mashup development. Furthermore, there are various restrictions and technical limitations that arise from the design of the web browser, such as the same-origin principle (Rudeman, 2010) that prevents a web client from easily downloading data from multiple web sites. Common file formats such as XML and JSON (JavaScript Object Notation), techniques such as RESTful web services (Fielding & Taylor, 2002), and JavaScript libraries such as Dojo (http://www.dojotoolkit.org/), jQuery (http://jquery.com/) and Scriptaculous (http://script.aculo.us/) have turned out to be invaluable in mashup development, though.

Both server-side and client-side mashups can be implemented. Today, most mashup development tools (see a summary of the tools in Nyrhinen, Salminen, Mikkonen & Taivalsaari (2009) and Taivalsaari (2009)) are intended for server-side use, that is, the downloading, processing and generation of web content is performed on the server. In client-side mashups, in contrast, the downloading and combination of web content is performed on the client (e.g., in a web browser running on a desktop computer), typically utilizing the JavaScript language (Flanagan, 2006; Crockford, 2008) and additional libraries to implement the application.

In this paper we examine technical development of client-side mashups and the characteristics of client-side mashups in general. Based on our hands-on experiences in developing various client-side mashups and tools, we provide a novel set of guidelines that can help a developer choose the right methods when building new mashups. Even though mashup patterns for enterprise server-side mashups have been described (Ogrinz, 2009), this kind of a set of practical methods for creating client-side mashups has not been available earlier. Furthermore, in order to enable different vendors to provide service interfaces and to compose mashup clients, we describe a reference architecture for client-side mashups. In summary, the goal of the paper is to give an extended overview of our experiences on client-side mashup development. This paper is an extended version of our earlier papers (Salminen, Nyrhinen, Mikkonen & Taivalsaari, 2010; Mikkonen & Salminen, 2011).

The paper is structured as follows. Section 2 discusses mashups and mashup development in general. In Section 3, we group our findings into practical guidelines that can be applied to the development of new mashups. In Section 4, we describe our mashup reference architecture. In Section 5, we provide hands-on mashup development examples based on real-life applications that we have created. In Section, 6 we discuss the experiences and lessons learned during the implementation of these applications. In Section 7, we draw some conclusions and discuss the directions for future work, including the steps that pave the way towards *mashware* (Taivalsaari, 2009) – full-fledged mashup applications that consist of software components that have been downloaded from multiple sites and then dynamically combined into new applications.

2. Background and Related Work

A typical software mashup combines data, images, code, and other content from multiple sources into a new user experience. In today's mashups, data and content are downloaded most commonly from different web sites, but in principle data from any source (such as the user's local computer or an intranet database) can be used. In addition to aggregating data from multiple sources, mashup applications typically provide an alternative user interface or add advanced filtering or visualization capabilities to a web service.

An important distinction exists between portals and mashups. Portals are web pages that contain information that is retrieved from different sources. Usually the user interface of a portal consists of numerous "portlets", separate pieces of content that are presented in unified way. However, in contrast to mashups these portlets are isolated from each other, and cannot communicate with each other. Mashups are more integrated applications that are constructed in such a fashion that the user typically cannot distinguish data origins. Furthermore, mashups usually create entirely new visualizations from the information they are based on instead of just aggregating the information into a single view.

For historical reasons, the majority of mashups are generated on the web server. Since numerous restrictions exist for composing mashups inside the web browser (on the client), the tools that have been introduced for mashup development are often server-based. Most of these tools include a hosting service that is also located on the web server. The server-side approach allows the developer to circumvent certain typical, recurring limitations, such as cross-domain AJAX requests that are not permitted in an off-the-shelf browser because of security issues. There is a W3C working draft document Cross-Origin Resource Sharing (van Kesteren, 2010) that provides guidelines on how the cross-domain issues could be possibly solved in the future.

Despite the ever-increasing role of the web browser as target platform for mashups and software applications more generally, the web browser is not an essential requirement for mashup development. On the contrary, there are technologies that utilize custom-built, special-purpose web runtimes or native clients that can bypass the above mentioned limitations and can offer better performance, e.g., by performing the client-side mashup generation using native processing capabilities. For instance, mashups intended for mobile devices often utilize a custom-built client environment. Differences between these two types of mashups, i.e. client- and server-side mashups, are presented in Table 1.

Table 1. *Differences between server-side and client-side mashups.*

Server-side mashups	**Client-side mashups**
Processing takes place on a server.	Processing takes place on a client.
Resources can be scaled up.	Resources are limited.
Amount of network traffic high.	Amount of network traffic is usually low.
No offline functionality.	Offline functionality possible.
No same origin policy issues.	Same origin policy is an issue if the runtime environment is a web browser.
When the application is updated, no user interaction is required.	Updating may require user interaction.

Software mashup development has gained a lot of research interest recently, and different patterns and trends can be identified. For example, Wong & Hong (2008) have categorized mashups into five different groups: *aggregation*, *alternate UI & in-situ use*, *personalization*, and *focused view of data and real time monitoring*. Furthermore, Lee, Tang, Tsai & Chen (2008) present seven mashup patterns: *data source*, *process*, *consumer*, *enterprise*, *client-side*, *server-side* and *developer assembly mashups.* In addition, a number of challenges related to mashup development have been pointed out. As stated by Zang, Rosson & Nasser (2008), mashup developers encounter problems mainly in three areas: *API functionality*, *documentation* and *coding details*. Issues related to API functionality in their research were, for example, authentication and performance problems. Some developers were concerned about the lack of proper documentation, i.e., API reference, tutorials and examples. The programming skills needed for creating compelling mashups in JavaScript were also identified as hard to learn. Mashup architectures for enterprise use have been studied before by López, Bellas, Pan & Monoto (2008; 2009). Their work describes a server-side mashup tool that defines four-layer architecture for mashups. This tool can be used to create mashup in declarative fashion.

Mashup development can be facilitated considerably with purpose-built tools. We have provided a detailed summary of available mashup development tools in an earlier paper (Taivalsaari, 2009) (see also Nestler (2008) and Yu, Benatallah, Casati & Daniel (2008)). Examples of mashup development tools include *Dapper* (http://www.dapper.net/open/) and *IBM Mashup Center* (http://www-01.ibm.com/software/info/mashup-center/) – IBM's enterprise mashup platform. As it is typical in the domain of mashup development, the pace of development is rapid and many tools have been released and then later discontinued – sometimes after only a relatively short time frame. Examples of discontinued tools include *Microsoft Popfly* and *Google Mashup Editor*.

3. Guidelines for Successful Mashup Development

Based on our experiences and observations we will next provide guidelines for successful mashup development. The guidelines are divided into a number of areas, starting from technical issues in mashup design and issues related to interfacing with existing web services, ranging to broader issues such as standardization and legislation.

3.1 Mashup Design

Design applications according to software engineering principles. When designing mashups, adhere to established software engineering principles. Whenever possible, separate the user interface, the business logic and the data from each other. Later we describe our mashup reference architecture that can be used as a starting point. Utilize JavaScript libraries that enhance cross-platform compatibility by taking into account the differences between web browsers. If a mashup is based on JavaScript and Ajax, choose a suitable library that has a good support for JSON, JSONP (Özses & Ergül, 2009) and XML parsing if necessary. Evaluate the services that you want to utilize thoroughly. Create a plan for the implementation and follow the plan.

Design for diversity. There are numerous web browsers and web-enabled devices in use today, and new types of devices are introduced regularly. These target environments are not homogeneous but may differ considerably from each other, especially if the mashup is intended to be used in mobile devices as well. Generally speaking, it is advisable to design for the lowest common denominator. Keep in mind the traditional limitations of mobile devices. For example, input methods that are common in desktop computers – such as *onmouseover* events or right-clicking – might not be available or usable on a mobile device. Battery consumption, screen size and issues caused by limited network bandwidth or intermittent connections must be kept in mind. If your mashup application runs in a web browser, it is possible to detect the presence of mobile browsers and provide a mobile-optimized version of the application.

One size does not fit all. Although it would be nice to develop only a single solution for all target platforms, in most cases the platform for which a mashup is intended cannot be neglected. Mobile, desktop and server mashups usually require a different approach. The essential question is how the differences should be taken into account. If the majority of computation is performed on the server, even those client devices that have limited resources and capabilities may be able to run the mashup. Nevertheless, be prepared to write several versions of your mashup if you intend to reach from desktop environments all the way to mobile devices with limited screens and resources.

Pay special attention to security. Web security is based on a number of relatively simple principles, such as the use of SSL/TLS and the same origin policy (Rudeman, 2010). Bypassing the browser security mechanisms is not recommended. Although mashups commonly obtain data from multiple domains, the downloading should be performed in such a way that executable code can only arrive from a single domain. For example, the use of JSONP (JSON with Padding) is a security hazard, since the use of **<script>** tags exposes the application to cross-site scripting attacks. It is therefore important to choose services accessed with JSONP carefully. In addition, it is recommended to use the native JSON parsing in the browser or use a well-established JavaScript library, instead of "parsing" JSON with the fundamentally unsafe *eval()* method (see Crockford (2008)). As a general principle, third-party code should never be *eval*'d without sanitization before execution.

Test carefully. Test your mashup carefully. Run tests also with a low bandwidth connection and in the offline mode. Use tools and libraries that make testing easier. It is recommended to choose a suitable JavaScript library with unit testing capabilities and to utilize debugging tools such as the Firebug add-in for the Firefox browser. Do testing with different browsers and browser versions, since there are still considerable differences between browsers, especially with those browsers that are available on mobile devices. Furthermore, remember to provide useful feedback to the user about the state of the application. This is the first general principle mentioned in ten usability heuristics originally developed by Molich & Nielsen (1990) and later refined by Nielsen (1994). If your mashup is intended to be used as a building block for further mashups, separate the developer and end-user messages clearly. A debugging message popping up is invaluable for a developer, but it can drive an end-user away.

Consider using quality evaluation frameworks. Cappiello, Daniella & Matera (2009) have studied the mashup components from the quality point of view. Their evaluation criteria contain various categories such as the software API, the data provided by a component, the documentation, and the user interface. In addition to more

tangible quality measures, one cannot overlook the stability aspect either, since the used components should be available in the future as well to ensure the functionality of the application.

Be visually impressive & keep usability in mind. Combine data and other content in an innovative, visually impressive way. Use innovative data visualization techniques to help the user to handle complex information. Most users "see only what they see" and their perception of quality is based almost completely on the look-and-feel of a system. Therefore, user interface issues are really important. Edward Tufte's books on data visualization (e.g., (Tufte, 1983)) can serve as a helpful starting point. We have covered usability issues related to mashup development in other papers (see e.g. Nyrhinen et al. (2009)); we intentionally exclude the more detailed treatment of that subject from of this paper.

3.2 Interfacing with Web Services

Build on solid ground. Prefer established, well-documented web services and APIs that are of high quality as well as stability in order to guarantee that the required web services or components will be available also in the future. The probability that the service will exist in the future is higher when using APIs and services from established companies. The stability of the interfaces should not be underestimated, since even established services are typically changed over time. Consider using open databases and wikis as sources for your mashups, since this will decrease the dependence on a single vendor. However, the use of community-provided data and services has possible drawbacks as well, since such services tend to be more likely to change or be temporarily unavailable.

Adapt new technologies carefully. If you are planning to make use of experimental technology, bear in mind that the interfaces, support in browsers and other details are likely to change unnoticed. Furthermore, new technologies are not available for all users. If reasonable, consider providing an alternative version of the mashup with the same functionality implemented with established technologies. Documentation of new frameworks and interfaces may be nonexistent or obsolete. Examples or tutorials can be missing. These factors can slow down development and make it more difficult. Moreover, experimental technology may cease to exist and you need to find another way to create the mashup.

