ICWE 2016 Rapid Mashup Challenge: Introduction

Florian Daniel[1] and Martin Gaedke[2]

[1] Politecnico di Milano, Via Ponzio 34/5, 20133 Milano, Italy
florian.daniel@polimi.it
[2] Technische Universität Chemnitz, Str. der Nationen 62, 09111 Chemnitz, Germany
martin.gaedke@informatik.tu-chemnitz.de

Abstract. The ICWE 2016 Rapid Mashup Challenge is the second installment of a series of challenges that aim to engage researchers and practitioners in showcasing and discussing their work on assisting mashup development. This introduction provides the reader with the general context of the Challenge, its objectives and motivation, and the requirements contributions were asked to satisfy so as to be eligible for participation. A summary of the contributions that were selected for presentation in the 2016 edition anticipates the content of the remainder of this volume.

Keywords: Mashups, Mashup tools, Challenge, Benchmarking

1 Context and Objective

Mashups, that is "composite applications developed starting from reusable data, application logic and/or user interfaces typically, but not mandatorily, sourced from the Web" [1], have been the subject of research and industrial study for several years by now. Over the same time span, we also witnessed an iterative specialization of the term: while in the beginning all types of applications developed by reusing resources from the Web were referred to as "mashups," today – also thanks to the pioneering efforts by the mashup community – this kind of integration of Web-accessible resources has become common practice in software engineering and can, hence, no longer be considered a practice that is easy to isolate from software engineering in general. Today, the term is instead more focused on those applications that are developed with the help from a so-called *mashup tool* (possibly ranging from full-fledged integrated development environments to dedicated programming libraries) that aims to ease mashup development via suitable abstractions and automations. Thanks to the respective simplified development processes, often mashups are further associated with *end users* that not necessarily possess programming knowledge but that nevertheless may want to or be required to develop own situational applications, e.g., to automate tasks in their everyday business.

In line with last year's edition [2], the ICWE 2016 Rapid Mashup Challenge[1] acknowledges this peculiarity of mashups and puts its focus on the techniques,

[1] http://challenge.webengineering.org/

approaches, libraries, and tools that researchers and practitioners have come up with so far to aid the development of mashups – to all types of users and/or programmers. This perspective is different from the perspectives of similar challenges known from other contexts or communities. For instance, the Semantic Web Challenge[2] focuses on the application of Semantic Web [3] technologies in the development of software with commercial potential, large user bases, or functionality that is useful and of societal value. The AI Mashup Challenge[3], instead, more specifically focuses on mashups that use AI (Artificial Intelligence) technology (e.g., machine learning and data mining, machine vision, natural language processing, reasoning, ontologies) and intelligence to mashup existing resources. The Rapid Mashup Challenge, instead, does not limit its focus to any specific technology and rather aims to understand how mashups can be developed, independently of how their internals look like.

The purpose of the Challenge is further that of comparing the proposed development approaches with each other, so as to stimulate the interchange of ideas among researchers and practitioners and to cross-fertilize them. Yet, objectively comparing approaches that are as diverse as mashup approaches is a hard endeavor. In fact, while last year we used (i) an explicit feature checklist to be filled by authors to assess the expressive power of the proposed approaches and (ii) a common mashup development scenario to assess the elegance and ease of development, this year we decided to leave proponents more freedom in showcasing their approach and did not impose any specific development scenario. The reason for this choice is that the proposals we received this year[4] were characterized by very diverse levels of maturity, ranging from production-ready instruments to proof-of-concept prototypes. Requiring all of them to support one and a same development scenario would have meant precluding the less mature approaches from participating, as they simply would not have been able to provide the necessary development support to implement a full-fledged mashup. Since we however felt that also the less mature works proposed ideas and solutions that had merits and deserved presentation, we opted for a less rigorous comparison in favor of more variety and a better representation of the latest state of the art in mashup development assistance.

In the following, we summarize the structure of the Challenge and the feature checklist. Then we briefly introduce the approaches that were selected for presentation in the 2016 edition of the Challenge, while the papers in the remainder of this volume (extensions of the initial participation proposals) explain each of the approaches in more detail and describe the respective demonstrations given live during the Challenge.

2 Structure of Challenge

The Challenge was again organized into four phases:

[2] http://challenge.semanticweb.org/
[3] http://aimashup.org/
[4] http://challenge.webengineering.org/program/

1. **Admission**: Submission of applications. Each application had to include a brief description of the proposed tool/approach and a filled feature checklist, so as to allow the organizers to pre-select proposals based on topic and interestingness.
2. **Preparation**: If a proposal was accepted to the challenge, the authors could use the time from the notification till the Challenge to prepare an as effective as possible demonstration of their approach. This preparation phase gave the authors almost six weeks to prepare for the event.
3. **Competition**: During the Challenge, participants had to give both a presentation and a live demonstration of how to build a mashup – both within a maximum of 20 minutes. Last year, we strictly limited the time of the live demonstration to 10 minutes, giving all participants the same time for the development of a same mashup. Since this year we did without the reference mashup, we allowed participants to use their time as best as they felt.
4. **Post-challenge**: Preparation of post-challenge paper explaining the proposed solution and giving technical details about the approach and how it was used to rapidly build their demonstration mashup.

The goal of this structure is to have authors focus more on the practical aspects before the Challenge (the preparation of their demonstration), while asking them to concentrate on the conceptual and scientific aspects afterwards (with the writing of the paper to be included in this volume).

3 Feature Checklist

In order to facilitate the comparison of approaches, authors were required to accompany their submission with a filled feature checklist that describes the two key parts of the evaluation, i.e., the nature of the mashups that their tool/approach allows one to develop and the development features of the proposed tool. The following subsections describe the features in more detail.

3.1 Mashup Features

In order to be able to compare the mashups produced by the different approaches during the Challenge, the mashup features proposed by Daniel and Matera [1] were taken as reference:

- **Mashup type:** The mashup type expresses the positioning of the mashup at one or more of the three layers of the typical application stack (data, logic, presentation), depending on where the mashup's integration logic is positioned. *Data mashups* operate at the data layer, integrate data sources, and are typically published again as data sources (e.g., RSS feeds or RESTful Web services). *Logic mashups* integrate components at the application logic layer, reuse data and application logic (e.g., Web services), and are typically published as Web services. *User Interface (UI) mashups* are located at the

presentation layer, integrate UI components/widgets, and are published as Web applications that users can interact with via the Web browser. Finally, *hybrid mashups* span multiple layers of the application stack.
- **Component types:** The types of mashups introduced above strongly relate to the types of the components they integrate. *Data components* comprise RSS and Atom feeds, XML, JSON, CSV and similar data resources, web data extractions, micro-formats, but also SOAP or RESTful web services that are used as data services only. *Logic components* comprise SOAP and RESTful web services, JavaScript APIs and libraries, device APIs, and API extractions. *UI components* comprise code snippets and JavaScript UI libraries, Java portlets, widgets and gadgets, web clips and extracted UI components.
- **Runtime location:** There are generally a variety of possible architectural configurations that may be adopted for the development of mashups, compatibly with the requirements of the chosen components. *Client-side* mashups are executed in the client browser. *Server-side* mashups are executed in the server. *Client-server* mashups are distributed over client and server, and both parts interact the one with the other at runtime.
- **Integration logic:** The integration logic tells how integration happens, that is, how components are used to form a composite application and how they are enabled to communicate with each other (if at all). *UI-based integration* applies exclusively to UI components and uses the graphical layout of the mashup's user interface to render UI components in parallel next to each other inside one or more web pages. *Orchestrated integration* applies to all kinds of components and consists in a centralized composition logic. *Choreographed integration* is for all those types of components that are able to comply with a given convention (oftentimes also called a contract or protocol), so as to manage integration without a central coordinator.
- **Instantiation lifecycle:** The last aspect of mashups considered is how long an instantiated mashup is running. *Stateless mashups* do not require keeping any internal state for their execution and end after processing. *Short-living* mashups are mashups that last the time of a user session, i.e., as long as a user is interacting with the mashup in the client browser, and terminate with the closing of the client browser. *Long-living* mashups may last longer than a user session, that is, they survive even after the user closes the browser with the rendered mashup or after the first invocation of the mashup.