Anticipate changes. Do not expect services and their interfaces and data formats to remain the same over time. Avoid binding your application too tightly to a specific service or a data format. Avoid "scraping" data directly from an HTML page, since data read in such a format will almost surely become unusable or unavailable sooner or later. Well-designed abstraction and adaptation layers make it easier to switch to another service when needed. Whenever possible, write code that is loosely coupled with foreign components/data, especially when utilizing unreliable and frequently changing web resources.

Send notification and provide a fallback solution upon service breakdown. Construct mechanisms that notify you (the developer) automatically if your application breaks down. Implement fallback mechanisms, e.g., by automatically connecting to an alternative web service, to ensure the functionality of your application is preserved even while some of the required services are unavailable. The use of JavaScript error handling constructs such as *try...catch* statements is strongly encouraged, since mashup applications are far more likely to suffer from unexpected errors than traditional applications are.

3.3 Broader Considerations

Follow standards and recommendations. When using content or data from other websites and services, rely primarily on established services with well-documented APIs that provide content using open standards and commonly used formats, such as RSS, XML, JSON and so forth. Data that is available in a standardized format is much more likely to remain available in the same format in the future, whereas service-specific formats are much more likely to change. In addition, follow the recommended JavaScript programming guidelines and practices (Crockford, 2008) to improve the quality, security and performance of your application.

Pay attention to legal matters. Legal and intellectual property issues have to be considered carefully before using any external web service or API. Different vendors have very different views on mashups. Some vendors encourage their services to be used in as many places as possible, while some vendors strictly forbid the use of their services in other contexts. HTML "screen scraping" – that is, harvesting data directly from web pages by imitating a user accessing the site – is troublesome from the legal point of view. Remember that some of the content available in web services may be coming from third parties, and their (copy)rights must be honored as well. For example, the Flickr API can be used to access content even with "all rights reserved" notices, and the mashup

developer must comply with the photo owners' restrictions as defined in Flickr API Terms of Service (http://www.flickr.com/services/api/tos/).

4. Reference Architecture for Client-Side Mashups

While the development of mashups has sometimes been considered mostly an *ad hoc* activity – see for instance (Harmann, Doorley & Klemmer, 2008) – there are commonalities in the goals of different mashup systems. To begin with, all these projects build on top of parallel activities and the innovativeness of a large group of independent service providers. Moreover, they aim at providing superior user experience without overlooking other important quality attributes of the system such as security, performance, availability and malleability. Furthermore, since mashups are generally built using the available means for combining components, mashup projects can build upon other mashups. However, little attention has been paid to the architectural choices that have been made in mashup projects, and the creators of these projects have often come to their solutions intuitively or via trial and error. Hence there is little guidance available for the creators of new mashups regarding the architectural structuring of these systems.

In this section, we describe a reference architecture structure for client-side mashup development (Figure 1). As is common with reference architectures – generic structures that define terminology and functions of a system in a certain field of problems – a practical implementation of such a reference architecture may consist of a subset of the components presented here – some of the components may be merged or omitted totally if they play no role in the application that is being constructed.

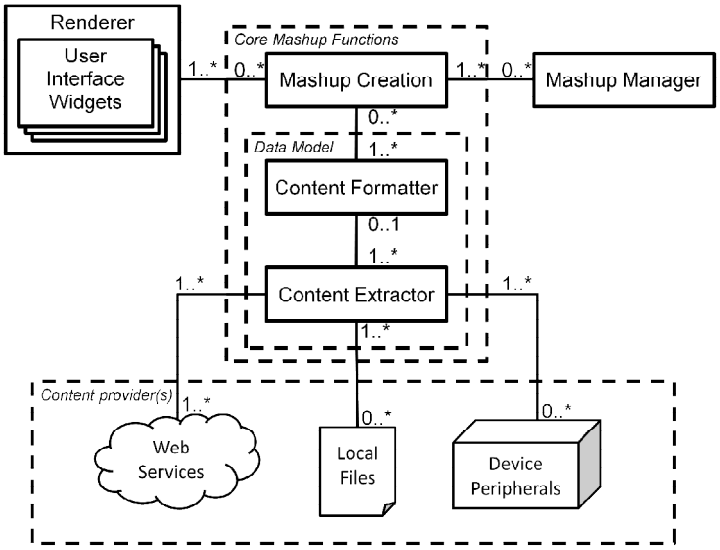


Figure 1. Reference architecture for client-side mashups

4.1 Reference Architecture Components

Renderer. Since mashups are generally used for visualizing available data, a user interface layer is needed. Because visualization commonly requires graphical elements, most mashup systems rely on an existing set of widgets that usually also support interaction with the user. Numerous widget libraries are available. The use of a certain set of widgets typically has a major effect on the look and feel of the application. Note that widget libraries tend to utilize very different binding techniques to connect user interface elements and data, placing a significant impact on the overall design of the application. In some cases, depending on the capabilities offered by the widget library, it may be necessary to merge mashup creation operations with the user interface components. However, in the general case, these functions are distinct: the user interface is responsible for rendering the data produced by the mashup function and for interacting with the user in general, whereas the underlying mashup functionality performs the actual combination of data that creates the data to be rendered.

Mashup creation. The requirements for combining data tend to vary significantly between different types of mashups. Many applications compose content in different layers, placing for example graphical shapes on top of a map offered by one of the content providers. Some other systems apply pipes-and-filters architectural pattern in

which content is processed in different way. As a general principle, mashup creation should occur independently of user interface aspects. However, in practice user interface aspects such as layering will have a significant impact on mashup creation.

Mashup manager. To overcome the fragile nature of mashups and to account for potential changes in the source material used for mashups, a mashup system should have a mashup manager component. This component ensures that mashup content is available and has the ability to adapt to changes that may occur for instance, when mashup content providers change their interfaces. For example, the mashup manager may decide to use a backup service when the primary service is not available. Similarly, the mashup manager is responsible for defining whether the system should run in online or offline mode; the component may maintain local storage to support offline use of the service. The mashup manager is responsible for security, as well as selecting the renders, formatters and extractors that are used for a particular client device.

Content formatter and extractor. Content formatter and extractor together compose the mashup data model, i.e., the intermediate representation that is used for generating the visual representation of the mashup. The data model maintains the internal data that is used by the application for various purposes. The data model includes data accessors and extractors that download data from different sources (e.g., the Web, the local file system or device peripherals), as well as data formatters that manipulate the data provided by the extractors into an intermediate representation that is easier to work with. Internally, content formatters and extractors can be layered, with different layers being responsible for different tasks, such as low-level interfacing, data scraping, formatting changes, and so forth.

Content providers. In order to extract content, there need to be interfacing capabilities to the content providers, be it data on the Web, local data residing in the device itself, or something that is generated on the fly based on, e.g., location (GPS) data. While not a part of the reference architecture per se, these features are an integral part of any mashup system, be it a system that is building on top of an already existing web service or simply something that is readily available in the device itself.

Our architecture can be compared with two other popular web application architectures: *three-tier* and *Model-View-Controller* (MVC) architecture. Three-tier architecture is a server-client architecture in which presentation, application logic and data are separated into independent modules (Eckerson, 1995). As the presentation module never interacts directly with the data, three-tier architecture is conceptually linear (see Figure 2). MVC architecture originates from the Smalltalk system (Reenskaug, 1979a; 1979b), and it was originally designed to facilitate the creation of graphical user interfaces (Krasner & Pope, 1988). The MVC architecture consists of three components: a Model that stores and manages data, a View that renders the user interface and a Controller that defines how the user interface reacts to user input. In contrast to three-tier architecture, MVC architecture is triangular as presented in Figure 2.

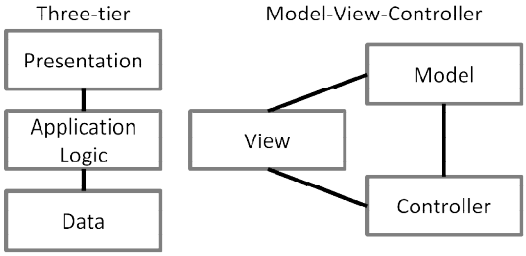


Figure 2. Three-tier architecture (left) and MVC architecture (right)

Our mashup reference architecture resembles the three-tier architecture model. It is linear in a sense that the renderer (presentation) always interacts with the content formatter and extractor (data) through the mashup creation module (application logic). However, as we mentioned earlier, in some cases it is necessary to merge the mashup creation operations with the user interface aspects. Therefore, some implementations may resemble the MVC model. In that case the renderer (view) interacts with the mashup creation module (controller), and instead of the mashup creation module handling the result from the content formatter and extractor (model), the result is passed directly to the renderer. In these kinds of mashups it is typical that the user interface widgets show only one type of content.

Next, we introduce a number of sample systems and applications upon which the reference architecture above is based.

5. Sample Client-Side Mashups

In this section we present three hands-on examples of mashups that we have developed. The mashups are built on set of popular web service APIs to demonstrate different techniques and options that can be used in client-side mashup development. Even though our examples are developed as client-side mashups, some parts of the functionality reside on the server-side. For instance, mashups using Google Maps rely on Google's servers that provide the map tiles as part of the graphical user interface. Note that many of these mashups have been created for use in mobile devices. We have described the special aspects related to mashup development for mobile devices in another paper (Nyrhinen et al., 2009).

Since JavaScript is the only programming language that is readily available inside all commercial web browsers, the JavaScript language has become the dominant development language for mashups that require client-side processing. All the mashup examples shown in this section have been written in JavaScript. In general, the mashups that we built have been implemented using typical Web 2.0 technologies such as Ajax, XHTML and JavaScript. For those applications that are intended for mobile devices, we have used Lively for Qt (http://lively.cs.tut.fi/qt) – a custom mobile runtime environment that makes the Qt APIs (http://qt.nokia.com/) accessible from JavaScript code – as well as Qt Quick, which is a declarative technology for creating mobile applications. Qt is a popular cross-platform application and UI development framework. The Qt APIs can considerably simplify the development of compelling mashups especially for Nokia devices.

5.1 Twitter Image Mashup

Our first example, the Twitter Image Mashup (Figure 3) combines Twitter (http://twitter.com) entries ("tweets") with images from Flickr (http://www.flickr.com/), a popular image hosting website. The idea of the mashup is to present tweets in visually attractive way and combine those with images that are related to the tweet's topic. Using the Twitter user name that the user has provided, the application acquires the last two hundred tweets and presents them on a timeline. When the user selects a tweet, one of the words in the entry is selected randomly and used as a keyword for the Flickr search. The image retrieved is then displayed alongside the tweet. In Table 2 we have described how this mashup follows the guidelines and the reference architecture presented earlier.

Figure 3. Twitter Image Mashup (Image of cars by Luke Jones, available under Creative Commons Attribution License).

Table 2. *Evaluation of the Twitter Image mashup in view of our guidelines and reference architecture*

Mashup Design	Using the mashup with mobile devices is possible, as renderer widgets are compatible with most mobile web browsers. Renderer scales the content according to the screen resolution of the host device. Widgets used in the mashup can be manipulated with touch screen. Data parsing is done with jQuery's XML and JSON parsers.
Interfacing with web services	The mashup uses both JSONP and XML requests through two different proxies to access content from other domains. Mashup is built on the well-established Twitter and Flickr services. Only basic functionality of Flickr (XML) and Twitter (JSON) APIs are used.
Broader considerations	Flickr TOS (Terms of service) require showing a certain message on mashup.
Mashup reference architecture	The content extractor and formatter are combined into a single function, separate for each service. The mashup creation function is executed when a user clicks items on the timeline. Renderer consists of a third party timeline and image presenting widgets. Mashup manager functionality is integrated into content extractor that uses fallback mechanisms to ensure partial functionality.

The Twitter and Flickr services used in this mashup are reliable and established web services. However, if either of the services were inaccessible for some reason, it would be easy to replace the service used with another. For example, if the Flickr image search would become inoperative, our application could use Google Image Search instead. As the image used to visualize the Twitter entry is defined by a URL, only those functions that form the request and parse the response would need to be modified.

In order to simplify the development of this application, and to make the application visually more attractive, some JavaScript libraries were again utilized. The multipurpose JavaScript library jQuery (http://jquery.com) was used for parsing the XML formatted data. The draggable timeline was created using SIMILE Timeline widget (http://code.google.com/p/simile-widgets/). The image gallery component Highslide JS (http://highslide.com/) was used to enable zooming and moving of Flickr images inside the browser viewport.

The Twitter API enables fetching data in the JSON format, which simplifies development considerably. We used embedded <*script*> tags in our HTML code to bypass the restrictions arising from the same-origin policy (Rudeman, 2010). In addition, we utilize a technology called JSONP. JSONP makes it possible to provide a callback function to which the JSON data is passed on so that it can be processed further after the retrieval. Listing 1 shows the HTML code that initiates the fetching of the Twitter user latest status and handles the response from Twitter. When the data has been obtained, the execution proceeds to the callback function *twitterUserStatusCallback*. This function processes the JSON file that is obtained using the Twitter user API by adding the user tweets into the timeline widget.

```
function getTwitterTimeline(username) {
  var url = "http://twitter.com/statuses/user_timeline/" +
    username +
    ".json?callback=twitterTimelineCallback&count=200";
  var script = document.createElement('script');
  script.setAttribute('src', url);
  document.getElementsByTagName('head')[0].appendChild(script);
}

function twitterTimelineCallback(obj) {
  if(obj != undefined) {
    addTweetsToTimeline(obj);
  }
}
```

Listing 1. Using JSONP and the a dynamically added <script> tag to bypass the same-origin policy

Note that this example is not particularly elegant or complete. For instance, it does not take into account security issues that arise from the vulnerability of JSONP to man-in-the-middle attacks; in general, JSONP should not be used to transport sensitive data. If used, the handled data should be sanitized and filtered properly before processing. In addition, using JSONP allows the remote site to inject any JavaScript code into the original website instead of expected clean data wrapped into the specified callback. Therefore, JSONP should not be used, if the remote site is untrustworthy.

Based on a random keyword found from the selected Twitter tweet, an image from Flickr is obtained. Because the Flickr API was used with XML instead of JSON, we could not use a dynamic <**script**> tag to access the data. Instead, we accessed the Flickr API with asynchronous AJAX calls.

To test different methods for bypassing the same origin security restrictions of the web browser, two variants of this mashup were developed. One was implemented using an Adobe Flash object based proxy and another one with a proxy on a server.

The first variant uses a Flash object based proxy to request image data from the Flickr API. This kind of an approach is possible only if the remote server provides a special file that grants access to remote domain Flash objects. Another restriction is that the user must have the Adobe Flash plugin installed and enabled in the web browser. If these two conditions are fulfilled, JavaScript functions can interact with the Flash component and use it to send cross-domain requests. Utilizing the Flash object based proxy does not require any server-side scripting, which may be a significant advantage in many situations.

The second variant utilizes a server-side PHP proxy for image data requests. This kind of a proxy does not have any special requirements for the remote server. No plugin component is required on the client side (in the web browser). However, the mashup application must be placed on the server with scripting support, which can be considered as a downside since in many cases the application developer does not have the necessary rights to control the content on the web server.

5.2 Webcam Map Mashup Using Qt

Our second sample mashup, Webcam Map Mashup (see Figure 4), combines maps from Google Maps (http://code.google.com/apis/maps/) and web camera images from Webcams.travel (http://webcams.travel/developers). The general idea of the application is to automatically locate the nearest web cameras when the user chooses a specific city or location by browsing the map. When a camera is found, a marker is placed on the map at the actual location of the camera. Furthermore, a small semi-transparent image of the corresponding camera view is displayed on the map. When the user places the mouse cursor over such an image, the image is enlarged and made opaque for better viewing. The user can drag the small images around or close them by double-clicking the desired images. In Table 3 we have described how this mashup follows the guidelines and the reference architecture presented earlier.

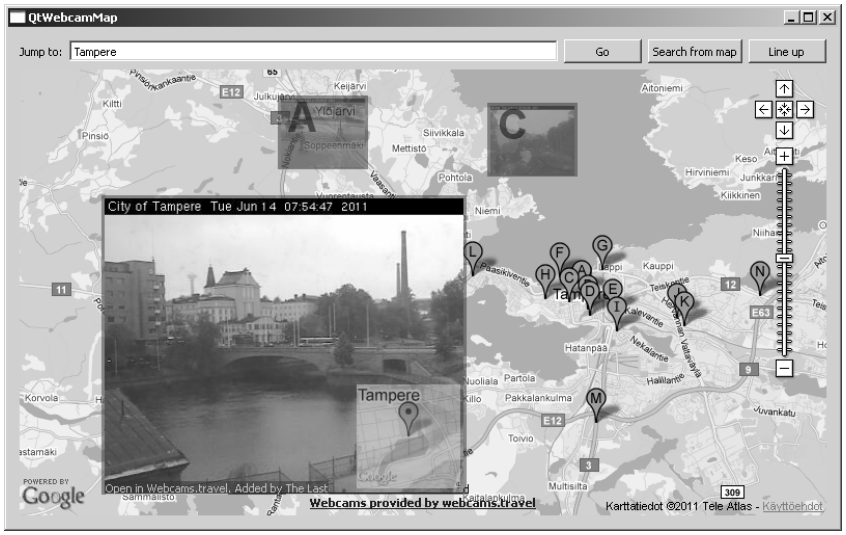


Figure 4. Webcam Map Mashup

Table 3. *Evaluation of the Webcam Map mashup in view of our guidelines and reference architecture*

Mashup Design	Qt-based scripting runtime is used for creating the user interface. On mobile device the user interface scales automatically to full screen and can be manipulated solely with touch input methods. The runtime requires quite a lot of processing power to run smoothly. A recent high-end mobile device is required for enjoyable user experience.
Interfacing with web services	Well-established Google Maps and Panoramio services are used as data sources for the mashup. The Google Maps API version is locked so that it will not change unnoticed.
Broader considerations	Panoramio terms of use require a certain message as well as some links to be shown on the mashup. Google's terms of use deny covering the Google logo placed on the map.
Mashup reference architecture	Qt classes for networking are used as content extractors when accessing Panoramio and Google Maps APIs. Whenever new content is received, it is reformatted and stored using an intermediate representation. Availability of new content triggers the mashup creation function that places the icons representing web cameras on top of the map. Renderer consists of Qt's widgets and a Google's Map component. Rendering data received from Panoramio is performed utilizing Google Map widgets. Mashup manager functionality is integrated into content extractor and mashup creation functions.

The Webcam Map mashup utilizes Qt application framework and the Qt JavaScript bindings constructed using an open source generator (http://labs.trolltech.com/page/Projects/QtScript/Generator). This kind of an approach does not suffer from the same limitations as the more traditional mashups running on the client in a web browser. For instance, the application can flexibly obtain data from any web site, and present the data utilizing the rich GUI widgets and other components offered by the Qt framework. Our experiences in developing mashups with Qt have been described more extensively in another paper (Nyrhinen et al., 2009).

The Google Maps API provides a flexible map component that can be used for visualizing webcam locations. To embed the map component, JavaScript code from Google needs to be added to the web page. The map component can be manipulated with numerous JavaScript functions that can be used, for example, to add or remove markers, change the map location or bind event handlers to user actions.

The *Webcams.travel API* (http://www.webcams.travel/) is used with requests compatible with the RESTful architecture (Fielding & Taylor, 2002). Information about webcams around the map center can be obtained with the standard HTTP/GET operation. The desired data is available in XML, JSON and PHP formats.

Terms of use of the Webcams.travel API (http://fi.webcams.travel/developers/terms) require that every webcam image displayed must have a link to the webcam at Webcams.travel webpage, and the name of the user who owns the webcam must be shown in the context of the webcam, with a link to the user profile. Furthermore, the terms of use require that a link to Webcams.travel with the text ”Webcams provided by webcams.travel” must be included. All these requirements were taken into account when composing the mashup, as can be seen in Figure 4.

The use of Qt makes cross-site network requests simple. Listing 3 shows an example in which a URL is created and the class *QNetworkAccessManager* is then used to request webcam data. When the request is complete, a separate callback function *getCamsCallback* will be invoked to handle the results.

```
QWebcamMap.prototype.getCams = function() {
  var requestStr = this.WT_REST_REQ
     + 'wtc.webcams.list_nearby&devid='
     + this.WT_DEVID
     + '&lat=' + mapCenter.y
     + '&lng=' + mapCenter.x
     + '&format=json';
  var am = new QNetworkAccessManager(this);
  am["finished(QNetworkReply*)"].connect(
     this, this.getCamsCallback
  );
  am.get(
     new QNetworkRequest(
        new QUrl(requestStr)
     )
  );
}
```

Listing 2. Function *getCams*

5.3 Multi-User Location Mashup

Our third example mashup, *Multi-User Location Mashup* (Figure 5), is a collaborative application that utilizes a map component and a location database that contains the current GPS location of the users. The general idea of the mashup is to interactively present the current location of all the users and their status over the map. Nicknames are used for identifying the different users and the status of each user is represented with a simple text string. In Table 4 we have described how this mashup follows the guidelines and the reference architecture presented earlier.

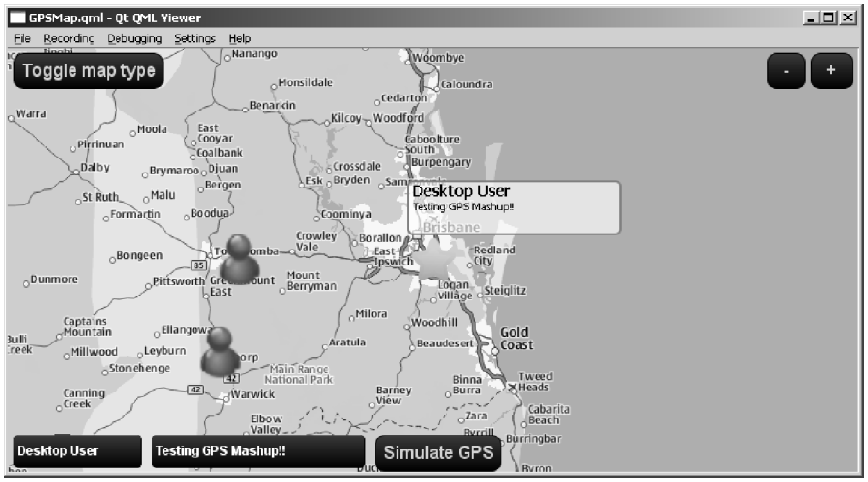


Figure 5. Multi-User Location Mashup built using Qt Quick, a location server and location-based GPS information

Table 4. *Evaluation of the Multi-user location mashup in view of our guidelines and reference architecture*

Mashup Design	Qt Quick is specially designed with touch interfaces in mind. The mashup can be used both desktop as well as mobile devices with touch screen. Colibri widget library is used for the user interface. Qt Quick runtime is designed for devices with limited processing power. Heavy operations are executed with native components. This makes the mashup responsive when it is executed on mobile devices. Some "eye candy" features such as animations and fade-outs are added to make the user interface more attractive.
Interfacing with web services	Qt Quick offers web service interfacing mechanisms that are similar to those offered by the web browsers. This makes communication with the JSON database straightforward. JSON data is parsed using a special purpose library.
Broader considerations	There are no legal issues related to this mashup. Since we use our own server, we could implement a data format that is convenient to work with.
Mashup reference architecture	Qt Quick elements for accessing web resources are used as content extractors. Mashup data updates occur in fixed intervals. When new data is received, it is stored and formatted into custom Qt Quick elements. Mashup function is implemented as a JavaScript function that updates the user interface. Renderer consists of Qt Quick widgets.