These five features allow one to easily classify mashups and to assess their internal complexity. Of course, this is not an exhaustive list of characteristics and many other distinguishing features could be examined [1]. Yet, for the sake of assessing the suitability and interestingness of approaches we considered these five features as enough.

3.2 Mashup Tool Features

The comparison of the features of the mashup tools/approaches was instead based on the work by Aghaee et al. [4].

- **Targeted end-user:** Determining which group of users is targeted by a mashup tool/approach is a strategic design issue decided by the developers. *Non-programmers* do not have programming skills. Yet, they may be interested in creating mashups as long as it does not require them to learn and use a programming language. *Local developers* are those non-programmers who usually have advanced knowledge in computer tools. *Expert programmers* have adequate programming skills and experience to develop mashups using programming and scripting languages (e.g., JavaScript and PHP).
- **Automation degree:** The automation degree of a mashup tool refers to how much of the development process can be undertaken by the tool on behalf of its users. *Full automation* of mashup development eliminates the need for direct involvement of users in the development process. *Semi-automatic* tools partially automate mashup development by providing guidance and assistance. *Manual* approaches do not provide any automated support during development; typically, these approaches come in the form of programming libraries or runtime middlewares.
- **Liveness level:** Tanimoto proposed the concept of liveness [5], according to which four levels of liveness can be distinguished. At *Level 1* (non-executable prototype mockup), a tool is just used to create prototype mashups that are not directly connected to any kind of run-time system. *Level 2* (explicit compilation and deployment steps) of liveness is characterized by mashup design blueprints that carry sufficient details to give them an executable semantics. *Level 3* (automatic compilation and deployment) tools support rapid deployment into operation, e.g., triggered by each edit-change or by an explicit action executed by the developer. *Level 4* (dynamic modification of running mashup) of mashup liveness is obtained by the tools that support live modification of the mashup code, while it is being executed.
- **Interaction technique:** There have been a number of interaction techniques through the use of which the barriers of programming can be lifted to its developers [6]. *Editable examples* let users modify and change the behavior of existing examples, instead of programming from scratch. In *form-based* interaction, users are asked to fill out a form to create a new or change the behavior of an existing object. *Programming by demonstration* suggests to teach a computer by example how to accomplish a particular task. *Spreadsheets* are one of the most popular and widely used end-user programming approaches to store, manipulate, and display complex data. *Textual DSLs* are languages targeted to address specific problems in a particular domain; they have a textual syntax that may or may not resemble an existing general-purpose programming language. A *visual language* (iconic), as opposed to a textual programming language, is any programming language that uses visual symbols, syntax, and semantics. Some visual languages support *wiring with implicit control flow*, where the control flow of the mashup is derived from its data flow graph. Other visual languages support *wiring with explicit control flow*, where the control flow is explicitly defined, for instance, by adding directed arrows connecting the boxes, or putting the boxes in a specific order (e.g., from left to right). *WYSIWYG* (What You See Is What

You Get) enables users to create and modify a mashup on a graphical user interface that is similar to the one that will appear when the mashup runs. *Natural language* allows developers to express their mashup via a restricted, controlled set of natural language constructs (e.g., a subset of English) that can be interpreted unequivocally by a runtime environment.
- **Online user community:** Online communities are an important resource in assisting developers, especially end-users, to program [7]. If a tool does not support any online community (*none*), it is harder to leverage on the experience of others. In *public* communities, the content is accessible to any user on the Web who wishes to join the community (with or without registration). In *private* communities, the authority to join the community is granted on the basis of compliance with some operator-specified criteria.