The Multi-User Location Mashup was developed as a Qt Quick widget, using declarative QML language to define the mashup user interface as well as the data handling functions. Similarly to Webcam Map Mashup described above, the application is liberated from the restrictions commonly associated with the web browser, especially the same origin policy. This application can also be executed in a mobile device. In that case, the application can utilize the GPS peripheral device that may be available on a mobile device. For testing purposes prerecorded GPS data can be used as a location data instead of live data. *Colibri* user interface widget library is used for buttons and text edit dialogs.

Since this application relies on native Qt interfaces that have access to mobile device peripherals, accessing the GPS peripheral is very straightforward by using a QML Coordinate element for map positioning. Likewise, a native QML Plugin component to show a map tiles is utilized, and therefore the actual mashup is created with very few lines of code. Listing 3 contains an example how the map user interface element is defined. The listing includes two calls to a procedural JavaScript function *setUsersOnMap* that positions the icons representing users on top of the map element.

```
Map {
  plugin: Plugin { name : "nokia"; }
  zoomLevel: 7;
  mapType: Map.SatelliteMapDay;
  center: Coordinate {
            latitude: defaultLatitude;
            longitude: defaultLongitude;
          }
  onCenterChanged: {
    setUsersOnMap();
  }
  onZoomLevelChanged: {
    setUsersOnMap();
  }
}
```

Listing 3. Defining a Map user interface element of a mashup

At the implementation level, this application utilizes an open source JSON database and JavaScript application server called *Persevere* (http://www.persvr.org/). Persevere is very convenient when using JavaScript, especially when accessing data through JSON HTTP/REST calls.

6. Experiences

Following the guidelines and the reference architecture was successful in our example mashup implementations. In the following we share our experiences on some particular details.

Mashups are generally very vulnerable to changes that occur when the data and interfaces offered by web services are modified. It is also relatively common for web services that are still under development or of beta quality to become unavailable for extensive periods of time. Our experience is that even well established web services may have occasional service breakdowns. Furthermore, mashups that rely on the availability of specific data and specific data formats and interfaces tend to break down easily when such changes occur. Such experiences motivated and underlined our advice to “send notification and provide a fallback solution upon service breakdown”.

Data exchange in mashups is commonly based on established formats such as JSON and XML. However, in many cases application-specific or service-specific custom formats are used. At times, the developer has to “scrape” and parse the required data manually from an HTML page using the limited facilities that are available in the web browser. This can be rather cumbersome and slow. Furthermore, the asynchronous loading of data that is characteristic of web applications today often causes timing problems when data is received from multiple sources. This is another reason to follow our original linear architecture model.

The overlapping nature of web services poses an interesting challenge. It is common that there are several competing services providing similar functionality. This makes it possible to implement fallback mechanisms without degradation of user experience. For example, a map component for a mashup application can be obtained from Google Maps, Microsoft Bing, Yahoo! Maps or OpenStreetMap map services. The terms of use, availability of

the services, features offered by the sites, and response times can vary considerably between different services, though. Moreover, the interfaces of these services are different, sometimes proprietary, and – for some of the services – relatively unstable and poorly documented.

There are many other technical challenges. For instance, some web services have limitations in their APIs regarding the maximum number of requests that are allowed from a single domain within a certain period of time – such limitations are important in protecting the service from denial of service (DoS) attacks. Furthermore, it is common for service providers to utilize mechanisms that monitor the number of requests from a single domain. Some popular services such as Google Maps require a proper API key when making API calls. An API key is a unique string that is issued by the service provider to identify application and bind it to a previously issued domain. Typically license terms strictly forbid the sharing of API keys. This can be a problem when developing widgets or independent web applications that do not use the typical browser-based approach to access the service.

There are major challenges regarding intellectual property rights as well. A mashup is typically bound to several services, with varying terms of use and licensing conditions. It may be impossible to combine certain pieces of available content due to copyright and licensing issues. Protecting the mashup itself from unauthorized use or copying is also very difficult. One can use obfuscation, but this will complicate the development of even more compelling mashups that would build on top of this particular (obfuscated) service.

Adhering to our mashup reference architecture can be difficult with some implementation technologies. For instance, the graphical widgets of the Qt library follow the MVC architecture in managing the relationship between the data and the way it is represented to the user. The dependencies assumed by the built-in MVC architecture in those components can make it difficult to adapt the components to a reference architecture such as ours. Such problems are common in any software system that utilizes multiple component frameworks or design paradigms simultaneously.

7. Conclusions and Future Directions

Software mashups – web applications that are based on the ability to combine code and content from multiple web sites all over the world – have given rise to an entirely new way of creating software. It is increasingly common to find compelling web applications that aggregate and deliver images, text, and other data from numerous web sites in an innovative and often entirely unforeseen fashion.

In this paper we presented a set of guidelines for mashup design and web service integration. The guidelines can help developers to create compelling, robust and maintainable mashups. In addition, we discussed the standardization and legal issues, as well as presented a reference architecture that can be used as a starting point for mashup development. Furthermore, we presented three hands-on examples of mashup applications, followed by a discussion on our experiences in developing them.

As we emphasized in the paper, mashup development is still a relatively immature area, leaving plenty of challenges for the future work. In general, the development of mashups is subject to a number of problems and limitations. Many of these problems arise from the fact that the web browser was originally designed for viewing relatively simple hypertext documents – not for executing full-blown web applications with complex interaction with various external services. The security mechanisms of the web browser are poorly suited to client-side mashups (Taivalsaari & Mikkonen, 2008). Liberating mashups from such restrictions with caching, proxies, and other fallback mechanisms is an important direction for the future work. Plenty of interesting work remains also in the area of methodological foundations for mashup development, not to mention intellectual property, copyright and licensing issues. One especially interesting future direction would be to evolve our current guidelines towards full-fledged design patterns for client-side mashup development.

The upcoming HTML 5 standard (Hickson, 2011) will open up interesting avenues for the mashup development. Support for techniques such as WebSockets, DOM storage, cross-document messaging and history management in a HTML5 compliant web browser will have a significant impact on mashup programming. It will be very interesting to test these techniques in practice, as well as to evaluate and understand the remaining limitations and challenges. Another interesting development is WebGL that enables procedural graphic interface in web browser. WebGL can be used for creating both 2D as well as 3D graphics without plug-ins components. This relieves developers from the limitations of the DOM-based I/O model of the web browser. From the viewpoint of mashup development, WebGL enables persuasive graphics and even social 3D spaces in which users can collaborate and share their data in novel, innovative ways. The development of large-scale mashups is another interesting area

for future work. At present, most mashups are relatively small in terms of the number of lines of code and overall complexity. The development of large-scale mashups consisting of tens or hundreds of thousands of lines of code is still largely an unexplored area.

Finally, we are especially excited about the emerging trend towards *mashware* (Taivalsaari, 2009) – full-fledged mashup applications that consist of software components that have been downloaded from multiple sites and then dynamically combined into new applications. So far, most mashups have been built around data and other static components (e.g., images) that have been downloaded from different sources. The future challenge is to go beyond data, and to enable the flexible combination of software components from all over the world. Possible examples of such components include UI components, specialized financial libraries, mathematical libraries, and so on. Mashware could potentially be created by the end users themselves by combining components that are suitable for their needs. The primary obstacle in the development of mashware is security; sadly, with the current security solutions on the Web, the development of applications that combine code from multiple origins is still fundamentally unsafe. We hope that this article, for its part, encourages people to develop solutions for this exciting new area.

References

Cappiello, C., Daniel, F., & Matera, M. (2009). A quality model for mashup components. In Proceedings of *the 9th international Conference on Web Engineering* (pp. 236-250). Berlin, Heidelberg: Springer-Verlag.

Crockford, D. (2008). *Javascript:The good parts.* O'Reilly Media, Inc.

Eckerson, W.W. (1995, January). Three tier client/server architecture: Achieving scalability, performance, and efficiency in client server applications, *Open Information Systems*, *10*(1).

Fielding, R. T. & Taylor, R. N. (2002). Principled design of the modern web architecture. *ACM Transactions on Internet Technology*, *2*(2), 115-150.

Flanagan, D. (2006). *JavaScript: The definitive guide*, 5th ed. O'Reilly Media.

Hickson, A. (2011). HTML 5. a vocabulary and associated APIs for HTML and XHTML. W3C Working Draft 25 May 2011. Retrieved June 2, 2011, from http://dev.w3.org/html5/spec/.

van Kesteren, A. (2010). Cross-origin resource sharing. W3C Working Draft, July 27, 2010. Retrieved June 2, 2011, from http://www.w3.org/TR/cors/.

Krasner, G. E. & Pope S.T. (1988) A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, *1*(3), 26-49.

Lee, C., Tang, S., Tsai, C., & Chen, Y. (2009). Toward a new paradigm: Mashup patterns in Web 2.0. *WSEAS Transactions on Information Sciences and Applications*, *6*(10), 1675-1686.

López, J., Bellas, F., Pan, A. & Monoto, P. (2009). A component-based approach for engineering enterprise mashups. In Proceedings of *the 9th International Conference on Web Engineering* (pp. 30-44). Berlin, Heidelberg, Springer-Verlag.

López, J., Pan, A., Bellas, F. & Montoto, P. (2008) Towards a reference architecture for enterprise mashups. *Actas de los Talleres de las Jornadas de Ingeniería del Software y Bases de Datos*, *2*(2), 67-76.

Mikkonen, T. & Salminen, A. (2011) Towards a reference architecture for mashups. In Proceedings of *the Second International Workshop on Variability, Adaptation and Dynamism in Software Systems and Services* (pp. 647-656). Berlin, Heidelberg, Springer-Verlag.

Molich, R. & Nielsen, J. (1990) Improving a human-computer dialogue. *Communications of the ACM 33*(3), 338-348.

Nestler, T. (2008). Towards a mashup-driven end-user programming of SOA-based applications. In Proceedings of *the 10th international Conference on Information Integration and Web-Based Applications & Services* (pp. 551-554). New York, NY: ACM.

Nielsen, J. (1994) Heuristic evaluation. In *Usability Inspection Methods*, (pp. 25-62). John Wiley & Sons, Inc., New York, NY, USA.

Nyrhinen, F., Salminen, A., Mikkonen, T. & Taivalsaari, A. (2009). Lively mashups for mobile devices. In Proceedings of *the First Annual International Conference on Mobile Computing, Applications, and Services* (pp. 123-141). Berlin, Heidelberg: Springer-Verlag.

Ogrinz, M. (2009). *Mashup patterns: Designs and examples for the modern enterprise*. Pearson Education Inc., Boston, MA, USA.

Reenskaug, T. (1979*) Thing-model-view-editor - An example from a planningsystem*. Technical note, Xerox PARC, May 1979.

Reenskaug, T. (1979) *Models-views-controllers*. Technical note, Xerox PARC, December 1979.

Ruderman, J. (2010). Same origin policy for JavaScript. June 14, 2010. Retrieved July 5, 2010, from https://developer.mozilla.org/en/Same_origin_policy_for_JavaScript.