Like for the mashup features, also in the case of the mashup tools/approaches many other characteristics could be considered (e.g., collaboration). The features selected for the Challenge, however, already provide good insight into the philosophy behind each approach, and we preferred to keep the list concise.

4 Participants

The following contributions were selected for participation in the Challenge[5]:

- *FlexMash 2.0 – Flexible Modeling and Execution of Data Mashups* by Pascal Hirmer and Michael Behringer: FlexMash is a data mashup tool that aims at facilitating the integration and processing of heterogeneous, dynamic data sources. It targets domain experts, features a graphical pipes and filter modeling paradigm, and supports the enforcement of non-functional requirements like security and robustness. The first version of the tool was presented during the 2015 edition of the Challenge; the new version comes with cloud-based execution and human interaction during runtime.
- *The SmartComposition Approach for Creating Environment-Aware Multi-Screen Mashups* by Michael Krug, Fabian Wiedemann, Markus Ast and Martin Gaedke: The SmartComposition approach is a UI mashup framework that supports local developers (non-experts) in creating environment-aware multi-screen mashups by leveraging on HTML markup only. Supported Web components can range from data sources to components that provide access to the Web of Things, e.g., to control actuators and access sensors. Mashup execution across multiple screens is enabled using a messaging service based on WebSockets.
- *Linked Widgets Platform for Rapid Collaborative Semantic Mashup Development* by Tuan-Dat Trinh, Ba-Lam Do, Peter Wetz, Peb Ruswono Aryan,

[5] We would like to thank Michael Luggen and Eduard Daoud for participating in the Challenge with their presentations of, respectively, the Uduvudu Editor and search-based mashup development. It's a pity that, due to time constraints, we were not able to include a long version of their proposals in these post-challenge proceedings.

Elmar Kiesling and A Min Tjoa: The Linked Widgets platforms is a mashup platform that combines the Semantic Web and mashups to help users integrate data and make informed decisions in decision making processes. The tool is based on a semantic model of mashup components that enables the automation of some typical data integration tasks, such as overcoming data heterogeneity and data exploration. In addition, the Linked Widgets platform supports a live, collaborative mashup development and execution model able to easily bring together multiple stakeholders.

- *End-User Development for the Internet of Things: EFESTO and the 5Ws composition paradigm* by Giuseppe Desolda, Carmelo Ardito and Maristella Matera: EFESTO is another tool that was presented as well in 2015. This new version comes with a novel rule-based composition paradigm (exhibiting similarities with the well-known IFTTT) that provides also for the composition of so-called smart objects, i.e., components that encapsulate sensors and/or actuators accessible via the Internet of Things. The described work targets end-users via a dedicated visual rule composition notation.
- *Toolet: an editor for Web-based tool appropriation by hobby programmers* by Jeremías P. Contell and Oscar Díaz: Toolet is an editor for Web appropriation, that is, for the ad-hoc adaptation of third-party Web applications to the needs of users performed by the users themselves. The level of abstraction to enable users to integrate and manipulate data proposed by Toolet is an original one based on Google Spreadsheets. The adaptation of Web applications is then based on Web augmentation techniques, which also cater for hobby programmers. Toolet is one of the early prototypes included in this volume.
- *On the Role of Context in the Design of Mobile Mashups* by Valerio Cassani, Stefano Gianelli, Maristella Matera, Riccardo Medana, Elisa Quintarelli, Letizia Tanca and Vittorio Zaccaria: This contribution introduced CAMUS, a design methodology and an accompanying platform for the design and fast development of Context-Aware Mobile mashUpS. The approach revolves around the concept of context to effectively cater to situational needs of users, while the target mashups are mobile applications. Internally the platform makes use of adaptable model-driven engineering techniques. The presented tool is an early prototype of the envisioned platform.

Table 1 summarizes the characteristics of the selected approaches as declared by the authors. Compared to last year, it is evident that the Internet of Things has percolated into the presented approaches, and most of the proposals aim at supporting hybrid mashups, featuring integration logics stemming from the data, logic and UI layers. The strong focus on end-users without significant programming skills is confirmed also this year, as is – in line with this observation – the focus on graphical development paradigms (ranging from editable examples to iconic and WYSIWYG paradigms) and dynamic, live (level 4) and automatic (level 3) development approaches.

Together, this selection of mashup approaches provides an intriguing snapshot of the current state of the art in research on mashup development aids.

Table 1: Overview of the mashup and mashup tool features declared by the approaches that participated in the ICWE 2016 Rapid Mashup Challenge.

			FlexMash 2.0	SmartComposition	Linked Widgets	Toolet	EFESTO	CAMUS
Mashup	**Mashup type**	Data mashups	✓				✓	✓
		Logic mashups						
		UI mashups						✓
		Hybrid mashups		✓	✓	✓		
	Component types	Data components	✓	✓	✓	✓	✓	✓
		Logic components		✓	✓	✓	✓	✓
		UI components		✓	✓	✓		✓
	Runtime location	Client-side only				✓		
		Server-side only					✓	
		Client-server	✓	✓	✓			✓
	Integration logic	UI-based integr.						
		Orchestration	✓		✓	✓	✓	✓
		Choreography		✓	✓			
	Instantiation lifecycle	Stateless	✓					
		Short-living		✓	✓			
		Long-living			✓	✓	✓	✓
Mashup tool	**Target end-user**	Local developers		✓			✓	✓
		Non-programmers	✓		✓			
		Expert programmers				✓		
	Automation degree	Full automation			✓			✓
		Semi-automation	✓		✓	✓	✓	
		Manual		✓	✓			
	Liveness level	Level 1 (mockup)						
		Level 2 (manual)				✓		
		Level 3 (automatic)	✓		✓			✓
		Level 4 (dynamic)		✓			✓	
	Interaction technique	Editable examples		✓				
		Form-based				✓		
		Progr. by demonstration						
		Spreadsheets				✓		
		Textual DSL						
		Visual (iconic)	✓				✓	✓
		Visual (wiring, implicit)			✓			
		Visual (wiring, explicit)				✓		
		WYSIWYG						✓
		Natural language						
	Online user community	None	✓	✓		✓	✓	✓
		Private						
		Public			✓			

Some proposals are already very mature and close to production systems (e.g., FlexMash 2.0, SmartComposition, Linked Widgets, search-based mashups, and EFESTO), while others are still in an early stage of development (e.g., Toolet and CAMUS). Yet, they all provide good insight into the research questions and technological trends researchers are intrigued by right now and that still ask for good questions before we can say that mashup development is properly assisted for all kinds of target developers.

We are confident that the reader will find the remainder of this volume, which provides detailed insight into the introduced approaches, as intriguing as we do.

References

1. Daniel, F., Matera, M.: Mashups: Concepts, Models and Architectures. Springer (2014)
2. Daniel, F., Pautasso, C., eds.: Rapid Mashup Development Tools - First International Rapid Mashup Challenge, RMC 2015, Rotterdam, The Netherlands, June 23, 2015, Revised Selected Papers. Volume 591 of Communications in Computer and Information Science., Springer (2016)
3. Berners-Lee, T., Hendler, J., Lassila, O.: The semantic web. Scientific American (May 2001) 34–43
4. Aghaee, S., Nowak, M., Pautasso, C.: Reusable decision space for mashup tool design. In Barbosa, S.D.J., Campos, J.C., Kazman, R., Palanque, P.A., Harrison, M.D., Reeves, S., eds.: EICS, ACM (2012) 211–220
5. Tanimoto, S.L.: Viva: A visual language for image processing. Journal of Visual Languages & Computing **1**(2) (1990) 127–139
6. Myers, B.A., Ko, A.J., Burnett, M.M.: Invited research overview: end-user programming. In: CHI'06 extended abstracts on Human factors in computing systems, ACM (2006) 75–80
7. Nardi, B.A.: A small matter of programming: perspectives on end user computing. MIT press (1993)