Salminen, A., Nyrhinen, F., Mikkonen, T. & Taivalsaari, A. (2010). Developing client-side mashups: experiences, guidelines and the road ahead. In Proceedings of *MindTrek 2010 Conference* (pp. 161-168). New York, NY: ACM.

Taivalsaari, A. (2009). *Mashware: The future of web applications*. Technical Report. UMI Order Number: SERIES13103., Sun Microsystems Laboratories, Inc.

Taivalsaari, A. & Mikkonen, T. (2008). Mashups and modularity: Towards secure and reusable web applications. In Proceedings of *1st Workshop on Social Software Engineering and Applications* (pp. 25-33). L'Aquila, IEEE Computer Society Press.

Tufte, E. R. (1983). *The visual display of quantitative information*. Graphics Press.

Wong, J. & Hong, J. (2008). What do we "mashup" when we make mashups? In Proceedings of *the 4th International Workshop on End-User Software Engineering* (pp. 35-39). New York, NY, ACM.

Yu, J., Benatallah, B., Casati, F., & Daniel, F. (2008). Understanding mashup development. *IEEE Internet Computing*, *12*(5), 44-52.

Zang, N., Rosson, M. & Nasser, V. 2008. Mashups: Who? What? Why?. *In CHI '08 Extended Abstracts on Human Factors in Computing Systems* (pp. 3171-3176). New York, NY, ACM.

Özses, S. & Ergül, S. (2009). Cross-domain communications with JSONP. February 24, 2009. Retrieved June 2, 2011, from http://www.ibm.com/developerworks/library/wa-aj-jsonp1/.

Publication VII

Mashups in Web 3.0

A. Salminen

MASHUPS IN WEB 3.0

Arto Salminen
Department of Software Systems, Tampere University of Technology, P.O. Box 553, FI-33101, Tampere, Finland
arto.salminen@tut.fi

Keywords: Mashups, Web 3.0.

Abstract: Web has developed into a platform where applications live as services. This is referred to as Web 2.0. The next version, Web 3.0, refers to using the Web in a new way in new domains. In addition to realizing semantic web, Web 3.0 includes other advantageous concepts too. This paper discusses about mashups in Web 3.0 and describes how mashups will be an integral part of it. Moreover, we will point out some remarkable technical solutions that enable new kind of mashups and speculate about the time when these mashups can be fully implemented and realized.

1 INTRODUCTION

The way the Web is used has gone through significant changes. The Web 1.0 was a simple platform for browsing static documents that were connected with hyperlinks. Next version, Web 2.0, introduced user created content in a significant degree as well as collaboration between users. Communities and social networks as well as different services that enabled sharing videos, images and texts became popular. Mashups, web applications that integrate resources — i.e. the content created by users and enterprises — over the web were developed into a new breed of software that is widely utilized in different domains including mobile and desktop platforms. Currently, we are experiencing a paradigm shift towards web-based software (Taivalsaari, Mikkonen, Anttonen and Salminen, 2011), which consists of resources that can be located anywhere in the world and require no installation or manual updates. This has had a great effect not only on how software is used but also on development and deployment of software (Taivalsaari et al., 2011). In other words, what used to be a document browsing platform has become a means of communication with short messages and live audio, a place for music and video entertainment as well as a platform for full fledged applications such as text and spreadsheet editors and games. However, the evolution of the Web has not ceased, and as technical and other barriers are overcome it turns into version 3.0.

There is no clear definition for Web 3.0. Some use the term as a synonym for semantic web (Hendler, 2009). Others, however, think that Web 3.0 refers to new ways to use the web, and using it in new domains (Silva, Saleh, Rahman and El Saddik, 2008). Our perception of Web 3.0 is the latter. Similarly to Silva et al. (2008), we define Web 3.0 as "tomorrows web" that is ubiquitous and pervasive, which, in addition to semantic web, utilizes also other concepts such as ambient intelligence, smart interfaces and intelligent agents. In addition to these, in our view the Web 3.0 includes the concept of mashware, software created as a mashup as described in a technical report by Taivalsaari (2009). Mashware enables personalization and on-fly customization of web applications; ideally automatically or by the users.

As already pointed out, mashups combine content from more than one source into an integrated experience. This ability to aggregate content leverages the power of the Web to support worldwide sharing, accessing and reusing resources from different locations or in different contexts. Consequently, mashups, as well as mashware, demonstrate the capability of the Web to act as global-scale distribution channel for arbitrary distributed applications.

In this paper we discuss about mashups utilizing Web 3.0 concepts and technologies and present our view on what will happen in the future in the domain of mashups. In addition we describe forthcoming issues that are related to software mashupping using new web technologies. To gain better understanding about how Web 3.0 will benefit mashup developer, we start with a brief background study about the

concept.

This paper is structured as follows. In Section 2, we present the concepts that are included and discussed about under term "Web 3.0". In Section 3, we describe mashups that are enabled by Web 3.0 technologies and present examples of applications available today. Finally, in Section 4, we conclude the paper.

2 WEB 3.0 CONCEPT

We do not use term "Web 3.0" as a synonym for semantic web. Our perception of Web 3.0 contains not only the ideas introduced as "semantic web" but a lot more. Term "semantic web" refers to extending web documents so that information in them has clear meaning understandable for machines (Berners-Lee, Hendler and Lassila, 2001; Shadbolt, Berners-Lee and Hall, 2006). It can be used to create interoperable websites that make information exchange effortless. For example, semantic web has been successful in the domain of scientific publications (Das, Goetz, Girard and Clark, 2009). In our view, semantic web is an enabler for Web 3.0 applications but there are other aspects as well.

In the Web 3.0, the connection with the rest of the world via Internet is pervasive. It is available at everywhere for everyone at anytime. This has been achieved already as mobile terminals — that have become inexpensive enough for everyone to purchase — enable us to be online at all times, without interruption. However, as new web-enabled devices have been introduced, web applications can reach new fields of everyday devices. Game consoles, televisions and set top boxes already contain web applications and similar capabilities are spreading into cars, book reading devices and picture frames, for instance. Furthermore, progress of ubiquitous technologies (Weiser, 1991) makes everyday artefacts connected to the Internet, thus transforming their data into resources for applications.

Another interesting development related to pervasive computing is ambient intelligence, which is a slightly different concept. Ambient media or intelligence refers to systems that are unobtrusive, context aware, personalized, adaptive and anticipatory (Zelkha, Epstein, Birrell and Dodsworth, 1998). It emphasizes the ability of devices to communicate and make decisions independently without user interaction. Ambient media has been included in Web 3.0 concept by Silva et al. (2008). This provides another view to web as a pervasive platform used for making daily tasks easier.

Our view to Web 3.0 includes the concept of mashware (Taivalsaari, 2009) as well. In mashware, the idea of mashups has been expanded into software. Mashware is created from software components that are retrieved from all over the web and composed together without static linking or preprocessing. Realizing mashware in the full extend would allow really large-scale collaboration between developers as application components could be shared and reused without restrictions. Similarly, what has been achieved with current mashup tools (see a detailed summary in (Taivalsaari, 2009)), mashware applications could be developed possibly by the end-users themselves. Mashware, naturally, requires well-defined interfaces and we believe that a lot can be learnt from the work already done in the domain of semantic web.

Interestingly, Google's CEO Eric Schmidt described ideas similar to mashware when he gave his definition for Web 3.0 at Seoul Digital Forum in 2007. Schmidt's definition remarked that Web 3.0 applications will be pieced together, relatively small, able to be run on any device, fast and customizable, distributed via social networks and using data stored in the cloud. Certain characteristics, i.e. "pieced together", "fast and customizable" and "using data stored in the cloud", are features of mashware as well. However, according to Taivalsaari (2009), mashware is not limited to small applications, and distribution through social connections. Furthermore, mashware does not require universal-scale cross-platform compatibility even though this can be achieved with certain technologies.

Mashups rely on web services that are accessed through APIs. Therefore, mashups benefit greatly from recent developments called Open Web and Open API. The term Open Web refers to development promoted by Open Web Foundation, which is founded by major web organizations and aimed at promoting specifications that are royalty free and compatible with open licenses. This makes using different interfaces in co-operation easier. An interface following Open Web recommendations and available to be used by different parties is referred to as an Open API.

2.1 Enabling Technologies

Web 3.0 is driven by technological challenges that include implementing semantic web, expanding web browser capabilities, having reliable high bandwidth network connections, and linking physical world with the Web.

Semantic Web. Giving semantics for data refers to turning it into information that can be processed independently by machines. In the domain of the Web this means giving well-defined structure and meaning for data stored currently within unstruc-

tured and meaningless web documents. Languages and frameworks that can be used to achieve this already exist. RDF (Resource Description Framework), for instance, can be used with OWL (Web Ontology Language) or XML (eXtensible Markup Language) to describe the meaning of data within structured web documents.

In addition, there are solutions that can be used with the HTML, language currently used to describe most documents available in the Web. Microformats and microdata, the former described by the microformats community (*http://www.microformats.org*) and the latter introduced as part of HTML5 specification (Hickson, 2011), can be used to annotate the content with machine-readable labels. These solutions are lightweight, simple and evolutionary, in contrast to RDF used with OWL or XML, which can be described as revolutionary, but more complex and difficult to understand as well.

Web Browser Capabilities. Web browser performance has been increasing at unforeseen pace during last few years. Performance in JavaScript execution has skyrocketed thanks to development of powerful engines during ongoing JavaScript engine race started in 2008 between major browser vendors. In addition to code execution performance, also document rendering speed has evolved.

Two developments, HTML5 and WebGL, have significantly improved capabilities of web browser as a next generation application platform. HTML5 specification (originally named Web Applications 1.0) determines features that are typical for desktop applications and makes them available in browsers implemented as native features. These features include support for drag and drop, local data storage (offline functionality), drawing surface available for direct access by graphics hardware as well as video and audio playing capabilities. Even though HTML5 specification is still at the draft stage many features have already been implemented by browser vendors and included in stable versions of web browsers.

WebGL is a technology that enables hardware accelerated 3D graphics in a web browser. This is remarkable because it allows visually attractive games and other applications to be created without browser plug-ins. WebGL can be used for 2D graphics as well, and this enables developers to create graphics in procedural style without web browsers document object model (DOM), which is aimed at presenting static documents. WebGL specification has reached its first stable version number 1.0. Because of WebGL is a very low level interface, numerous higher level frameworks and libraries have been developed to make it easier and faster to create WebGL applications.

Cross-origin Resource Sharing. One of the significant problems in composing mashups has been web browsers poor capability to communicate across domains. Because of restricting security model of web browsers, also known as the Same Origin Policy, to be able to communicate directly with different web services one has need to use some cumbersome workarounds, such as dynamically inserted script elements and JSON with padding (JSONP). These workarounds are typically prone to security threats. Thanks to a recent specification called CORS (Cross-origin resource sharing) by W3C, mashup developers will be able to make cross-domain requests in similar fashion as the same domain requests. However, CORS needs to be implemented by service providers and it is currently supported by only a few services.

High Bandwidth Connections. High bandwidth Internet connections are already available for households and the price of subscribing for a high bandwidth connection in western countries has decreased. However, numerous areas exist where the prices are still relatively high. In addition to fixed Internet connections, mobile connections with fixed rate data plans have rapidly become common in western countries. According to the OECD's latest statistics (December 2011) there are 309 million fixed and 590 million mobile broadband subscriptions. Number of wireless subscriptions rose 26 % during last year whereas the number of fixed subscriptions was increased only by 5.8 %.

In spite of the increase in mobile subscriptions, mobile Internet is still often unreliable, and it suffers from different kinds of defects. For instance, mobile connections typically suffer from long latency times and issues on handover situations, which can be major shortcomings with certain types of applications. Luckily, new cellular broadband technologies are providing solutions for these and other technical issues.

Linking Physical World with the Web. Linking the physical world with web capable applications such as mashups can add a new dimension to the user experience of a mashup. Location-awareness has already been proven successful in mobile games, in which it has been found to be a very attractive feature (Korhonen, Saarenpää and Paavilainen, 2008). In addition to player location, physical artefacts have been incorporated into games as well (Reid, 2008). Similar idea would benefit mashups as well. However, linking not only location but also physical items into the mashup requires some infrastructure to be built. The infrastructure, however, does not need to be high-end technology, but simple 2D bar codes containing small amount of necessary information of the linkage may be enough.

3 MASHUPS IN WEB 3.0

Mashups will take advantage on developing web technologies and concepts introduced along Web 3.0. In the following, we discuss about how mashups benefit from Web 3.0 technologies today, in near and in distant future.

3.1 Mashups Today

Mobile Mashups. Most successful mobile mashups today are those used for communication with multiple instant messaging services. These are available for all mobile operating systems and used quite widely. Reasons behind this success of instant messaging mashups are, first the obvious need for applications of this kind caused by rivalling instant messaging service providers, and second the fact that mobile devices suit particularly well for communicating with other people. Another type of a successful mobile mashup is map-based mashups. Typically, these mashups show some additional location-related information on top of a map, for instance some mashups show other user locations on a map.

Pervasive Mashups. Mashups can be found in everyday devices as well. Some televisions, for instance, include media front-end applications that have capabilities to present videos from multiple web services. Furthermore, mashup for presenting weather information is another popular application in this kind of device.

HTML5 Mashups. Utilizing HTML5 in mashups is already possible. With HTML5 creating mashups is more straightforward. It enables using video and audio elements in mashups without the need for plug-ins. With HTML5's WebSockets it is possible to create real-time collaborative mashups, as they can be used for low latency bi-directional communication without the overhead caused by HTTP, which is used before for applications of this kind.

Using W3C's Geolocation API it is possible to create location-aware mashups. This enables some context-awareness such as location dependent searches and filtering. However, location accuracy gained with this technology is sometimes very poor. For instance, when travelling with a train and connecting to the web with the internal WLAN of the train, Geolocation usually points to one of the stations that may be on the other side of the country.

As HTML5 specification and other new APIs are relatively new and still at draft stage, it is necessary to create fallback mechanisms if it is desired to be sure that mashups relying on these techniques work for all users. This adds complexity of mashup architectures and implementations.

3.2 Mashups in Near Future

Some Web 3.0 technologies are already available outside research laboratories in commercial devices. We anticipate that mashups relying on such technologies will be available in the near future. In the following, we discuss about mashups that are built on these technologies.

WebGL Mashups. WebGL specification enables hardware accelerated 3D graphics in a web browser. This technology has not been yet utilized in mashups. 3D graphics could be well fitted into social mashups which would be used to communicate with other users in virtual spaces. Another way to use WebGL in mashups could be creating new visualizations for services, for instance video and picture services. Furthermore, 3D enables new ways to present complex data. For instance, a stock mashup that tries to compress enormous amount of information in one screen could benefit from using WebGL graphics to make the information easier to interpret.

Mashups Accessing Arbitrary Mobile Peripherals. Mashups with access to mobile device peripherals will become common in the near future. One domain of such mashups is augmented reality. Augmented reality mashup could access the camera of the device and add information related to the context of the user on top of the real time video. The information would be retrieved from web services, for instance a mashup could use a visual search service to provide photos and details of an unidentified plant or access Wikipedia to provide additional information about sights located around the user. Mashups of this kind would rely on accurate information about user location as well as device orientation.

Ambient Mashups. With ambient media, there are possibilities for even more advanced mashups. Using a mobile terminal to have effortless access to content relevant to the user context, and combining this context with the resources of the Web, can be very valuable for the user, especially if the mashup could work autonomously without explicit user input. Furthermore, if the mashup allows user collaboration, even richer user experiences can be provided.

Mashware. Another development in the near future will be the first fully mashware applications. Mashware will be combined from multiple components according to user's needs. For instance, a mashware video player could be constructed from components that are added according to which services the user

would like to access, what types of videos are played and what type of device the player is executed on. If, for instance, YouTube (*http://www.youtube.com/*) and Vimeo (*http://vimeo.com/*) services were used, the application would include interfaces only for those services and exclude interfacing components for Qik (*http://qik.com/*) and Yahoo Video (*http://screen.yahoo.com/*). The player could add the rendering component according to the video types so that opening a video in Ogg Theora, H.264 and Adobe Flash format would result in adding a corresponding component to the application. Furthermore, the user interface of the player would be added as a component so that a living room media centre would have different look and feel than a mobile version of the same application. These application components could be downloaded over the web from different services following a common interface specification. Naturally it is possible to provide a closed repository of components as well.

High-bandwidth Mashups. As mobile network connections get faster it will be possible to develop mashup applications that combine resources that require a lot of bandwidth. This can be for instance a high quality real time mobile television combined with related web content.

Mashups Embedded in Everyday Devices. In addition to televisions, other devices will have mashup applications as well. Mashups will be included in devices such as vehicles and game consoles. For instance a car navigator with a web connection could show live weather information and web camera images in addition to driving instructions. Furthermore, games can use mashups to add changing content into the game worlds. These mashups embedded in everyday devices will turn more advanced as well when the full potential of these specialized platforms is combined with web resources in imaginative ways.

Mashups in Games. In the future, mashups will be implemented within games as well. Web content from different sources can elaborate games in numerous ways. Dynamic content has already proven to be successful in games (Vanhatupa, 2009). For instance, game character's presence could be added with images and messages from players account on social networking services. Game worlds could include content from the real world, for example news casts could be played and real weather conditions could be reflected. Furthermore, games that are relying on user context, such as location-aware and augmented reality games, could naturally include some web content that would complement the experience.

Mashups Utilizing Microformats. Microformats, i.e. small details of information embedded in existing HTML documents, will be used in mashups as well. These lightweight semantics are already widely available, and using them in mashups is only a matter of time. Microformats are currently used by search engines to annotate search results. In mashups microformats could be used as an input when requesting relevant content. Another option where microformats are useful is content filtering.

3.3 Mashups in Distant Future

Some Web 3.0 technologies are not yet deployed in public, or they have some technical issues that need to be overcome before using them more widely. We anticipate that it will take some time for mashups relying on such technologies to emerge. In the following, we discuss about mashups that will be available in distant future as technical barriers are broken.

3D Mashware. As 3D web technologies have just reached somewhat stable stage, it will take a while for the first 3D mashware applications to become existence. Adding third dimension to graphics requires a new approach to user interface development and therefore existing user interface frameworks cannot be used. Furthermore, as the WebGL is a rather low level interface, creating mashware demands much attention to details. However, it is likely that existing higher level libraries and frameworks build for WebGL can be used as a stepping stone. Yet another technical restriction is the inability to use canvas element to render content formatted in HTML. This makes it hard to reuse the existing visual elements of the web that rely on Document Object Model (DOM).

Mashups Relying on Semantic Web. The full-fledged semantic web is still around the corner. Utilizing semantic web's full power in mashware will result in mashups that are capable of including new services and content types automatically. Mashups could benefit from semantic web's knowledge about relations of artefacts (and users) in the web and use it to provide information that is most relevant to the user.

Mashups Utilizing Large-scale Physical Infrastructure. Mashups that take advantage of physical infrastructure build in everyday devices will eventually be available, but because of required installations of hardware, this is not possible in the near future. However, mashups utilizing such physical connection can be really innovative and useful. For instance, Near-Field Communication (NFC) tags installed into a retail store could be used as an input for mashup providing price comparison information as well as other information about products such as possible use scenarios.

4 CONCLUSIONS

In this paper, we argued that mashups and mashware are at the core of what is known as Web 3.0, the next version of the web. Some mashups that utilize Web 3.0 technologies are already available and in wide use. In addition to describing some examples of today's Web 3.0 mashups we pointed out mashups and mashware applications that will likely be available in the near future. Furthermore, we discussed some more advantageous ideas that will take somewhat longer to appear.

From software engineering point of view mashups set an interesting challenge. Mashup design has been previously considered as ad hoc activity with minimal relation to software engineering practices, architecting or disciplined development (Hartmann, Doorley and Klemmer, 2008). However, our research has been focused on describing disciplined guidelines for mashup development (Salminen, Nyrhinen, Mikkonen and Taivalsaari, 2010) as well as general architecture for mashups (Mikkonen and Salminen, 2011).

As technical issues are being solved in accelerating pace, our position is that mashups and mashware will play an important role in how we will be using the Internet in Web 3.0 era.

REFERENCES

Berners-Lee, T., Hendler, J. and Lassila, O. (2001, May). The Semantic Web. *Scientific American*, 29-37.

Das, S., Goetz, M., Girard, L. and Clark, T. (2009). Scientific Publications on Web 3.0 (pp. 107-129). *Proceedings of 13th International Conference on Electronic Publishing*, Milan, Italy.

Hartmann, B., Doorley, S., Klemmer, S. (2008). Hacking, Mashing, Gluing: Understanding Opportunistic Design. *IEEE Pervasive Computing.* 7(3), 46-54.

Hendler, J. (2009). Web 3.0 Emerging. *IEEE Computer*, January, 42(1), 111-113.

Hickson, I. (2011). *HTML Microdata, Editors draft*. Retrieved March 12, 2012, from http://dev.w3.org/html5/md/

Korhonen, H., Saarenpää, H., and Paavilainen, J. (2008). Pervasive Mobile Games - A New Mindset for Players and Developers. *Proceedings of the 2nd International Conference on Fun and Games* (pp. 21-32). Springer-Verlag, Berlin,Heidelberg.

Mikkonen, T. & Salminen, A. (2011). Towards a Reference Architecture for Mashups (pp. 647-656). *Proceedings of the Second International Workshop on Variability, Adaptation and Dynamism in Software Systems and Services*. Berlin, Heidelberg, Springer-Verlag.

Reid, J., (2008). Design for Coincidence: Incorporating Real World Artifacts in Location-based Games. *Proceedings of the 3rd international conference on Digital Interactive Media in Entertainment and Arts* (pp. 18-25). ACM, New York, NY, USA.

Salminen, A., Nyrhinen, F., Mikkonen, T. and Taivalsaari, A. (2010). Developing Client-side Mashups: Experiences, Guidelines and the Road Ahead (pp. 161-168). *Proceedings of MindTrek 2010 Conference*. ACM, New York, NY, USA.

Silva, J., Saleh, A., Rahman, M. and El Saddik, A. (2008). Web 3.0: A Vision for Bridging the Gap between Real and Virtual. *Proceeding of the 1st ACM international workshop on Communicability design and evaluation in cultural and ecological multimedia system* (pp. 9-14). ACM, New York, NY, USA.

Shadbolt, N., Berners-Lee, T., and Hall, W. (2006). The Semantic Web Revisited. *IEEE Intelligent Systems* 21(3), 96-101.

Taivalsaari, A. (2009). Mashware: The Future of Web Applications. *Sun Microsystems Laboratories Technical Report TR-2009-181*.

Taivalsaari, A., Mikkonen, T., Anttonen, M., Salminen, A. (2011). The Death of Binary Software: End-User Software Moves to the Web (pp. 17-23). *Proceedings of the 9th International Conference on Creating, Connecting and Collaborating through Computing*, Kyoto, Japan.

Vanhatupa, J-M. (2009). Generative Approach for Extending Computer Role-playing Games at Run-time (pp. 177-188). *Proceedings of 11th Symposium on Programming Languages and Software Tools and 7th Nordic Workshop on Model Driven Software Engineering*, Tampere, Finland.

Weiser, M. (1991). The Computer for the 21st Century. *Scientific American,* 265(3), 3-11.

Zelkha, E., Epstein, B., Birrell, S., Dodsworth, C. (1998). From Devices to "Ambient Intelligence", *Digital Living Room Conference*.

Publication VIII

Implementing mobile mashware architecture: Downloadable components as on-demand services

T. Mikkonen and A. Salminen

Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 10 (2012) 553 – 560

Procedia Computer Science

The 9th International Conference on Mobile Web Information Systems

Implementing mobile mashware architecture: Downloadable components as on-demand services

Tommi Mikkonen[a*] and Arto Salminen[a]

[a]*Department of Software Systems, Tampere University of Technology, FI-33720 Tampere, Finland*

Abstract

The software industry is in the middle of a paradigm shift from desktop to mobile and web-based software. In the new era, applications increasingly live on the Web as services that lend themselves for runtime configuration. The associated delivery model, referred to on-demand software, or Software-as-a-Service (SaaS), implies that applications do not require installation or manual upgrades by the end users, as they are loaded on the fly. Furthermore, applications that build on resources offered by other applications, referred to as mashups, offer increasingly interesting opportunities. We believe that the trend towards using the web as an application platform will only strengthen in the future, and that instead of individual applications, it will also be possible to use application components in the same way we today download complete applications – in essence the Web is used as an architecture and transport for distributed applications, similarly to e.g. Corba at the level of principal idea. In this paper, we provide an overview for mashware computing, where downloadable components form applications in a piecemeal fashion, and enable rich access to resources of devices in a programmer-friendly way. Furthermore, applications can be easily built by components created by others and available as services. Finally, we demonstrate the capabilities of the concept with a sample application and discuss the lessons learned during the design process.

Keywords: Web applications; mashup applications, on-demand software; mashware

1. Introduction

The software is in the middle of a paradigm shift installable desktop systems towards mobile and web-based software. In the new era, applications live on the Web as services. The associated delivery model, referred to on-demand software, or Software-as-a-Service (SaaS), implies that applications no longer re-

* Corresponding author. Tel.: +358 40 8490 749.
E-mail address: tommi.mikkonen@tut.fi.

1877-0509 © 2012 Published by Elsevier Ltd.
doi:10.1016/j.procs.2012.06.071

quire installation or manual upgrades. While being envisioned for a long time, the facilities of the Web are fueling the transition in various forms, in part culminating in the possibility to run complete applications online. Examples of such systems include in particular business applications such as accounting, customer relationship management (CRM), enterprise resource planning (ERP), invoicing, human resource management (HRM), and content management (CM). In contrast, more traditional online applications for the public end users include games, email, as well as office appliances in the form of Google Docs.

Within the field of web applications, mashups – applications that build on resources offered by other applications, but combine them in an unforeseen fashion into an integrated experience – have become increasingly common. We believe that this trend, enabled by the fact that for the first time we have a global, uniform distribution channel, will only strengthen in the future, and that instead of individual applications using resources, it will also be possible to use application components in the same way we today download complete applications as well as access on-line resources. This will then further pave the way towards a component architecture that enables mixing and matching of pieces of software of different origins, as well as creates a market for 3rd party software components in the global scale. In particular, these facilities will be helpful in mobile setting, where usability issues, loading times and other restrictions have formed an obstacle for application development.

In this paper, we first provide an overview for our vision of mashware computing, originally presented in [13, 14], where downloadable components form applications in a piecemeal fashion and enable rich access to resources of devices in a programmer-friendly way. In this context, term resource is used in a wide sense, referring to real resources such as files, but also to services such as translation or rendering. As a technical contribution, we demonstrate the capabilities of the concept with a proof-of-concept implementation, which is the first concrete realization of the vision.

The rest of this paper is structured as follows. Section 2 discusses the background of the paper and provides an interview for the on-demand software delivery model and mashup development, with particular focus on modularity issues. Sections 3 and 4 form the core of this paper by revisiting the concept of mashware and by introducing in detail how we have implemented a mashware component that can be easily included in complete web applications. In Section 5, we provide an overview of an example application that has been constructed in accordance with mashware principles. In Section 6, we provide an extended discussion on the lessons learned when composing the implementation as well as on related systems that already exist. Towards the end of the paper, in Section 7 we draw the final conclusions of our work.

2. Background

The World Wide Web has undergone a number of phases to enable the development of on-demand software. Initially, web pages were little more than simple textual documents with limited user interaction capabilities, enabling only applications based on hyperlinks and full-page updates, which were rapidly enhanced with graphics support and form-based data entry. Gradually, with the introduction of DHTML [2] as well as numerous Rich Internet Application technologies, such as Adobe Flash and Microsoft Silverlight, it became feasible to create increasingly interactive web pages with built-in support for advanced graphics and animation, culminating in “Web 2.0” technologies, commonly associated with systems such as Ajax [1], Ruby on Rails [15], and Google Web Toolkit (GWT) [11]. Today, we are on our way towards more complex applications, that increasingly builds on the networking capabilities and pervasiveness of the web.

The ability to download systems on-demand basis produces many benefits over the conventional model of application deployment, where installation plays a major role. On-demand applications need not be installed by the users, and therefore they can be more flexibly upgraded than their desktop counterparts. Furthermore, since the same service is commonly made available to numerous users, configuration and

version management complications are considerably easier to solve than in the traditional model, where clients at least potentially use different versions. Finally, the service provider has access to all data as well as the behavior of the users, making it simpler to design exhaustive test systems that focus on the most important aspects of the system.

A particular characteristic of today's on-demand applications is that they are commonly loaded on-the-fly from the Web, and the browser acts as their runtime environment. Consequently, applications build on application programming interfaces (APIs) based on web technologies, including for example HTTP, REST, SOAP and JSON, which commonly lend themselves for other use as well. This characteristic enables the development of increasingly complex 3rd party applications that reuse the already existing content and services. Furthermore, these applications – so called mashups that "mash up" content from various sites into an integrated experience – can be build by developers that are not directly associated with the original developers of the reused services.

As already pointed out, a mashup is a web site or application that combines content from more than one source (from multiple web sites) into an integrated experience. In other words, mashups are content aggregates that leverage the power of the Web to support worldwide sharing of content that conventionally would not have been easily accessible or reusable in different contexts or from different locations.

While present mashup applications often build on maps or photos with overlays, this is not a technical restriction. In contrast, the content can be anything as long as it can be meaningfully combined with other information available on the Web, e.g., price comparison information combined with product specifications, latest product news and user reviews or blogs. In addition to so called consumer mashups, mashups are used in enterprises as well to combine private information of the company with publicly available services. The key aspect is that the content must be available in a format that can be reused easily in other contexts. Furthermore, interfaces that remain unchanged over time are needed in order to develop long-lasting, robust mashup services that do not need constant upgrades as services they build on are modified.

Despite the recent emergence of tools and the general interest and hype around web technologies, mashups are not actually anything new. In software development, it has been a common practice or desire to build more advanced software systems out of prefabricated, reusable components developed by other software developers. The desire for reusable software components was first expressed by McIlroy and other participants of the NATO Software Engineering conference back in 1968 [5], and techniques for software reuse have been investigated for decades. However, mashup development differs from conventional software reuse in several important ways, originally listed in [13, 14]:

- In mashup development there is a lot more focus on reusing the *content* rather than the *implementation* of a web site. While standardized formats for various content formats, such as images and videos for example, exist, it is often surprisingly difficult to reuse the implementation of a web site in other contexts. For example, the current web technologies do not make it easy to specify which parts of the web site are intended to be reusable in other contexts and which are not. In the same fashion, many mashups reuse the *visual representation* of sites only (e.g., a map or the layout of a web site), while others reuse the content (substance) separately from its visual representation.
- Mashups are far more dynamic than more conventionally used compiled, binary software components. Since mashups are all about combining content from multiple web sites in a highly dynamic fashion, they cannot be built easily with static programming languages that require advance compilation and static type checking.
- Because of the increased focus on content rather than on implementation techniques, the mashup developer base is different from conventional software development projects. A mashup developer does not necessarily have any formal training or background in software development. Rather, it is far more common for them to have some kind of a media background.
- The distribution and sharing power of the Web makes it exceptionally easy to reuse content in unforeseen, unexpected ways. Basically, anything that is made available on the Web is instantly accessible to

anybody anywhere in the world with a web browser. This increases the potential content user and reuser base by several orders of magnitude compared with conventional software components that are typically distributed in a far more controlled and limited fashion. Often, the developer of a web site may not be aware at all that content from his or her site is being used in other contexts as well. The same also applies to implementations, but due to the above complications their reuse becomes more complex in practice.

Despite the above issues, mashups demonstrate the capability of the Web to act as a distribution channel for arbitrary applications, making it the first truly pervasive and uniform communication media for distributed applications. Consequently it can be used as a basis for a component architecture on top of which distributed applications can be build in the global scale.

3. Towards software components as on-demand-service

In recent years, significant progress has been made in turning web engineering into a real engineering field; for a comprehensive overview, the reader is referred to [3]. However, we argue that the development practices for web applications are still far from the maturity levels of traditional software development, and there is still an impedance mismatch between web-based software development and software engineering [7].

We believe that the evolution of web technologies will eventually lead us to mashware – mashup software that leverages source code and software components that are downloaded dynamically from all over the world. Such software can dramatically improve the productivity of software development, allowing massive reuse of software components across the planet. However, without new software architectures, methodologies and systematic approaches towards the development of such software, the gap between web development and software engineering will only grow wider. To avoid this, research is needed in several areas related to web development, including security, modularity, and legal aspects, as well as improved software engineering methodologies to foster the development of mashware systems in systematic fashion.

In the absence of security restrictions that prevent the downloading of executable code to the browser from different domains, the content in mashups could be executable code as well. In fact, we believe that mashware – or on-demand software components – is the next logical step in the evolution of the Web as a software platform. By this, we refer to a generalized form of mashup-based software development, in which applications can be composed by dynamically combining code and other content originating from web sites from all over the world. For instance, the user interface widgets of an application might be downloaded from one site, storage features from another site, the localization capabilities from a third site, and so on, based on the availability of best components for each purpose. We refer to such a model for application development as mashware – software as a worldwide mashup. The general idea, originally presented in [7, 13, 14] at the level of concept, is depicted in Fig. 1. In the figure, we assume that the developer is building a web application to visualize stock market information. The application consists of a main application – downloaded from the developer's own web server – that will dynamically download the other necessary components from other web sites. These components include: (1) widget library for presenting the user interface of the application, (2) stock graph visualization library for creating graphs, (3) stock quote / market data interface available from a third site, and (4) localization (L10N) components for customizing the data and the language for a specific country. All components are downloaded from different web servers and used dynamically without compilation, static linking, or explicit installation.

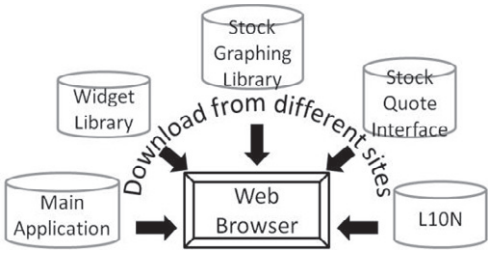


Fig. 1. The mashware concept: software components as on-demand services that can be downloaded on the fly.

For obvious reasons, one cannot expect that the downloaded components could be arbitrary. Instead, in order to be use the different components for their tasks, interfaces in these components must be designed in a disciplined fashion, preferably following the software engineering principles that would allow their evolution over time. However, this task can be simplified, since unlike conventional binary applications, web applications are generally deployed in textual form, using representations such as HTML, XML, CSS and JavaScript source code, causing a shift towards dynamic languages [10]. This results in increased flexibility in application creation, as well as tolerance towards certain kinds of problems.

Finally, mashware based applications are particularly promising in the mobile domain, where it is common that applications can considerably benefit from the possibility to use hardware acceleration, binding to binary resources, and other forms of tighter integration with the device itself. With the proposed approach, it is possible to create, deploy, and distribute components that are capable of interacting with the device in an enhanced fashion. For applications, this results in improved capabilities and superior user experience, enabled by the new bindings.

4. Proof-of-concept implementation

Our mashware implementation follows a general mashup reference architecture described in an earlier paper [6]. The reference architecture (see Fig. 2a) determines the following main components for a mashup application: mashup manager, content providers, data model including content extractors and formatters, mashup creation and renderers. We have applied the reference architecture to a mashware implementation (see Fig. 2b for application structure) that is based on a RESTful mashware metadata repository and a client application that includes the capability to combine its necessary software components on the fly. Furthermore, on the client side, we have a mashware manager that includes functionality for searching, downloading and enabling components originating from the repository. The repository includes components for extracting, formatting and rendering the mashup content. In addition, the client-side application includes a mashup creation module that determines the business logic of the mashup and combines the components acquired with the mashware manager in a meaningful way. All these components are summarized in technical terms in the following.

Mashware repository. The mashware repository, which can be generalized into a set of repositories, has a RESTful interface [4] for requesting metadata of mashware components. The interface is self-descriptive, and components can be requested based on their capabilities. For instance, a list of all possible renderers is located in an URI *http://example.com/renderers*, and rendering components capable of presenting content with 'image/jpeg' MIME type can be searched with an HTTP GET request *http://example.com/renderers[contentTypes='imagejpeg']*. Similar structure applies for content extractors and formatters, too. The component metadata is stored as JavaScript objects described in JSON (JavaScript Object Notation).

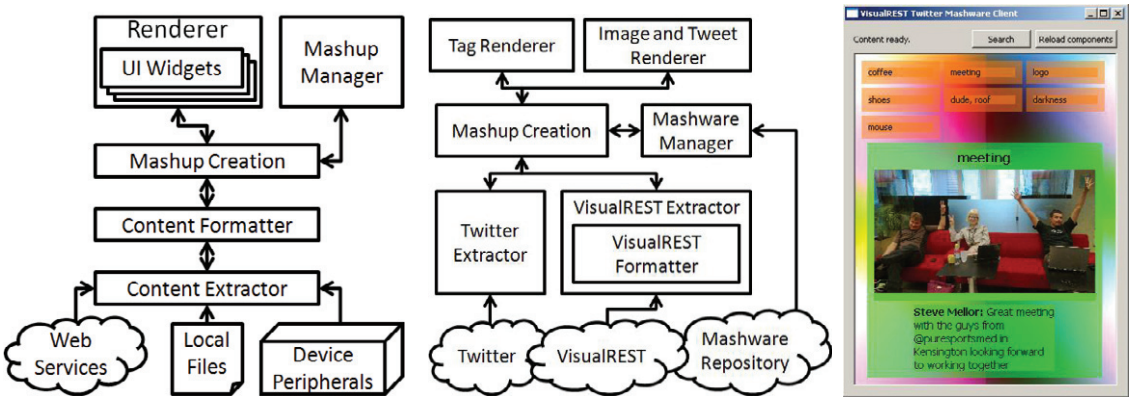


Fig. 2. (a) General reference architecture for mashups [6]; (b) Example mashware application structure; (c) A screenshot of the example application running on a desktop environment. Implementation of the mashware application follows the general reference architecture.

Mashware manager. The mashware manager includes functionality that is needed for searching, downloading, and selecting components from the repository. In our implementation, we have used only a single manager, but in accordance to the overall scheme, also this component could be replaced by another, loaded dynamically as needed as well. Caching of the components can be implemented in mashware manager, and it can be used to extend components with some methods if it is necessary. In our implementation, when the mashup creation module requests a component, the mashware manager downloads the matching code, evaluates the code, creates a new instance of the JavaScript component, initializes and returns it. Furthermore, it is possible to extend the system towards more autonomous selection of components.

Content extractors. Content extractors provide means to download content from different sources over the web, and they can be located in the repository based on their description. These can include data in the web, local data in the device as well as context dependent data, such as GPS location. Collaborative mashups can be composed, as extractors can be used to include content created simultaneously in another mashup.

Content formatters. Content formatters are used to change and remove content items provided by content extractors before they are forwarded to renderers. In our implementation, a content formatter can be used to expand a content provider to provide the data in a new format, but they can be used as a standalone component as well. Furthermore, it is possible to use web services to alter the data. For instance, a content formatter could use MashReduce programming model [12] in order to make heavy operations for content data on the server-side.

Renderers. Renderers are used to present the content items for the user. They are not limited to present the content in the fashion we are used to on the original services. For instance, a renderer could display the Flickr images in a 3D scene of an art exhibition room, thus creating richer user experience and potentially more attractive way to consume content.

Mashup creation. The mashup creation part is programmed by a mashup developer to combine the mashware components in a meaningful way. As it uses the mashware manager, the technical implementation can be very straightforward, and it simply passes the data from a component to another according to desired business logic. Introducing a simple graphical tool to determine the mashup creation algorithm is included in our future plans.

To overcome numerous implementation details that affect the development inside the browser, including in particular browsers' security model which is eschewed for real applications [13], our design is based on a mobile Qt-based JavaScript framework called Lively for Qt (*http://lively.cs.tut.fi/qt*) as a runtime environment on the client-end of our mashware system, similarly to our earlier paper [9]. However, the runtime does not form an essential restriction in the experiment, and consequently other environments could be used as well.

5. Example application

The example mashware application can be seen in Fig. 2c. The idea of the application is to download tagged images from VisualREST system (*http://visualrest.cs.tut.fi*), which can be used to access the content in user's handheld devices and social networking services [8]. Tags are rendered on the user interface and when a tag is selected the tagged image content is shown on another rendering component. Selecting a tag triggers a search where the tag is used as a keyword to popular Twitter micro blogging service (*http://twitter.com*). Found Twitter entries ("tweets") are displayed along with the images in the same rendering component. If there is more than one image or tweet to display, they are changed in regular intervals.

The Fig. 2 presents the example application structure. The application utilizes two content extractors to gain access to VisualREST and Twitter content. A content formatting component is used to alter the data downloaded from VisualREST to be more straightforward to handle. Moreover, the application uses two types of rendering components: one to present tags and another to display images and tweets. The rest of the user interface (i.e. buttons and the drawing surface) is merged into the mashup creation module.

6. Discussion

At present, there are some widely used services that enable the development of web applications as mashware, but they are commonly available only in a very specialized, degenerated form. The most obvious example in Google Maps API (*http://code.google.com/apis/maps/index.html*), which is available for use in a programmatic fashion and that is partly loaded in the application that wishes to use the Maps API. However, in many regards Google Maps also defines the capabilities of the whole application, and takes over almost all activities that can be associated with the map component. Consequently we regard Google Maps more of an application framework for map-based applications rather than true downloadable component. Still, we believe that similar services, targeted for different domains, are a necessity for generalizing the approach of this paper into a true ecosystem that can be used for real applications.

An obvious concern about the development of downloadable applications is the lack of available and applicable standards. Consequently, while building on well-established services, their APIs are almost exclusively proprietary and vendor-specific. For example, assuming that one would wish to replace Google Maps with another service in the above example, it would most likely be a time-consuming task to redesign the application around OpenStreetMaps API (*http://wiki.openstreetmap.org/wiki/API*), for example. To create a true component market, standards that define the interfaces to the different services should be created and deployed en masse online. Examples of such standards can be found in the field of agent systems (*http://www.fipa.org/repository/standardspecs.html*), where interoperability of agents (or components) produced by different parties is a key issue.

Finally, while security issues form a major concern in the design of mashup applications, similar considerations are needed in the development of mashware based system. In this paper, we were using a proof-of-concept implementation based on proprietary technology, which liberated us from such concerns, but when run inside the standard browser security concerns similar to those presented in [7, 13, 14] can be raised. We believe that the new, emerging standards such as World Wide Web Consortium's Security

Activity Proposal (*http://www.w3.org/2011/07/security-activity.html*) will eventually solve these issues. However, at the same time we expect that even eventually when finalized, the deployment of standard-compliant implementations requires time.

7. Conclusions

The World Wide Web is the most powerful medium for sharing information in the history of human-kind. For the first time, we have a truly uniform distribution system that can be used for the development of applications in the global scale. Consequently, we believe that applications that exist now have only scratched the surface of the true potential of applications that are created in accordance with the principles of distributed computing using a component model based on the characteristics of the Web.

In this paper, we have addressed an architecture where the Web is used as a basis for a distributed component architecture. A proof-of-concept system created for a mobile device was also introduced, together with a sample application that demonstrates the power and flexibility of the approach. In the long run, we plan to experiment with the approach in a more extensive fashion, with the particular research interest lying in the creation of an ecosystem where reusable components are available online en masse.

References

1. D. Crane, E. Pascarello, D. James, Ajax in Action, Manning Publications (2005).
2. D. Goodman, Dynamic HTML: The Definitive Reference, O'Reilly Media (2006).
3. G. Kappel, B. Pröll, S. Reich and W. Retschitzegger (eds.), Web Engineering: The Discipline of Systematic Development of Web Applications, John Wiley and Sons (2006).
4. Masse, M. REST API Design Rulebook. O'Reilly Media (2012).
5. M.D. McIlroy, Mass produced software components, In P. Naur, B. Randell (eds.), 1968 NATO Working Conference on Software Engineering, Garmisch, Germany, October 7-11, pp. 88-98 (1968).
6. T. Mikkonen and A. Salminen, Towards a reference architecture for mashups, In proc. of the int. conference on On the move to meaningful internet systems, Springer-Verlag, Berlin, Heidelberg, pp. 647-656 (2011).
7. T. Mikkonen, and A. Taivalsaari, The mashware challenge: Bridging the gap between web development and software engineering, 18th ACM SIGSOFT Int. Symposium on the Foundations of Software Engineering FSE-18, Santa Fe, NM, USA, pp. 245-249 (2010).
8. N. Mäkitalo, H. Peltola, J. Salo and T. Turto, VisualREST: A Content Management System for Cloud Computing Environment, In Proc. of the SEAA'2011 - 37th EUROMICRO Conference on Software Engineering and Advanced Applications, pp. 183-188 (2011).
9. F. Nyrhinen, A. Salminen, T. Mikkonen and A. Taivalsaari, Lively Mashups for Mobile Devices. In Proc. of the First Int. Conference on Mobile Computing, Applications and Services, San Diego, CA, USA, October 26-29, 2009.
10. L.D. Paulson, Developers Shift to Dynamic Programming Languages, IEEE Computer, February, pp. 12-15 (2007).
11. C. Prabhakar, Google Web Toolkit: GWT Java Ajax Programming, Packt Publishing (2007).
12. J. Salo, T. Aaltonen and T. Mikkonen, MashReduce – Server-Side Mashups for Mobile Devices, In Proc. of the 6th International Conference on Grid and Pervasive Computing, Oulu, Finland, May 11-13 (2011).
13. A. Taivalsaari, Mashware: The Future of Web Applications, Sun Microsystems Laboratories Technical Report TR-2009-181, February (2009).
14. A. Taivalsaari and T. Mikkonen, Mashups and Modularity: Towards Secure and Reusable Web Applications, In Proc. of the First Workshop on Social Software Engineering and Applications, L'Aquila, Italy, September 16, (2008).
15. B. Tate, Ruby on Rails: Up and Running. O'Reilly Media, (2006).